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

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>List of Tables</strong></td>
<td>x</td>
</tr>
<tr>
<td><strong>List of Figures</strong></td>
<td>xiv</td>
</tr>
<tr>
<td><strong>1 General</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Scope</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Normative references</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Terms and definitions</td>
<td>2</td>
</tr>
<tr>
<td>1.4 Implementation compliance</td>
<td>5</td>
</tr>
<tr>
<td>1.5 Structure of this International Standard</td>
<td>5</td>
</tr>
<tr>
<td>1.6 Syntax notation</td>
<td>6</td>
</tr>
<tr>
<td>1.7 The C++ memory model</td>
<td>6</td>
</tr>
<tr>
<td>1.8 The C++ object model</td>
<td>7</td>
</tr>
<tr>
<td>1.9 Program execution</td>
<td>8</td>
</tr>
<tr>
<td>1.10 Multi-threaded executions and data races</td>
<td>11</td>
</tr>
<tr>
<td>1.11 Acknowledgments</td>
<td>15</td>
</tr>
<tr>
<td><strong>2 Lexical conventions</strong></td>
<td>17</td>
</tr>
<tr>
<td>2.1 Separate translation</td>
<td>17</td>
</tr>
<tr>
<td>2.2 Phases of translation</td>
<td>17</td>
</tr>
<tr>
<td>2.3 Character sets</td>
<td>18</td>
</tr>
<tr>
<td>2.4 Preprocessing tokens</td>
<td>19</td>
</tr>
<tr>
<td>2.5 Alternative tokens</td>
<td>20</td>
</tr>
<tr>
<td>2.6 Tokens</td>
<td>21</td>
</tr>
<tr>
<td>2.7 Comments</td>
<td>21</td>
</tr>
<tr>
<td>2.8 Header names</td>
<td>21</td>
</tr>
<tr>
<td>2.9 Preprocessing numbers</td>
<td>22</td>
</tr>
<tr>
<td>2.10 Identifiers</td>
<td>22</td>
</tr>
<tr>
<td>2.11 Keywords</td>
<td>23</td>
</tr>
<tr>
<td>2.12 Operators and punctuators</td>
<td>23</td>
</tr>
<tr>
<td>2.13 Literals</td>
<td>24</td>
</tr>
<tr>
<td><strong>3 Basic concepts</strong></td>
<td>34</td>
</tr>
<tr>
<td>3.1 Declarations and definitions</td>
<td>34</td>
</tr>
<tr>
<td>3.2 One definition rule</td>
<td>36</td>
</tr>
<tr>
<td>3.3 Scope</td>
<td>39</td>
</tr>
<tr>
<td>3.4 Name lookup</td>
<td>45</td>
</tr>
<tr>
<td>3.5 Program and linkage</td>
<td>59</td>
</tr>
<tr>
<td>3.6 Start and termination</td>
<td>62</td>
</tr>
<tr>
<td>3.7 Storage duration</td>
<td>65</td>
</tr>
<tr>
<td>3.8 Object lifetime</td>
<td>69</td>
</tr>
<tr>
<td>3.9 Types</td>
<td>73</td>
</tr>
<tr>
<td>3.10 Lvalues and rvalues</td>
<td>78</td>
</tr>
<tr>
<td>3.11 Alignment</td>
<td>80</td>
</tr>
</tbody>
</table>

© ISO/IEC N4296
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4</td>
<td>The <code>asm</code> declaration</td>
</tr>
<tr>
<td>7.5</td>
<td>Linkage specifications</td>
</tr>
<tr>
<td>7.6</td>
<td>Attributes</td>
</tr>
<tr>
<td>8</td>
<td>Declarators</td>
</tr>
<tr>
<td>8.1</td>
<td>Type names</td>
</tr>
<tr>
<td>8.2</td>
<td>Ambiguity resolution</td>
</tr>
<tr>
<td>8.3</td>
<td>Meaning of declarators</td>
</tr>
<tr>
<td>8.4</td>
<td>Function definitions</td>
</tr>
<tr>
<td>8.5</td>
<td>Initializers</td>
</tr>
<tr>
<td>9</td>
<td>Classes</td>
</tr>
<tr>
<td>9.1</td>
<td>Class names</td>
</tr>
<tr>
<td>9.2</td>
<td>Class members</td>
</tr>
<tr>
<td>9.3</td>
<td>Member functions</td>
</tr>
<tr>
<td>9.4</td>
<td>Static members</td>
</tr>
<tr>
<td>9.5</td>
<td>Unions</td>
</tr>
<tr>
<td>9.6</td>
<td>Bit-fields</td>
</tr>
<tr>
<td>9.7</td>
<td>Nested class declarations</td>
</tr>
<tr>
<td>9.8</td>
<td>Local class declarations</td>
</tr>
<tr>
<td>9.9</td>
<td>Nested type names</td>
</tr>
<tr>
<td>10</td>
<td>Derived classes</td>
</tr>
<tr>
<td>10.1</td>
<td>Multiple base classes</td>
</tr>
<tr>
<td>10.2</td>
<td>Member name lookup</td>
</tr>
<tr>
<td>10.3</td>
<td>Virtual functions</td>
</tr>
<tr>
<td>10.4</td>
<td>Abstract classes</td>
</tr>
<tr>
<td>11</td>
<td>Member access control</td>
</tr>
<tr>
<td>11.1</td>
<td>Access specifiers</td>
</tr>
<tr>
<td>11.2</td>
<td>Accessibility of base classes and base class members</td>
</tr>
<tr>
<td>11.3</td>
<td>Friends</td>
</tr>
<tr>
<td>11.4</td>
<td>Protected member access</td>
</tr>
<tr>
<td>11.5</td>
<td>Access to virtual functions</td>
</tr>
<tr>
<td>11.6</td>
<td>Multiple access</td>
</tr>
<tr>
<td>11.7</td>
<td>Nested classes</td>
</tr>
<tr>
<td>12</td>
<td>Special member functions</td>
</tr>
<tr>
<td>12.1</td>
<td>Constructors</td>
</tr>
<tr>
<td>12.2</td>
<td>Temporary objects</td>
</tr>
<tr>
<td>12.3</td>
<td>Conversions</td>
</tr>
<tr>
<td>12.4</td>
<td>Destructors</td>
</tr>
<tr>
<td>12.5</td>
<td>Free store</td>
</tr>
<tr>
<td>12.6</td>
<td>Initialization</td>
</tr>
<tr>
<td>12.7</td>
<td>Construction and destruction</td>
</tr>
<tr>
<td>12.8</td>
<td>Copying and moving class objects</td>
</tr>
<tr>
<td>12.9</td>
<td>Inheriting constructors</td>
</tr>
<tr>
<td>13</td>
<td>Overloading</td>
</tr>
<tr>
<td>13.1</td>
<td>Overloadable declarations</td>
</tr>
<tr>
<td>13.2</td>
<td>Declaration matching</td>
</tr>
</tbody>
</table>
13.3 Overload resolution ........................................ 303
13.4 Address of overloaded function ............................. 323
13.5 Overloaded operators ........................................ 325
13.6 Built-in operators .......................................... 329
14 Templates ........................................................................ 333
14.1 Template parameters ............................................ 334
14.2 Names of template specializations ............................ 337
14.3 Template arguments ............................................ 339
14.4 Type equivalence ................................................ 344
14.5 Template declarations .......................................... 345
14.6 Name resolution .................................................. 364
14.7 Template instantiation and specialization ............... 379
14.8 Function template specializations .......................... 392
15 Exception handling ...................................................... 414
15.1 Throwing an exception ........................................ 415
15.2 Constructors and destructors ................................. 417
15.3 Handling an exception ......................................... 417
15.4 Exception specifications ....................................... 419
15.5 Special functions .............................................. 424
16 Preprocessing directives ............................................. 426
16.1 Conditional inclusion .......................................... 427
16.2 Source file inclusion .......................................... 429
16.3 Macro replacement ............................................ 430
16.4 Line control ..................................................... 435
16.5 Error directive .................................................. 435
16.6 Pragma directive ............................................... 435
16.7 Null directive ................................................... 436
16.8 Predefined macro names ...................................... 436
16.9 Pragma operator ................................................. 437
17 Library introduction ................................................... 438
17.1 General ......................................................... 438
17.2 The C standard library ........................................ 439
17.3 Definitions ..................................................... 439
17.4 Additional definitions ......................................... 442
17.5 Method of description (Informative) ...................... 442
17.6 Library-wide requirements .................................. 448
18 Language support library ........................................... 468
18.1 General ......................................................... 468
18.2 Types ......................................................... 468
18.3 Implementation properties .................................. 469
18.4 Integer types .................................................. 478
18.5 Start and termination ........................................ 479
18.6 Dynamic memory management ............................ 481
18.7 Type identification ............................................ 487
18.8 Exception handling ........................................... 489
18.9 Initializer lists .................................................. 494

Contents v
18.10 Other runtime support ............................................... 495

19 **Diagnostics library** ................................................. 498
    19.1 General .......................................................... 498
    19.2 Exception classes .............................................. 498
    19.3 Assertions ...................................................... 502
    19.4 Error numbers .................................................. 502
    19.5 System error support ........................................... 502

20 **General utilities library** ........................................ 514
    20.1 General .......................................................... 514
    20.2 Utility components .............................................. 514
    20.3 Pairs ............................................................. 519
    20.4 Tuples ........................................................... 523
    20.5 Compile-time integer sequences ................................ 533
    20.6 Class template **bitset** ...................................... 534
    20.7 Memory ........................................................... 541
    20.8 Smart pointers .................................................. 556
    20.9 Function objects ................................................. 582
    20.10 Metaprogramming and type traits .............................. 602
    20.11 Compile-time rational arithmetic ............................... 622
    20.12 Time utilities .................................................. 625
    20.13 Class template **scoped_allocator_adaptor** ............... 640
    20.14 Class **type_index** ........................................... 646

21 **Strings library** .................................................... 648
    21.1 General .......................................................... 648
    21.2 Character traits ............................................... 648
    21.3 String classes .................................................. 654
    21.4 Class template **basic_string** ............................... 658
    21.5 Numeric conversions ............................................ 684
    21.6 Hash support .................................................... 686
    21.7 Suffix for **basic_string** literals .............................. 686
    21.8 Null-terminated sequence utilities ............................. 686

22 **Localization library** .............................................. 690
    22.1 General .......................................................... 690
    22.2 Header `<locale>` synopsis .................................... 690
    22.3 Locales .......................................................... 691
    22.4 Standard **locale** categories ................................ 703
    22.5 Standard code conversion facets ............................... 742
    22.6 C library locales ................................................. 743

23 **Containers library** .............................................. 745
    23.1 General .......................................................... 745
    23.2 Container requirements ......................................... 745
    23.3 Sequence containers ............................................ 774
    23.4 Associative containers ......................................... 805
    23.5 Unordered associative containers ............................... 823
    23.6 Container adaptors ............................................. 842
### Iterators library

24.1 General ................................................. 852
24.2 Iterator requirements .................................... 852
24.3 Header `<iterator>` synopsis ............................... 857
24.4 Iterator primitives ....................................... 860
24.5 Iterator adaptors .......................................... 864
24.6 Stream iterators .......................................... 877
24.7 Range access ............................................. 883
24.8 Container access .......................................... 884

### Algorithms library

25.1 General .................................................. 886
25.2 Non-modifying sequence operations ....................... 897
25.3 Mutating sequence operations ............................ 902
25.4 Sorting and related operations ........................... 910
25.5 C library algorithms ...................................... 922

### Numerics library

26.1 General .................................................. 924
26.2 Numeric type requirements ................................ 924
26.3 The floating-point environment ............................ 925
26.4 Complex numbers .......................................... 926
26.5 Random number generation ................................ 936
26.6 Numeric arrays ........................................... 980
26.7 Generalized numeric operations ......................... 1001
26.8 C library .................................................. 1004

### Input/output library

27.1 General .................................................. 1009
27.2 Iostreams requirements ................................... 1010
27.3 Forward declarations ...................................... 1010
27.4 Standard iostream objects ................................. 1012
27.5 Iostreams base classes .................................... 1014
27.6 Stream buffers ............................................ 1032
27.7 Formatting and manipulators .............................. 1041
27.8 String-based streams ..................................... 1069
27.9 File-based streams ........................................ 1079

### Regular expressions library

28.1 General .................................................. 1095
28.2 Definitions ............................................... 1095
28.3 Requirements ............................................. 1096
28.4 Header `<regex>` synopsis ............................... 1098
28.5 Namespace std::regex_constants .......................... 1105
28.6 Class `regex_error` ...................................... 1108
28.7 Class template `regex_traits` ............................ 1108
28.8 Class template `basic_regex` ............................. 1112
28.9 Class template `sub_match` ............................... 1117
28.10 Class template `match_results` ......................... 1123
28.11 Regular expression algorithms ......................... 1128
28.12 Regular expression iterators ............................ 1133
28.13 Modified ECMAScript regular expression grammar

29 Atomic operations library
29.1 General
29.2 Header <atomic> synopsis
29.3 Order and consistency
29.4 Lock-free property
29.5 Atomic types
29.6 Operations on atomic types
29.7 Flag type and operations
29.8 Fences

30 Thread support library
30.1 General
30.2 Requirements
30.3 Threads
30.4 Mutual exclusion
30.5 Condition variables
30.6 Futures

A Grammar summary
A.1 Keywords
A.2 Lexical conventions
A.3 Basic concepts
A.4 Expressions
A.5 Statements
A.6 Declarations
A.7 Declarators
A.8 Classes
A.9 Derived classes
A.10 Special member functions
A.11 Overloading
A.12 Templates
A.13 Exception handling
A.14 Preprocessing directives

B Implementation quantities

C Compatibility
C.1 C++ and ISO C
C.2 C++ and ISO C++ 2003
C.3 C++ and ISO C++ 2011
C.4 C++ and ISO C++ 2014
C.5 C standard library

D Compatibility features
D.1 Increment operator with bool operand
D.2 register keyword
D.3 Implicit declaration of copy functions
D.4 Dynamic exception specifications
D.5 C standard library headers
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alternative tokens</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>Identifiers with special meaning</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>Keywords</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>Alternative representations</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>Types of integer literals</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>Escape sequences</td>
<td>27</td>
</tr>
<tr>
<td>7</td>
<td>String literal concatenations</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>Relations on <code>const</code> and <code>volatile</code></td>
<td>78</td>
</tr>
<tr>
<td>9</td>
<td><code>simple-type-specifiers</code> and the types they specify</td>
<td>160</td>
</tr>
<tr>
<td>10</td>
<td>Relationship between operator and function call notation</td>
<td>308</td>
</tr>
<tr>
<td>11</td>
<td>Conversions</td>
<td>316</td>
</tr>
<tr>
<td>12</td>
<td>Value of folding empty sequences</td>
<td>353</td>
</tr>
<tr>
<td>13</td>
<td>Library categories</td>
<td>438</td>
</tr>
<tr>
<td>14</td>
<td>C++ library headers</td>
<td>448</td>
</tr>
<tr>
<td>15</td>
<td>C++ headers for C library facilities</td>
<td>449</td>
</tr>
<tr>
<td>16</td>
<td>C++ headers for freestanding implementations</td>
<td>450</td>
</tr>
<tr>
<td>17</td>
<td><code>EqualityComparable</code> requirements</td>
<td>451</td>
</tr>
<tr>
<td>18</td>
<td><code>LessThanComparable</code> requirements</td>
<td>451</td>
</tr>
<tr>
<td>19</td>
<td><code>DefaultConstructible</code> requirements</td>
<td>451</td>
</tr>
<tr>
<td>20</td>
<td><code>MoveConstructible</code> requirements</td>
<td>451</td>
</tr>
<tr>
<td>21</td>
<td><code>CopyConstructible</code> requirements (in addition to <code>MoveConstructible</code>)</td>
<td>451</td>
</tr>
<tr>
<td>22</td>
<td><code>MoveAssignable</code> requirements</td>
<td>452</td>
</tr>
<tr>
<td>23</td>
<td><code>CopyAssignable</code> requirements (in addition to <code>MoveAssignable</code>)</td>
<td>452</td>
</tr>
<tr>
<td>24</td>
<td><code>Destructible</code> requirements</td>
<td>452</td>
</tr>
<tr>
<td>25</td>
<td><code>NullablePointer</code> requirements</td>
<td>454</td>
</tr>
<tr>
<td>26</td>
<td><code>Hash</code> requirements</td>
<td>454</td>
</tr>
<tr>
<td>27</td>
<td>Descriptive variable definitions</td>
<td>455</td>
</tr>
<tr>
<td>28</td>
<td>Allocator requirements</td>
<td>455</td>
</tr>
<tr>
<td>29</td>
<td>Language support library summary</td>
<td>468</td>
</tr>
<tr>
<td>30</td>
<td>Header <code>&lt;stddef&gt;</code> synopsis</td>
<td>468</td>
</tr>
<tr>
<td>31</td>
<td>Header <code>&lt;limits&gt;</code> synopsis</td>
<td>478</td>
</tr>
<tr>
<td>32</td>
<td>Header <code>&lt;cfloat&gt;</code> synopsis</td>
<td>478</td>
</tr>
<tr>
<td>33</td>
<td>Header <code>&lt;stdlib&gt;</code> synopsis</td>
<td>479</td>
</tr>
<tr>
<td>34</td>
<td>Header <code>&lt;setjmp&gt;</code> synopsis</td>
<td>496</td>
</tr>
<tr>
<td>35</td>
<td>Header <code>&lt;signal&gt;</code> synopsis</td>
<td>496</td>
</tr>
<tr>
<td>36</td>
<td>Header <code>&lt;stdalign&gt;</code> synopsis</td>
<td>496</td>
</tr>
<tr>
<td>37</td>
<td>Header <code>&lt;stdarg&gt;</code> synopsis</td>
<td>496</td>
</tr>
<tr>
<td>38</td>
<td>Header <code>&lt;stdbool&gt;</code> synopsis</td>
<td>496</td>
</tr>
<tr>
<td>39</td>
<td>Header <code>&lt;stdlib&gt;</code> synopsis</td>
<td>496</td>
</tr>
<tr>
<td>40</td>
<td>Header <code>&lt;ctime&gt;</code> synopsis</td>
<td>497</td>
</tr>
</tbody>
</table>
Diagnoses library summary ........................................ 498
Header `<cassert>` synopsis ....................................... 502
Header `<cerrno>` synopsis ......................................... 503
General utilities library summary .................................. 514
Header `<cstdlib>` synopsis ......................................... 555
Header `<cstring>` synopsis ......................................... 555
Primary type category predicates .................................. 607
Composite type category predicates ................................. 608
Type property predicates .......................................... 608
Type property queries ............................................. 615
Type relationship predicates ...................................... 616
Const-volatile modifications .................................... 617
Reference modifications .......................................... 618
Sign modifications ................................................. 618
Array modifications ................................................ 619
Pointer modifications ............................................. 619
Other transformations ............................................ 620
Expressions used to perform ratio arithmetic ..................... 624
Clock requirements ................................................. 628
Header `<ctime>` synopsis .......................................... 640
Strings library summary ........................................... 648
Character traits requirements .................................... 649
basic_string(const Allocator&) effects ........................ 663
basic_string(const basic_string&) effects ...................... 663
basic_string(const basic_string&, size_type, size_type, const Allocator&) effects 663
basic_string(const charT*, size_type, const Allocator&) effects 664
basic_string(const charT*, const Allocator&) effects ......... 664
basic_string(size_t, charT, const Allocator&) effects ........ 664
basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects 665
operator=(const basic_string&) effects ........................ 665
operator=(basic_string&&) effects ................................ 665
compare() results ................................................... 678
Potential mbstate_t data races ................................... 688
Header `<cctype>` synopsis ........................................ 688
Header `<cwctype>` synopsis ....................................... 688
Header `<cwchar>` synopsis ........................................ 688
Header `<cstdlib>` synopsis ....................................... 689
Header `<cstring>` synopsis ....................................... 689
Header `<ctime>` synopsis ......................................... 689
Localization library summary .................................... 690
Locale category facets ........................................... 694
Required specializations .......................................... 695
do_in/do_out result values ...................................... 712
do_unshift result values ......................................... 713
Integer conversions ............................................... 716
Length modifier .................................................... 717
Integer conversions ............................................... 721

List of Tables xi
List of Figures

1  Expression category taxonomy ............................................. 78
2  Directed acyclic graph ......................................................... 244
3  Non-virtual base ................................................................. 245
4  Virtual base ................................................................. 246
5  Virtual and non-virtual base ................................................. 246
6  Name lookup ............................................................... 248
7  Stream position, offset, and size types [non-normative] ............. 1009
1 General

1.1 Scope

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

1.2 Normative references

The following referenced documents are indispensable for the application 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) With the qualifications noted in Clauses 18 through 30 and in C.5, the C standard library is a subset of the C++ standard library.
1.3 Terms and definitions

For the purposes of this document, the following definitions apply.

17.3 defines additional terms that are used only in Clauses 17 through 30 and Annex D.

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

1.3.1 argument
<function call expression> expression in the comma-separated list bounded by the parentheses

1.3.2 argument
<function-like macro> sequence of preprocessing tokens in the comma-separated list bounded by the parentheses

1.3.3 argument
<throw expression> the operand of throw

1.3.4 argument
<template instantiation> expression, type-id or template-name in the comma-separated list bounded by the angle brackets

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

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

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

1.3.7 dynamic type
<glvalue> type of the most derived object (1.8) to which the glvalue denoted by a glvalue expression refers

[Example: if a pointer (8.3.1) p whose static type is “pointer to class B” is pointing to an object of class D, derived from B (Clause 10), the dynamic type of the expression *p is “D.” References (8.3.2) are treated similarly. — end example]

1.3.8 dynamic type
<prvalue> static type of the prvalue expression

§ 1.3
ill-formed program
program that is not well formed

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

1.3.11 implementation limits
restrictions imposed upon programs by the implementation

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

1.3.13 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: The extended character set is a superset of the basic character set (2.3). — end note ]

1.3.14 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

1.3.15 parameter
<function-like macro> identifier from the comma-separated list bounded by the parentheses immediately following the macro name

1.3.16 parameter
<template> template-parameter

1.3.17 signature
<function> name, parameter type list (8.3.5), and enclosing namespace (if any)
[ Note: Signatures are used as a basis for name mangling and linking. — end note ]

1.3.18 signature
<function template> name, parameter type list (8.3.5), enclosing namespace (if any), return type, and template parameter list

§ 1.3
1.3.19 signature
signature
<function template specialization> signature of the template of which it is a specialization and its template arguments (whether explicitly specified or deduced)

1.3.20 signature
signature
<class member function> name, parameter type list (8.3.5), class of which the function is a member, cv-qualifiers (if any), and ref-qualifier (if any)

1.3.21 signature
signature
<class member function template> name, parameter type list (8.3.5), class of which the function is a member, cv-qualifiers (if any), ref-qualifier (if any), return type, and template parameter list

1.3.22 signature
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)

1.3.23 static type
static type
type of an expression (3.9) resulting from analysis of the program without considering execution semantics
[Note: 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]

1.3.24 undefined behavior
undefined behavior
behavior for which this International Standard imposes no requirements
[Note: Undefined behavior may be expected when this International Standard 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. —end note]

1.3.25 unspecified behavior
unspecified behavior
behavior, for a well-formed program construct and correct data, that depends on the implementation
[Note: The implementation is not required to document which behavior occurs. The range of possible behaviors is usually delineated by this International Standard. —end note]

1.3.26 well-formed program
well-formed program
C++ program constructed according to the syntax rules, diagnosable semantic rules, and the One Definition Rule (3.2).
1.4 Implementation compliance

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

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

(2.1) — If a program contains no violations of the rules in this International Standard, a conforming implementation shall, within its resource limits, accept and correctly execute that program.

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

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

3 For classes and class templates, the library Clauses specify partial definitions. Private members (Clause 11) are not specified, but each implementation shall supply them to complete the definitions according to the description in the library Clauses.

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

5 The names defined in the library have namespace scope (7.3). A C++ translation unit (2.2) obtains access to these names by including the appropriate standard library header (16.2).

6 The templates, classes, functions, and objects in the library have external linkage (3.5). The implementation provides definitions for standard library entities, as necessary, while combining translation units to form a complete C++ program (2.2).

7 Two kinds of implementations are defined: a hosted implementation and a freestanding implementation. For a hosted implementation, this International Standard 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 (17.6.1.3).

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

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

1.5 Structure of this International Standard

1 Clauses 2 through 16 describe the C++ programming language. That description includes detailed syntactic specifications in a form described in 1.6. For convenience, Annex A repeats all such syntactic specifications.

2 Clauses 18 through 30 and Annex D (the library clauses) describe the Standard C++ library. That description includes detailed descriptions of the templates, classes, functions, constants, and macros that constitute the library, in a form described in Clause 17.

2) “Correct execution” can include undefined behavior, depending on the data being processed; see 1.3 and 1.9.

3) This documentation also defines implementation-defined behavior; see 1.9.
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 International Standard, 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.

1.6 Syntax notation

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

\{ expression_{opt}\}

indicates an optional expression enclosed in braces.

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

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

1.7 The C++ 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 (2.3) and the eight-bit code units of the Unicode UTF-8 encoding form and is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the low-order bit; the most significant bit is called the high-order bit. The memory available to a C++ program consists of one or more sequences of contiguous bytes. Every byte has a unique address.

[Note: The representation of types is described in 3.9. —end note]

A memory location is either an object of scalar type or a maximal sequence of adjacent bit-fields all having non-zero 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 (1.10) can update and 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 non-zero width. —end note]
[Example: A structure declared as

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

contains four separate memory locations: The field 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]

1.8 The C++ object model [intro.object]

The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An object is a region of storage. [Note: A function is not an object, regardless of whether or not it occupies storage in the way that objects do. — end note] An object is created by a definition (3.1), by a new-expression (5.3.4) or by the implementation (12.2) when needed. The properties of an object are determined when the object is created. An object can have a name (Clause 3). An object has a storage duration (3.7) which influences its lifetime (3.8). An object has a type (3.9). The term object type refers to the type with which the object is created. Some objects are polymorphic (10.3); 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 (Clause 5) used to access them.

Objects can contain other objects, called subobjects. A subobject can be a member subobject (9.2), a base class subobject (Clause 10), or an array element. An object that is not a subobject of any other object is called a complete object.

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

(3.1) — If x is a complete object, then x is the complete object of x.

(3.2) — Otherwise, the complete object of x is the complete object of the (unique) object that contains x.

If a complete object, a data member (9.2), 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.

Unless it is a bit-field (9.6), a most derived object shall have a non-zero 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 (3.9) shall occupy contiguous bytes of storage.

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 that are not bit-fields may have the same address if one is a subobject of the other, or if at least one is a base class subobject of zero size and they are of different types; otherwise, they shall have distinct addresses.4

[Example:

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

4) 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 (1.9).]
Note: C++ provides a variety of fundamental types and several ways of composing new types from existing
types (3.9). — end note

1.9 Program execution

The semantic descriptions in this International Standard define a parameterized nondeterministic abstract
machine. This International Standard 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.  

Certain aspects and operations of the abstract machine are described in this International Standard as
implementation-defined (for example, \text{sizeof(int)}). These constitute the parameters of the abstract ma-
ishine. Each implementation shall include documentation describing its characteristics and behavior in these
respects. 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 International Standard
as unspecified (for example, evaluation of expressions in a \text{new-initializer} if the allocation function fails to
allocate memory (5.3.4)). Where possible, this International Standard 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 International Standard as undefined (for example, the effect
of attempting to modify a \text{const} object). [Note: This International Standard 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 International
Standard places no requirement on the implementation executing that program with that input (not even
with regard to operations preceding the first undefined operation).

If a signal handler is executed as a result of a call to the \text{raise} function, then the execution of the handler is
sequenced after the invocation of the \text{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]

An instance of each object with automatic storage duration (3.7.3) 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).

The least requirements on a conforming implementation are:

\begin{itemize}
  \item Access to volatile objects are evaluated strictly according to the rules of the abstract machine.
  \item 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 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{itemize}

This provision is sometimes called the “as-if” rule, because an implementation is free to disregard any requirement of this
International Standard 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.

This documentation also includes conditionally-supported constructs and locale-specific behavior. See 1.4.
These collectively are referred to as the *observable behavior* of the program. [Note: More stringent correspondences between abstract and actual semantics may be defined by each implementation. — end note]

[Note: Operators can be regrouped according to the usual mathematical rules only where the operators really are associative or commutative.\(^7\) For example, in the following fragment

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

the expression statement behaves exactly the same as

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

due to the associativity and precedence of these operators. Thus, the result of the sum \((a + 32760)\) is next added to \(b\), and that result is then added to \(5\) which results in the value assigned to \(a\). On a machine in which overflows produce an exception and in which the range of values representable by an `int` is \([-32768,+32767]\), the implementation cannot rewrite this expression as

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

since if the values for \(a\) and \(b\) were, respectively, -32754 and -15, the sum \(a + b\) would produce an exception while the original expression would not; nor can the expression be rewritten either as

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

or

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

since the values for \(a\) and \(b\) might have been, respectively, 4 and -8 or -17 and 12. However on a machine in which overflows do not produce an exception and in which the results of overflows are reversible, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur. — end note]

\(^10\) A *full-expression* is an expression that is not a subexpression of another expression. [Note: in some contexts, such as unevaluated operands, a syntactic subexpression is considered a full-expression (Clause 5). — end note] 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. A call to a destructor generated at the end of the lifetime of an object other than a temporary object is an implicit full-expression. 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.

[Example:

```c
struct S {
    S(int i): I(i) { }
    int& v() { return I; }
private:
    int I;
};

S s1(1);    // full-expression is call of S::S(int)
S s2 = 2;    // full-expression is call of S::S(int)

void f() {
```

\(^7\) Overloaded operators are never assumed to be associative or commutative.
if (S(3).v())  // full-expression includes lvalue-to-rvalue and int to bool conversions, performed before
// temporary is deleted at end of full-expression
{
}
}
—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 (8.3.6) are considered to be created in the expression that calls the function, not the expression that defines the default argument. —end note]

Accessing an object designated by a volatile glvalue (3.10), 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 sub-expression) 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 to a volatile object is evaluated the side effect is considered complete, even though some external actions implied by the call (such as the I/O itself) or by the volatile access may not have completed yet.

Sequenced before is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread (1.10), which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, 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]

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.

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 scalar object is unsequenced relative to either another side effect on the same scalar object or a value computation using the value of the same scalar object, and they are not potentially concurrent (1.10), the behavior is undefined. [Note: The next section imposes similar, but more complex restrictions on potentially concurrent computations. —end note]

[Example:

```c
void f(int, int);
void g(int i, int* v) {
    i = v[i++]; // the behavior is undefined
    i = 7, i++, i++; // i becomes 9
    i = i++ + 1; // the behavior is undefined
    i = i + 1; // the value of i is incremented
    f(i = -1, i = -1); // the behavior is undefined
}
```

8) As specified in 12.2, 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.
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. [Note: Value computations and side effects associated with different argument expressions are unsequenced. — end note] Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function.\(^9\) 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 5.3.4. For another example, invocation of a conversion function (12.3.2) 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.

### 1.10 Multi-threaded executions and data races

A thread of execution (also known as a thread) is a single flow of control within a program, including the initial invocation of a specific top-level function, and recursively including every function invocation subsequently executed by the thread. [Note: When one thread creates another, the initial call to the top-level function of the new thread is executed by the new thread, not by the creating thread. — end note] Every thread in a program can potentially access every object and function in a program.\(^10\) 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 standard. The execution of the entire program consists of an execution of all of its threads. [Note: Usually the execution can be viewed as an interleaving of all its threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving, as described below. — end note] Under a freestanding implementation, it is implementation-defined whether a program can have more than one thread of execution.

A signal handler that is executed as a result of a call to the \texttt{raise} function belongs to the same thread of execution as the call to the \texttt{raise} function. Otherwise it is unspecified which thread of execution contains a signal handler invocation.

Implementations should ensure that all unblocked threads eventually make progress. [Note: Standard library functions may silently block on I/O or locks. Factors in the execution environment, including externally-imposed thread priorities, may prevent an implementation from making certain guarantees of forward progress. — end note]

Executions of atomic functions that are either defined to be lock-free (29.7) or indicated as lock-free (29.4) are lock-free executions.

\begin{enumerate}
\item If there is only one unblocked thread, 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]
\item 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
\end{enumerate}

\(^9\) In other words, function executions do not interleave with each other.
\(^10\) An object with automatic or thread storage duration (3.7) is associated with one specific thread, and can be accessed by a different thread only indirectly through a pointer or reference (3.9.2).
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 International Standard, this property is sometimes termed lock-free. — end note}

5 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 section 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]

6 Two expression evaluations conflict if one of them modifies a memory location (1.7) and the other one accesses or modifies the same memory location.

7 The library defines a number of atomic operations (Clause 29) and operations on mutexes (Clause 30) 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]

8 All modifications to a particular atomic object \( M \) occur in some particular total order, called the modification order of \( M \). If \( A \) and \( B \) are modifications of an atomic object \( M \) and \( A \) happens before (as defined below) \( B \), then \( A \) shall precede \( B \) in the modification order of \( M \), which is defined below. [Note: This states that the modification orders must respect the “happens before” relationship. — end note] [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]

9 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

(9.1) — is performed by the same thread that performed \( A \), or

(9.2) — is an atomic read-modify-write operation.

10 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 (29.3). [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]

11 An evaluation \( A \) carries a dependency to an evaluation \( B \) if

(11.1) — the value of \( A \) is used as an operand of \( B \), unless:
— B is an invocation of any specialization of `std::kill_dependency` (29.3), or

— A is the left operand of a built-in logical AND (`&&`, see 5.14) or logical OR (`||`, see 5.15) operator, or

— A is the left operand of a conditional (`?`, see 5.16) operator, or

— A is the left operand of the built-in comma (`,`) operator (5.19);

or

— A writes a scalar object or bit-field M, B reads the value written by A from M, and A is sequenced before B, or

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

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

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

— A performs a release operation on an atomic object M, and, in another thread, B performs a consume operation on M and reads a value written by any side effect in the release sequence headed by A, or

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

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

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

— A synchronizes with B, or

— A is dependency-ordered before B, or

— for some evaluation X

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

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

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

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

An evaluation A happens before an evaluation B if:

— A is sequenced before B, or

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

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:

15. (15.1) \( A \) happens before \( B \) and

15. (15.2) 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]

16. 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]

17. 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]

18. 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]

19. 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]

20. 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]

21. [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]

22. [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]

23. Two actions are potentially concurrent if

23. (23.1) they are performed by different threads, or

23. (23.2) they are unsequenced, and at least one is performed by a signal handler.

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. \[Note:\ It can be shown that programs that correctly use mutexes and \texttt{memory\_order\_seq\_cst} operations to prevent all data races and use no other synchronization operations behave as if the operations executed by their constituent threads were simply interleaved, with each value computation of an object being taken from the last side effect on that object in that interleaving. This is normally referred to as \texttt{“sequential consistency”}. However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result must perform an undefined operation. —end note\]

Two accesses to the same object of type \texttt{volatile 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 \texttt{volatile 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 \(B\).

\[Note:\ Compiler transformations that introduce assignments to a potentially shared memory location that would not be modified by the abstract machine are generally precluded by this standard, since such an assignment might overwrite another assignment by a different thread in cases in which an abstract machine execution would not have encountered a data race. This includes implementations of data member assignment that overwrite adjacent members in separate memory locations. Reordering of atomic loads in cases in which the atomics in question may alias is also generally precluded, since this may violate the coherence rules. —end note\]

\[Note:\ Transformations that introduce a speculative read of a potentially shared memory location may not preserve the semantics of the C++ program as defined in this standard, since they potentially introduce a data race. However, they are typically valid in the context of an optimizing compiler that targets a specific machine with well-defined semantics for data races. They would be invalid for a hypothetical machine that is not tolerant of races or provides hardware race detection. —end note\]

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

- (27.1) terminate,
- (27.2) make a call to a library I/O function,
- (27.3) access or modify a volatile object, or
- (27.4) perform a synchronization operation or an atomic operation.

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

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.

1.11 Acknowledgments


2 Portions of the library Clauses of this International Standard are based on work by P.J. Plauger, which was published as \textit{The Draft Standard C++ Library} (Prentice-Hall, ISBN 0-13-117003-1, copyright ©1995 P.J. Plauger).
3 POSIX® is a registered trademark of the Institute of Electrical and Electronic Engineers, Inc.
4 All rights in these originals are reserved.
2 Lexical conventions

2.1 Separate translation

The text of the program is kept in units called *source files* in this International Standard. A source file together with all the headers (17.6.1.2) and source files included (16.2) via the preprocessing directive `#include`, less any source lines skipped by any of the conditional inclusion (16.1) 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 (3.5) 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 (3.5). — end note]

2.2 Phases of translation

The precedence among the syntax rules of translation is specified by the following phases.\(^{1}\)

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 (2.3) 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 (i.e., using the `\uXXXX` notation), are handled equivalently except where this replacement is reverted 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 (2.4) and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment.\(^ {12}\) 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

---

\(^{1}\) Implementations must behave as if these separate phases occur, although in practice different phases might be folded together.

\(^{12}\) 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.
is produced by token concatenation (16.3.3), 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 (2.13.3, 2.13.5); if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.\(^{13}\)

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. (2.6). The resulting tokens are syntactically and semantically analyzed and translated as a translation unit. [Note: The process of analyzing and translating the tokens may occasionally result in one token being replaced by a sequence of other tokens (14.2). — end note] [Note: Source files, translation units and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation. — end note]

8. Translated translation units and instantiation units are combined as follows: [Note: Some or all of these may be supplied from a library. — end note] Each translated translation unit is examined to produce a list of required instantiations. [Note: This may include instantiations which have been explicitly requested (14.7.2). — end note] The definitions of the required templates are located. It is implementation-defined whether the source of the translation units containing these definitions is required to be available. [Note: An implementation could encode sufficient information into the translated translation unit so as to ensure the source is not required here. — end note] All the required instantiations are performed to produce instantiation units. [Note: These are similar to translated translation units, but contain no references to uninstantiated templates and no template definitions. — end note] The program is ill-formed if any instantiation fails.

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

2.3 Character sets  

[lex.charset]

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:\(^{14}\)

\[
\begin{align*}
abcde\ fghijklmnopqrstuvwxyz \\
ABCD\ EFGHIJKLmnopqrstuvwxyz \\
0123456789
\end{align*}
\]

\[
\begin{align*}
\_\ \{\ \}\[\ \\#\ \(\ \(\ <\ >\ %\ :\ ;\ .\ ?\ +\ -\ /\ \&\ \|\ \!=\ ,\ \"\ \'
\end{align*}
\]

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

\[
\text{hex-quad:}
\]

\[
\text{hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit}
\]

\(^{13}\) An implementation need not convert all non-corresponding source characters to the same execution character.

\(^{14}\) 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.
The character designated by the universal-character-name \UNNNNNNNN is that character whose character short name in ISO/IEC 10646 is NNNNNNN; the character designated by the universal-character-name \uNNNN is that character whose character short name in ISO/IEC 10646 is 0000NNNN. If the hexadecimal value for a universal-character-name corresponds to a surrogate code point (in the range 0xD800–0xDFFF, 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.  

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

2.4 Preprocessing tokens

Each preprocessing token that is converted to a token (2.6) shall have the lexical form of a keyword, an identifier, a literal, an operator, or a punctuator.

A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing token are: header names, identifiers, preprocessing numbers, character literals (including user-defined character literals), string literals (including user-defined string literals), preprocessing operators and punctuators, and single non-white-space characters that do not lexically match the other preprocessing token categories. If a ’ or a ” character matches the last category, the behavior is undefined. Preprocessing tokens can be separated by white space; this consists of comments (2.7), or white-space characters (space, horizontal tab, new-line, vertical tab, and form-feed), or both. As described in Clause 16, in certain circumstances during translation phase 4, white space (or the absence thereof) serves as more than preprocessing token separation. White space can appear within a preprocessing token only as part of a header name or between the quotation characters in a character literal or string literal.

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

(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

A sequence of characters resembling a universal-character-name in an r-char-sequence (2.13.5) does not form a universal-character-name.
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

\[
\text{encoding-prefix} R \text{ raw-string}
\]

(3.2) — Otherwise, if the next three characters are <: and the subsequent character is neither : nor >, the < is treated as a preprocessor 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 preprocessor token, even if that would cause further lexical analysis to fail.

[ 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 ]

2.5 Alternative tokens [ lex.digraph ]

Alternative token representations are provided for some operators and punctuators.\(^\text{16}\)

In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling.\(^\text{17}\) 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>&lt;:</td>
<td>[</td>
<td>and</td>
<td>&amp;&amp;</td>
<td>and_eq</td>
<td>&amp;=</td>
</tr>
<tr>
<td>%&gt;</td>
<td>)</td>
<td>bitor</td>
<td></td>
<td></td>
<td>or_eq</td>
</tr>
<tr>
<td>&lt;:</td>
<td>]</td>
<td>or</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;:</td>
<td>)</td>
<td>xor</td>
<td>-</td>
<td>not</td>
<td>!</td>
</tr>
<tr>
<td>%:</td>
<td>#</td>
<td>compl</td>
<td>-</td>
<td>not_eq</td>
<td>!^</td>
</tr>
<tr>
<td>%:%:</td>
<td>##</td>
<td>bitand</td>
<td>&amp;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^{16}\) 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, these alternative tokens that aren’t lexical keywords are colloquially known as “digraphs”.

\(^{17}\) Thus the “stringized” values (16.3.2) of { and <: will be different, maintaining the source spelling, but the tokens can otherwise be freely interchanged.

\[§ 2.5\]
2.6 Tokens

```
token:
  identifier
  keyword
  literal
  operator
  punctuator
```

1 There are five kinds of tokens: identifiers, keywords, literals,\(^{18}\) 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. \([\text{Note: Some white space is required to separate otherwise adjacent identifiers, keywords, numeric literals, and alternative tokens containing alphabetic characters. —end note]}\]

2.7 Comments

```
1 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. \([\text{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]}\]
```

2.8 Header names

```
header-name:
  &lt; h-char-sequence &gt;
  " 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 "
```

1 Header name preprocessing tokens shall only appear within a \#include preprocessing directive (16.2). 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 16.2.

2 The appearance of either of the characters \` or \\ or of either of the character sequences /* or // in a q-char-sequence or an h-char-sequence is conditionally-supported with implementation-defined semantics, as is the appearance of the character " in an h-char-sequence.\(^{19}\)

---

\(^{18}\) Literals include strings and character and numeric literals.

\(^{19}\) 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.
2.9 Preprocessing numbers

\[ p p \text{-} n u m b e r : \]
\[ \begin{\small}
   \text{digit} \\
   . \text{digit} \\
   p p \text{-} n u m b e r \ \text{digit} \\
   p p \text{-} n u m b e r \ \text{identifier-nondigit} \\
   p p \text{-} n u m b e r \ \text{'digit} \\
   p p \text{-} n u m b e r \ \text{'nondigit} \\
   p p \text{-} n u m b e r \ \text{e sign} \\
   p p \text{-} n u m b e r \ \text{E sign} \\
   p p \text{-} n u m b e r \ \text{.} \\
\end{\small} \]

1. Preprocessing number tokens lexically include all integer literal tokens (2.13.2) and all floating literal tokens (2.13.4).

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.

2.10 Identifiers

\[ \text{identifier:} \]
\[ \begin{\small}
   \text{identifier-nondigit} \\
   \text{identifier identifier-nondigit} \\
   \text{identifier digit} \\
\end{\small} \]

\[ \text{identifier-nondigit:} \]
\[ \begin{\small}
   \text{nondigit} \\
   \text{universal-character-name} \\
   \text{other implementation-defined characters} \\
\end{\small} \]

\[ \text{nondigit: one of} \]
\[ \begin{\small}
   \text{a b c d e f g h i j k l m} \\
   \text{n o p q r s t u v w x y z} \\
   \text{A B C D E F G H I J K L M} \\
   \text{N O P Q R S T U V W X Y Z} \\
\end{\small} \]

\[ \text{digit: one of} \]
\[ \begin{\small}
   0 1 2 3 4 5 6 7 8 9 \\
\end{\small} \]

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 E.1. The initial element shall not be a universal-character-name designating a character whose encoding falls into one of the ranges specified in E.2. Upper- and lower-case letters are different. All characters are significant.\(^{20}\)

2. The identifiers in Table 2 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 \text{identifier} grammar production. Unless otherwise specified, any ambiguity as to whether a given \text{identifier} has a special meaning is resolved to interpret the token as a regular \text{identifier}.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{override} & \textbf{final} \\
\hline
\end{tabular}
\caption{Identifiers with special meaning}
\end{table}

\(^{20}\) 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 \text{"\textbackslash}a\text{"} 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.

§ 2.10

22
3 In addition, some identifiers are reserved for use by C++ implementations and shall not be used otherwise; no diagnostic is required.

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

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

2.11 Keywords

The identifiers shown in Table 3 are reserved for use as keywords (that is, they are unconditionally treated as keywords in phase 7) except in an attribute-token (7.6.1) [Note: The export keyword is unused but is reserved for future use. — end note]:

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

2 Furthermore, the alternative representations shown in Table 4 for certain operators and punctuators (2.5) are reserved and shall not be used otherwise:

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

2.12 Operators and punctuators

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

preprocessing-op-or-punc: one of

```plaintext
{ } [ ] # ## ( )
<: :> <% %> %: %:%; ; : ...
new delete ? :: . .*
+ - * / % ^ & | 
= <= >= != && || ++ -- , ->
and and_eq bitand bitor compl not not_eq
or or_eq xor xor_eq
```
Each `preprocessing-op-or-punc` is converted to a single token in translation phase 7 (2.2).

2.13 Literals

2.13.1 Kinds of literals

There are several kinds of literals.\(^{21}\)

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

2.13.2 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:
  0x hexadecimal-digit
  0X hexadecimal-digit
  hexadecimal-literal \(^{opt}\) hexadecimal-digit

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-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\(^{opt}\)
```

\(^{21}\) The term “literal” generally designates, in this International Standard, those tokens that are called “constants” in ISO C.
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 literals 1048576, 1'048'576, 0X10000, 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 5 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>unsigned long int</td>
</tr>
<tr>
<td>l or L</td>
<td>long int</td>
<td>long int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>long 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>Both u or U and l or L</td>
<td>unsigned long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>ll or LL</td>
<td>long long int</td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>Both u or U and ll or LL</td>
<td>unsigned long int</td>
<td>unsigned long long int</td>
</tr>
</tbody>
</table>

If an integer literal cannot be represented by any type in its list and an extended integer type (3.9.1) can represent its value, it may have that extended integer type. If all of the types in the list for the literal are signed, the extended integer type shall be signed. If all of the types in the list for the 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.

22) The digits 8 and 9 are not octal digits.
2.13.3 Character literals

character-literal:
  encoding-prefixopt c-char-sequence '  
encoding-prefix: one of 
  u8 u U L  
c-char-sequence:  
  c-char  
  c-char-sequence c-char  
c-char:  
  any member of the 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 c d e f g h \  
  \a \b \c \d \e \f \g \h \i  
octal-escape-sequence:  
  \ octal-digit  
  \ octal-digit octal-digit  
  \ octal-digit octal-digit octal-digit  
hexadecimal-escape-sequence:  
  \x hexadecimal-digit  
  hexadecimal-escape-sequence hexadecimal-digit

1 A character literal is one or more characters enclosed in single quotes, as in 'x', optionally preceded by u8, u, U, or L, as in u8'w', u'x', U'y', or L'z', respectively.

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

3 A character literal that begins with u8, such as u8’w’, is a character literal of type 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.

4 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 literal containing a single c-char is equal to its ISO 10646 code point value, provided that the code point is representable with a single 16-bit code unit. (That is, provided it is a basic multi-lingual plane code point.) If the value is not representable within 16 bits, the program is ill-formed. A char16_t literal containing multiple c-chars is ill-formed.

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

6 A character literal that begins with the letter L, such as L’z’, is a wide-character literal. A wide-character
literal has type \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.

[\textit{Note:} The type \texttt{wchar_t} is able to represent all members of the execution wide-character set (see 3.9.1). — end note] The value of a wide-character literal containing multiple \texttt{c-chars} is implementation-defined.

7 Certain nongraphic 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++14 and ISO C.} and the backslash \, can be represented according to Table 6. The double quote ” and the question mark ?, can be represented as themselves or by the escape sequences \" and \? respectively, but the single quote ‘ and the backslash \ shall be represented by the escape sequences \’ and \\ respectively. Escape sequences in which the character following the backslash is not listed in Table 6 are conditionally-supported, with implementation-defined semantics. An escape sequence specifies a single character.

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

8 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 \texttt{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 literals with no prefix) or \texttt{wchar_t} (for literals prefixed by \texttt{L}). [\textit{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]

9 A universal-character-name is translated to the encoding, in the appropriate execution character set, of the character named. If there is no such encoding, the universal-character-name is translated to an implementation-defined encoding. [\textit{Note:} In translation phase 1, a universal-character-name is introduced whenever an actual extended character is encountered in the source text. Therefore, all extended characters are described in terms of universal-character-names. However, the actual compiler implementation may use its own native character set, so long as the same results are obtained. — end note]

\subsection*{2.13.4 Floating literals} [lex.fcon]

\begin{verbatim}
floating-literal: fractional-constant exponent-part_opt floating-suffix_opt
digit-sequence exponent-part floating-suffix_opt
\end{verbatim}

\footnote{They are intended for character sets where a character does not fit into a single byte.} They are intended for character sets where a character does not fit into a single byte.
fractional-constant:
  digit-sequence<opt> . digit-sequence
digit-sequence .

exponent-part:
  e sign<opt> digit-sequence
e E sign<opt> digit-sequence

sign: one of
  + -
digit-sequence:
  digit
digit-sequence signed opt digit

floating-suffix: one of
  f l F L

1 A floating literal consists of an integer part, a decimal point, a fraction part, an e or E, 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. Optional separating single quotes in a digit-sequence are ignored when determining its value. [Example: The 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 decimal point or the letter e (or E) and the exponent (not both) can be omitted. The integer part, the optional decimal point and the optional fraction part form the significant part of the floating literal. The exponent, if present, indicates the power of 10 by which the significant part is to be scaled. If the scaled value is in the range of representable values for its type, the result is the scaled value if representable, else the larger or smaller representable value nearest the scaled value, chosen in an implementation-defined manner. The type of a floating literal is double unless explicitly specified by a suffix. The suffixes f and F specify float, the suffixes l and L specify long double. If the scaled value is not in the range of representable values for its type, the program is ill-formed.

2.13.5 String literals [lex.string]

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.

1 A string-literal is a sequence of characters (as defined in 2.13.3) 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:

\[
\text{const char* p = R"(a\n b c)"; assert(std::strcmp(p, "a\n b c") == 0);}\]

— end note ]

5 [ Example: The raw string

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

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

\[
R"(??)"
\]
is equivalent to "\n??\n". The raw string

\[
R"#(??=\n)\n"
\]
is equivalent to "\n?)??=\n\n". — 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 (3.7).

9 For a UTF-8 string literal, each successive element of the object representation (3.9) 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.

§ 2.13.5 29
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.

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 (2.2), adjacent string-literal s are concatenated. If both string-literal s 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 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 7 has some examples of valid concatenations.

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

Characters in concatenated strings are kept distinct.

[Example:

"\xA" "\xB"

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

14 After any necessary concatenation, in translation phase 7 (2.2), \0 is appended to every string literal so that programs that scan a string can find its end.

15 Escape sequences and universal-character-names in non-raw string literals have the same meaning as in character literals (2.13.3), 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 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]
2.13.6 Boolean literals

boolean-literal:
  false
  true

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

2.13.7 Pointer literals

pointer-literal:
  nullptr

1 The pointer literal is the keyword nullptr. It is a prvalue of type std::nullptr_t. [Note: std::nullptr_t is a distinct type that is neither a pointer type nor a pointer to member type; rather, a prvalue of this type is a null pointer constant and can be converted to a null pointer value or null member pointer value. See 4.10 and 4.11. —end note]

2.13.8 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

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 (13.5.8). 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 (3.4.1). 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 (13.5.8) but not both. If S contains a raw literal operator, the literal L is treated as a call of the form

operator "" X(""n")

§ 2.13.8
Otherwise (\(S\) contains a literal operator template), \(L\) is treated as a call of the form

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

where \(n\) is the source character sequence \(c_1c_2\ldots c_k\). [Note: The sequence \(c_1c_2\ldots c_k\) 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

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

Otherwise, \(S\) shall contain a raw literal operator or a literal operator template (13.5.8) but not both. If \(S\) contains a raw literal operator, the literal \(L\) is treated as a call of the form

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

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

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

where \(f\) is the source character sequence \(c_1c_2\ldots c_k\). [Note: The sequence \(c_1c_2\ldots c_k\) can only contain characters from the basic source character set. — end note]

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

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

6 If \(L\) is a user-defined-character-literal, let \(ch\) be the literal without its ud-suffix. \(S\) shall contain a literal operator (13.5.8) 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:

```
long double operator "_w(long double);
std::string operator "_w(const char16_t*, std::size_t);
unsigned operator "_w(const char*);
```

```
int main() {
   1.2_w;       // calls operator "_w(1.2L)
   u"one"_w;   // calls operator "_w(u"one", 3)
   12_w;       // calls operator "_w(12")
   "two"_w;    // error: no applicable literal operator
}
```

— end example]

8 In translation phase 6 (2.2), 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 2.13.5. 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:

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

§ 2.13.8
— end example ]
3 Basic concepts [basic]

1 [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]

2 [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]

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

4 A name is a use of an identifier (2.10), operator-function-id (13.5), literal-operator-id (13.5.8), conversion-function-id (12.3.2), or template-id (14.2) that denotes an entity or label (6.6.4, 6.1).

5 Every name that denotes an entity is introduced by a declaration. Every name that denotes a label is introduced either by a goto statement (6.6.4) or a labeled-statement (6.1).

6 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.

7 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 (3.4).

8 Two names are the same if

8.1 — they are identifiers composed of the same character sequence, or

8.2 — they are operator-function-ids formed with the same operator, or

8.3 — they are conversion-function-ids formed with the same type, or

8.4 — they are template-ids that refer to the same class, function, or variable (14.4), or

8.5 — they are the names of literal operators (13.5.8) formed with the same literal suffix identifier.

9 A name used in more than one translation unit can potentially refer to the same entity in these translation units depending on the linkage (3.5) of the name specified in each translation unit.

3.1 Declarations and definitions [basic.def]

1 A declaration (Clause 7) 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:

1.1 — a static assertion (Clause 7),

1.2 — controlling template instantiation (14.7.2),

1.3 — use of attributes (Clause 7), and

1.4 — nothing (in the case of an empty-declaration).
A declaration is a definition unless it declares a function without specifying the function’s body (8.4), it contains the extern specifier (7.1.1) or a linkage-specification25 (7.5) and neither an initializer nor a function-body, it declares a static data member in a class definition (9.2, 9.4), it is a class name declaration (9.1), it is an opaque-enum-declaration (7.2), it is a template-parameter (14.1), it is a parameter-declaration (8.3.5) in a function declarator that is not the declarator of a function-definition, or it is a typedef declaration (7.1.3), an alias-declaration (7.1.3), a using-declaration (7.3.3), a static_assert-declaration (Clause 7), an attribute-declaration (Clause 7), an empty-declaration (Clause 7), an explicit instantiation declaration (14.7.2), or an explicit specialization (14.7.3) whose declaration is not a definition.

[Example: all but one of the following are definitions:

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

whereas these are just declarations:

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

—end example]

3 [Note: In some circumstances, C++ implementations implicitly define the default constructor (12.1), copy constructor (12.8), move constructor (12.8), copy assignment operator (12.8), move assignment operator (12.8), or destructor (12.4) member functions. —end note] [Example: given

```
#include <string>

struct C {
    std::string s; // std::string is the standard library class (Clause 21)
};

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

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

25) Appearing inside the braced-enclosed declaration-seq in a linkage-specification does not affect whether a declaration is a definition.
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 (7.1.6.3). — end note ]

5 A program is ill-formed if the definition of any object gives the object an incomplete type (3.9).

3.2 One definition rule [basic.def.odr]

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 (Clause 5) or a subexpression thereof. The set of potential results of an expression e is defined as follows:

(2.1) — If e is an id-expression (5.1.1), the set contains only e.

(2.2) — If e is a class member access expression (5.2.5), the set contains the potential results of the object expression.

(2.3) — If e is a pointer-to-member expression (5.5) whose second operand is a constant expression, the set contains the potential results of the object expression.

(2.4) — If e has the form (e1), the set contains the potential results of e1.

(2.5) — If e is a glvalue conditional expression (5.16), the set is the union of the sets of potential results of the second and third operands.

(2.6) — If e is a comma expression (5.19), the set contains the potential results of the right operand.

(2.7) — 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.

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 variable x whose name appears as a potentially-evaluated expression ex is odr-used by ex unless applying the lvalue-to-rvalue conversion (4.1) to x yields a constant expression (5.20) 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 (4.1) is applied to e, or e is a discarded-value expression (Clause
5). **This** is odr-used if it appears as a potentially-evaluated expression (including as the result of the implicit transformation in the body of a non-static member function (9.3.1)). A virtual member function is odr-used if it is not pure. A function whose name appears as a potentially-evaluated expression is odr-used if it is the unique lookup result or the selected member of a set of overloaded functions (3.4, 13.3, 13.4), unless it is a pure virtual function and either its name is not explicitly qualified or the expression forms a pointer to member (5.3.1). [Note: This covers calls to named functions (5.2.2), operator overloading (Clause 13), user-defined conversions (12.3.2), allocation function for placement new (5.3.4), as well as non-default initialization (8.5). A constructor selected to copy or move an object of class type is odr-used even if the call is actually elided by the implementation (12.8). — end note] An allocation or deallocation function for a class is odr-used by a *new-expression* appearing in a potentially-evaluated expression as specified in 5.3.4 and 12.5. A deallocation function for a class is odr-used by a delete expression appearing in a potentially-evaluated expression as specified in 5.3.5 and 12.5. 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 (12.4).26 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 12.8. A class is odr-used as specified in 8.5. A destructor for a class is odr-used if it is potentially invoked (12.4).

Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program; 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 12.1, 12.4 and 12.8). An inline function shall be defined in every translation unit in which it is odr-used.

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:

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

— end example] [Note: The rules for declarations and expressions describe in which contexts complete class types are required. A class type T must be complete if:

- an object of type T is defined (3.1), or
- a non-static class data member of type T is declared (9.2), or
- T is used as the object type or array element type in a *new-expression* (5.3.4), or
- an lvalue-to-rvalue conversion is applied to a gvalue referring to an object of type T (4.1), or
- an expression is converted (either implicitly or explicitly) to type T (Clause 4, 5.2.3, 5.2.7, 5.2.9, 5.4), 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 4), a dynamic_cast (5.2.7) or a static_cast (5.2.9), or
- a class member access operator is applied to an expression of type T (5.2.5), or
- the typeid operator (5.2.8) or the sizeof operator (5.3.3) is applied to an operand of type T, or

---

26) An implementation is not required to call allocation and deallocation functions from constructors or destructors; however, this is a permissible implementation technique.
— a function with a return type or argument type of type T is defined (3.1) or called (5.2.2), or
— a class with a base class of type T is defined (Clause 10), or
— an lvalue of type T is assigned to (5.18), or
— the type T is the subject of an alignof expression (5.3.6), or
— an exception-declaration has type T, reference to T, or pointer to T (15.3).

— end note

There can be more than one definition of a class type (Clause 9), enumeration type (7.2), inline function with external linkage (7.1.2), class template (Clause 14), non-static function template (14.5.6), static data member of a class template (14.5.1.3), member function of a class template (14.5.1.1), or template specialization for which some template parameters are not specified (14.7, 14.5.5) 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 3.4, shall refer to an entity defined within the definition of D, or shall refer to the same entity, after overload resolution (13.3) and after matching of partial template specialization (14.8.3), 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, and the object is initialized with a constant expression (5.20), and the object is not odr-used, and the object has the same value 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 three requirements described above (and, if the default argument has sub-expressions with default arguments, this requirement applies recursively). \[ Example: \]

```c
//translation unit 1:
struct X {
    X(int);
    X(int, int);
};
X::X(int = 0) {}  

class D: public X {};
D d2;  //X(int) called by D()
```

```c
//translation unit 2:
struct X {
    X(int);
```

\[ 27 \text{ } 8.3.6 \text{ } describes \text{ } how \text{ } default \text{ } argument \text{ } names \text{ } are \text{ } looked \text{ } up. \]
X(int, int);
};
X::X(int = 0, int = 0) { }

class D: public X { };//X(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 (14.6.3), and also to dependent names at the point of instantiation (14.6.2). If the definitions of D satisfy all these requirements, then the behavior is as if there were a single definition of D. If the definitions of D do not satisfy these requirements, then the behavior is undefined.

3.3 Scope

3.3.1 Declarative regions and scopes

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

2 [ Example: in

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

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

The names declared by a declaration are introduced into the scope in which the declaration occurs, except that the presence of a friend specifier (11.3), certain uses of the elaborated-type-specifier (7.1.6.3), and using-directives (7.3.4) alter this general behavior.

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

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

(4.2) exactly one declaration shall declare a class name or enumeration name that is not a typedef name and the other declarations shall all refer to the same variable or enumerator, or all refer to functions and function templates; in this case the class name or enumeration name is hidden (3.3.10). [ Note: A namespace name or a class template name must be unique in its declarative region (7.3.2, Clause 14). — end note ]

§ 3.3.1
[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 (7.1.6.3) and friend declarations (11.3) may introduce a (possibly not visible) name into an enclosing namespace; these restrictions apply to that region. Local extern declarations (3.5) 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]

5 [Note: The name lookup rules are summarized in 3.4. — end note]

### 3.3.2 Point of declaration

[Note: The name lookup rules are summarized in 3.4. — end note]

1 The point of declaration for a name is immediately after its complete declarator (Clause 8) and before its initializer (if any), except as noted below. [Example:

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

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

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

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

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

3 The point of declaration for a class or class template first declared by a class-specifier is immediately after the identifier or simple-template-id (if any) in its class-head (Clause 9). The point of declaration for an enumeration is immediately after the identifier (if any) in either its enum-specifier (7.2) or its first opaque-enum-declaration (7.2), whichever comes first. The point of declaration of an alias or alias template immediately follows the type-id to which the alias refers.

4 The point of declaration of a using-declaration that does not name a constructor is immediately after the using-declaration (7.3.3).

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

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

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

6 After the point of declaration of a class member, the member name can be looked up in the scope of its class. [Note: this is true even if the class is an incomplete class. For example,

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

— end note]

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

(7.1) — for a declaration of the form

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

the identifier is declared to be a class-name in the scope that contains the declaration, otherwise
for an elaborated-type-specifier of the form

class-key identifier

if the elaborated-type-specifier is used in the decl-specifier-seq or parameter-declaration-clause of a
function defined in namespace scope, the identifier is declared as a class-name in the namespace that
contains the declaration; otherwise, except as a friend declaration, the identifier is declared in the
smallest namespace or block scope that contains the declaration. [ Note: These rules also apply within
templates. — end note ] [ Note: Other forms of elaborated-type-specifier do not declare a new name,
and therefore must refer to an existing type-name. See 3.4.4 and 7.1.6.3. — end note ]

The point of declaration for an injected-class-name (Clause 9) is immediately following the opening brace of
the class definition.

The point of declaration for a function-local predefined variable (8.4) is immediately before the function-body
of a function definition.

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

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

— end example ]

[ Note: Friend declarations refer to functions or classes that are members of the nearest enclosing namespace,
but they do not introduce new names into that namespace (7.3.1.2). Function declarations at block scope
and variable declarations with the extern specifier at block scope refer to declarations that are members of
an enclosing namespace, but they do not introduce new names into that scope. — end note ]

[ Note: For point of instantiation of a template, see 14.6.4.1. — end note ]

3.3.3 Block scope [basic.scope.block]

A name declared in a block (6.3) is local to that block; it has block scope. Its potential scope begins at its
point of declaration (3.3.2) and ends at the end of its block. A variable declared at block scope is a local
variable.

The potential scope of a function parameter name (including one appearing in a lambda-declarator) or of
a function-local predefined variable in a function definition (8.4) begins at its point of declaration. If the
function has a function-try-block the potential scope of a parameter or of a function-local predefined variable
ends at the end of the last associated handler, otherwise it 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.

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

Names declared in the for-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 6.4.

3.3.4 Function prototype scope [basic.scope.proto]

In a function declaration, or in any function declarator except the declarator of a function definition (8.4),
names of parameters (if supplied) have function prototype scope, which terminates at the end of the nearest
enclosing function declarator.

3.3.5 Function scope

1 Labels (6.1) have function scope and may be used anywhere in the function in which they are declared. Only labels have function scope.

3.3.6 Namespace scope

1 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 (3.3.2) onwards; and for each using-directive (7.3.4) 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:

```cpp
namespace N {
    int i;
    int g(int a) { return a; }
    int j();
    void q();
}
namespace { int l=1; }
// the potential scope of l is from its point of declaration
// to the end of the translation unit
namespace N {
    int g(char a) { // overloads N::g(int)
        return l+a; // l is from unnamed namespace
    }

    int i; // error: duplicate definition
    int j(); // OK: duplicate function declaration
    int j() { // OK: definition of N::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 (5.1) applied to the name of its namespace or the name of a namespace which nominates the member’s namespace in a using-directive; see 3.4.3.2.

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 (3.3.2) 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.

3.3.7 Class scope

1 The following rules describe the scope of names declared in classes.

1) The potential scope of a name declared in a class consists not only of the declarative region following the name’s point of declaration, but also of all function bodies, default arguments, exception-specifications,
and brace-or-equal-initializers of non-static data members in that class (including such things in nested classes).

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

3) If reordering member declarations in a class yields an alternate valid program under (1) and (2), the program is ill-formed, no diagnostic is required.

4) 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.

5) 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 (8.3.6)).

Example:

```cpp
typedef int c;
enum { i = 1 };

class X {
    char v[i]; // error: i refers to ::i
    int f() { return sizeof(c); } // OK: X::c
    char c;
    enum { i = 2 }; }

typedef char* T;
struct Y {
    T a; // error: T refers to ::T
    typedef long T;
    T b;
};
typedef int I;
class D {
    typedef I I; // error, even though no reordering involved
};

— end example ]
```

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

(2.1) — in the scope of its class (as described above) or a class derived (Clause 10) from its class,

(2.2) — after the . operator applied to an expression of the type of its class (5.2.5) or a class derived from its class,

(2.3) — after the -> operator applied to a pointer to an object of its class (5.2.5) or a class derived from its class,

(2.4) — after the :: scope resolution operator (5.1) applied to the name of its class or a class derived from its class.
3.3.8 Enumeration scope

The name of a scoped enumerator (7.2) has *enumeration scope*. Its potential scope begins at its point of declaration and terminates at the end of the *enum-specifier*.

3.3.9 Template parameter scope

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

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

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

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

The potential scope of a template parameter name begins at its point of declaration (3.3.2) 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,*

```cpp
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 (3.3.10). *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*

*Note: Because the name of a template parameter cannot be redeclared within its potential scope (14.6.1), a template parameter’s scope is often its potential scope. However, it is still possible for a template parameter name to be hidden; see 14.6.1. — end note*
3.3.10 Name hiding

1 A name can be hidden by an explicit declaration of that same name in a nested declarative region or derived class (10.2).

2 A class name (9.1) or enumeration name (7.2) 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.

3 In a member function definition, the declaration of a name at block scope hides the declaration of a member of the class with the same name; see 3.3.7. The declaration of a member in a derived class (Clause 10) hides the declaration of a member of a base class of the same name; see 10.2.

4 During the lookup of a name qualified by a namespace name, declarations that would otherwise be made visible by a using-directive can be hidden by declarations with the same name in the namespace containing the using-directive; see (3.4.3.2).

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

3.4 Name lookup

1 The name lookup rules apply uniformly to all names (including typedef-names (7.1.3), namespace-names (7.3), and class-names (9.1)) wherever the grammar allows such names in the context discussed by a particular rule. Name lookup associates the use of a name with a declaration (3.1) of that name. Name lookup shall find an unambiguous declaration for the name (see 10.2). Name lookup may associate more than one declaration with a name if it finds the name to be a function name; the declarations are said to form a set of overloaded functions (13.1). Overload resolution (13.3) takes place after name lookup has succeeded. The access rules (Clause 11) 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 5).

2 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.

3 The injected-class-name of a class (Clause 9) is also considered to be a member of that class for the purposes of name hiding and lookup.

4 [Note: 3.5 discusses linkage issues. The notions of scope, point of declaration and name hiding are discussed in 3.3. —end note]

3.4.1 Unqualified name lookup

1 In all the cases listed in 3.4.1, the scopes are searched for a declaration in the order listed in each of the respective categories; name lookup ends as soon as a declaration is found for the name. If no declaration is found, the program is ill-formed.

2 The declarations from the namespace nominated by a using-directive become visible in a namespace enclosing the using-directive; see 7.3.4. For the purpose of the unqualified name lookup rules described in 3.4.1, 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 3.4.2. [Note: For purposes of determining (during parsing) whether an expression is a postfix-expression for a function call, the usual name lookup rules apply. The rules in 3.4.2 have no effect on the syntactic interpretation of an expression. For example,

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 (3.4.2) 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 A name used in the definition of a function following the function’s declarator-id 28 that is a member of namespace N (where, only for the purpose of exposition, N could represent the global scope) shall be declared before its use in the block in which it is used or in one of its enclosing blocks (6.3) 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:

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

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

    —end example]

7 A name used in the definition of a class X outside of a member function body, default argument, exception-specification, brace-or-equal-initializer of a non-static data member, or nested class definition 29 shall be declared in one of the following ways:

(7.1) — before its use in class X or be a member of a base class of X (10.2), or

(7.2) — if X is a nested class of class Y (9.7), 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), 30 or

28) 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.

29) This refers to unqualified names following the class name; such a name may be used in the base-clause or may be used in the class definition.

30) 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 (9.7).
(7.3) — if \( X \) is a local class (9.8) 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

(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.

[Example:

namespace M {
    class B { }
};

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

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

— end example] [Note: When looking for a prior declaration of a class or function introduced by a friend declaration, scopes outside of the innermost enclosing namespace scope are not considered; see 7.3.1.2. — end note] [Note: 3.3.7 further describes the restrictions on the use of names in a class definition. 9.7 further describes the restrictions on the use of names in nested class definitions. 9.8 further describes the restrictions on the use of names in local class definitions. — end note]

8 For the members of a class \( X \), a name used in a member function body, in a default argument, in an exception-specification, in the brace-or-equal-initializer of a non-static data member (9.2), or in the definition of a class member outside of the definition of \( X \), following the member’s declarator-id\(^{31}\), shall be declared in one of the following ways:

(8.1) — before its use in the block in which it is used or in an enclosing block (6.3), or

(8.2) — shall be a member of class \( X \) or be a member of a base class of \( X \) (10.2), or

(8.3) — if \( X \) is a nested class of class \( Y \) (9.7), 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)\(^{32}\), or

(8.4) — if \( X \) is a local class (9.8) 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

(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.

\(^{31}\) That is, an unqualified name that occurs, for instance, in a type in the parameter-declaration-clause or in the exception-specification.

\(^{32}\) 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: 9.3 and 9.4 further describe the restrictions on the use of names in member function definitions. 9.7 further describes the restrictions on the use of names in the scope of nested classes. 9.8 further describes the restrictions on the use of names in local class definitions. — end note ]

Name lookup for a name used in the definition of a `friend` function (11.3) defined inline in the class granting friendship shall proceed as described for lookup in member function definitions. If the `friend` function is not defined in the class granting friendship, name lookup in the `friend` function definition shall proceed as described for lookup in namespace member function definitions.

In a `friend` declaration naming a member function, a name used in the function declarator and not part of a `template-argument` in the `declarator-id` is first looked up in the scope of the member function’s class (10.2). 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]

During the lookup for a name used as a default argument (8.3.6) in a function `parameter-declaration-clause` or used in the `expression` of a `mem-initializer` for a constructor (12.6.2), 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: 8.3.6 further describes the restrictions on the use of names in default arguments. 12.6.2 further describes the restrictions on the use of names in a `ctor-initializer`. — end note ]
During the lookup of a name used in the constant-expression of an enumerator-definition, previously declared enumerators of the enumeration are visible and hide the names of entities declared in the block, class, or namespace scopes containing the enum-specifier.

A name used in the definition of a static data member of class X (9.4.2) (after the qualified-id of the static member) is looked up as if the name was used in a member function of X. [Note: 9.4.2 further describes the restrictions on the use of names in the definition of a static data member. — end note]

If a variable member of a namespace is defined outside of the scope of its namespace then any name that appears in the definition of the member (after the declarator-id) is looked up as if the definition of the member occurred in its namespace. [Example:

```cpp
namespace N {
  int i = 4;
  extern int j;
}

int i = 2;

int N::j = i; // N::j == 4

— end example]

A name used in the handler for a function-try-block (Clause 15) 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 14.6. — end note]

### 3.4.2 Argument-dependent name lookup

[basic.lookup.argdep]

When the postfix-expression in a function call (5.2.2) is an unqualified-id, other namespaces not considered during the usual unqualified lookup (3.4.1) may be searched, and in those namespaces, namespace-scope friend function or function template declarations (11.3) not otherwise visible may be found. These modifications to the search depend on the types of the arguments (and for template template arguments, the namespace of the template argument). [Example:

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

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

— end example]

For each argument type T in the function call, there is a set of zero or more associated namespaces and a set of zero or more associated classes to be considered. The sets of namespaces and classes is 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:

§ 3.4.2
If \( T \) is a fundamental type, its associated sets of namespaces and classes are both empty.

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 template 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]

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.

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

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

If an associated namespace is an inline namespace (7.3.1), 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 \emph{template-id}, its associated classes and namespaces also include those of its type \emph{template-arguments} and its template \emph{template-arguments}.

Let \( X \) be the lookup set produced by unqualified lookup (3.4.1) 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 \emph{using-declaration}, or
- a declaration that is neither a function or 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:

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

— end example

4 When considering an associated namespace, the lookup is the same as the lookup performed when the
associated namespace is used as a qualifier (3.4.3.2) except that:

(4.1) — Any using-directives in the associated namespace are ignored.
(4.2) — Any namespace-scope friend functions or friend function templates declared in associated classes are
visible within their respective namespaces even if they are not visible during an ordinary lookup (11.3).
(4.3) — All names except those of (possibly overloaded) functions and function templates are ignored.

3.4.3 Qualified name lookup [basic.lookup.qual]

1 The name of a class or namespace member or enumerator can be referred to after the :: scope resolution
operator (5.1) 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:

    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 (9.7) 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:

    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];  
    // not to: C::X C::arr[C::number];

— end example]
A name prefixed by the unary scope operator :: (5.1) 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 (3.4.3.2). The use of :: allows a global name to be referred to even if its identifier has been hidden (3.3.10).

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

If a pseudo-destructor-name (5.2.4) 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: 3.4.5 describes how name lookup proceeds after the . and -> operators. — end note]

### 3.4.3.1 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 (10.2), 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 10). [Note: A class member can be referred to using a qualified-id at any point in its potential scope (3.3.7). — end note] The exceptions to the name lookup rule above are the following:

   (1.1) — a destructor name is looked up as specified in 3.4.3;

   (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 3.4.5);

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

   (1.4) — the lookup for a name specified in a using-declaration (7.3.3) also finds class or enumeration names hidden within the same scope (3.3.10).

2 In a lookup in which function names are not ignored\(^\text{(33)}\) and the nested-name-specifier nominates a class C:

\(^{33}\) Lookups in which function names are ignored include names appearing in a nested-name-specifier, an elaborated-type-specifier, or a base-specifier.
— if the name specified after the nested-name-specifier, when looked up in C, is the injected-class-name of C (Clause 9), or

— in a using-declaration (7.3.3) 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:

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

— end example]
```

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

### 3.4.3.2 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 (7.3.1). If S′(X, m) is not empty, S(X, m) is S′(X, m); otherwise, S(X, m) is the union of S(Ni, m) for all namespaces Ni nominated by using-directives in X and its inline namespace set.

3 Given X::m (where X is a user-declared namespace), or given ::m (where X is the global namespace), if S(X, m) is the empty set, the program is ill-formed. Otherwise, if S(X, m) has exactly one member, or if the context of the reference is a using-declaration (7.3.3), 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:

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

namespace Z {  
    void h(double);  
}  

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

§ 3.4.3.2
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)
}

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

namespace A {
    int a;
}

namespace B {
  using namespace A;
}

namespace C {
  using namespace A;

\[ \text{§ 3.4.3.2} \]
namespace BC {
    using namespace B;
    using namespace C;
}

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

namespace D {
    using A::a;
}

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

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

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

namespace B {
    int b;
}

namespace A {
    using namespace B;
    int a;
}

namespace B {
    using namespace A;
}

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

—end example]
namespace A {
    struct x { };
    int x;
    int y;
}

namespace B {
    struct y { };
}

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

— end example ]

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

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

the unqualified-id shall name a member of the namespace designated by the nested-name-specifier or of an element of the inline namespace set (7.3.1) of that namespace. [ Example:

```c
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:

```c
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 ]

§ 3.4.3.2
3.4.4 Elaborated type specifiers

An elaborated-type-specifier (7.1.6.3) 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 (3.3.10).

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

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

the identifier is looked up according to 3.4.1 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-seq_opt identifier ;
```

the elaborated-type-specifier is a declaration that introduces the class-name as described in 3.3.2.

If the elaborated-type-specifier has a nested-name-specifier, qualified name lookup is performed, as described in 3.4.3, 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:

```c
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
    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 (7.1.6.3)
struct Base::Data; // error: cannot introduce a qualified type (7.1.6.3)
struct Base::Data* pBase; // error: Datum undefined
struct Base::Data* dpBase; // OK: refers to nested Data
```

— end example]
3.4.5 Class member access

1 In a class member access expression (5.2.5), 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 (14.2) 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 (5.2.5) 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 (5.2.4), the unqualified-id is looked up in the context 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 the type T of the object expression is of a class type C, the type-name is also looked up in the scope of class C. At least one of the lookups shall find a name that refers to (possibly cv-qualified) T. [Example:

```c
struct A { };

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

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

— end example]

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

```
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 3.4.3, 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 (14.2), 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:

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

§ 3.4.5 58
int main() {
    N::A a;
    a.operator A(); // calls N::A::operator N::A
}

— end example ]

3.4.6 Using-directives and namespace aliases

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

3.5 Program and linkage

A program consists of one or more translation units (Clause 2) linked together. A translation unit consists of a sequence of declarations.

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 (3.3.6) 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 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 9), or an unnamed class defined in a typedef declaration in which the class has the typedef name for linkage purposes (7.1.3); or

(4.4) a named enumeration (7.2), or an unnamed enumeration defined in a typedef declaration in which the enumeration has the typedef name for linkage purposes (7.1.3); or

(4.5) an enumerator belonging to an enumeration with linkage; or

(4.6) 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 (7.1.3), 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. [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
    }
}
```

There are three objects named i in this program. The object with internal linkage introduced by the
declaration in global scope (line #1), the object with automatic storage duration and no linkage introduced
by the declaration on line #2, and the object with static storage duration and external linkage introduced
by the declaration on line #3. — end example]

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

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

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

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

void q() { /* ... */ }       // some other, unrelated q
```

— end example]

Names not covered by these rules have no linkage. Moreover, except as noted, a name declared at block
scope (3.3.3) has no linkage. A type is said to have linkage if and only if:

(8.1) — it is a class or enumeration type that is named (or has a name for linkage purposes (7.1.3)) and the
name has linkage; or

(8.2) — it is an unnamed class or enumeration member of a class with linkage; or

§ 3.5
(8.3) — it is a specialization of a class template (Clause 14)\textsuperscript{34}; or
(8.4) — it is a fundamental type (3.9.1); or
(8.5) — it is a compound type (3.9.2) other than a class or enumeration, compounded exclusively from types that have linkage; or
(8.6) — it is a cv-qualified (3.9.3) 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

(8.7) — the entity has C language linkage (7.5), or
(8.8) — the entity is declared within an unnamed namespace (7.3.1), or
(8.9) — the entity is not odr-used (3.2) 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]

[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 };
    B<A> ba;             // declares B<A>::g(A) and B<A>::h(A)
    ba.g(a);             // OK
    ba.h(a);             // error: B<A>::h(A) not defined in the translation unit
    i(ba, a);            // OK
}
```

— end example]

Two names that are the same (Clause 3) and that are declared in different scopes shall denote the same variable, function, type, enumerator, template or namespace if

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

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

(9.3) — when both names denote functions, the parameter-type-lists of the functions (8.3.5) are identical; and

(9.4) — when both names denote function templates, the signatures (14.5.6.1) are the same.

\textsuperscript{34} A class template has the linkage of the innermost enclosing class or namespace in which it is declared.
After all adjustments of types (during which typedefs (7.1.3) 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 (8.3.4). 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 (7.5). — end note]

3.6 Start and termination

3.6.1 Main function

A program shall contain a global function called main, which is the designated start of the program. 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. 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 (8.3.5). 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) (17.5.2.1.4.2) 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 (3.5) 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 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) (18.5)) does not destroy any objects with automatic storage duration (12.4). 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 reaches the end of main without encountering a return statement, the effect is that of executing

return 0;

3.6.2 Initialization of non-local variables

There are two broad classes of named non-local variables: those with static storage duration (3.7.1) and those with thread storage duration (3.7.2). Non-local variables with static storage duration are initialized as a consequence of program initiation. Non-local variables with thread storage duration are initialized as a consequence of thread execution. Within each of these phases of initiation, initialization occurs as follows.

Variables with static storage duration (3.7.1) or thread storage duration (3.7.2) shall be zero-initialized (8.5) before any other initialization takes place. A constant initializer for an object o is an expression that is a
constant expression, except that it may also invoke \texttt{constexpr} constructors for \(o\) and its subobjects even if those objects are of non-literal class types \[\text{Note: such a class may have a non-trivial destructor — end note}\]. \textbf{Constant initialization} is performed:

- if each full-expression (including implicit conversions) that appears in the initializer of a reference with static or thread storage duration is a constant expression (5.20) and the reference is bound to a glvalue designating an object with static storage duration, to a temporary object (see 12.2) or subobject thereof, or to a function;
- if an object with static or thread storage duration is initialized by a constructor call, and if the initialization full-expression is a constant initializer for the object;
- if an object with static or thread storage duration is not initialized by a constructor call and if either the object is value-initialized or every full-expression that appears in its initializer is a constant expression.

Together, zero-initialization and constant initialization are called \textit{static initialization}; all other initialization is \textit{dynamic initialization}. Static initialization shall be performed before any dynamic initialization takes place. Dynamic initialization of a non-local variable with static storage duration is unordered if the variable is an implicitly or explicitly instantiated specialization, and otherwise is ordered \[\text{Note: an explicitly specialized static data member or variable template specialization has ordered initialization. — end note}\] Variables with ordered initialization defined within a single translation unit shall be initialized in the order of their definitions in the translation unit. If a program starts a thread (30.3), the subsequent initialization of a variable is unsequenced with respect to the initialization of a variable defined in a different translation unit. Otherwise, the initialization of a variable is indeterminately sequenced with respect to the initialization of a variable defined in a different translation unit. If a program starts a thread, the subsequent unordered initialization of a variable is unsequenced with respect to every other dynamic initialization. Otherwise, the unordered initialization of a variable is indeterminately sequenced with respect to every other dynamic initialization. \[\text{Note: This definition permits initialization of a sequence of ordered variables concurrently with another sequence. — end note}\] Variables with ordered initialization defined within a single translation unit shall be initialized in the order of their definitions in the translation unit. If a program starts a thread (30.3), the subsequent initialization of a variable is unsequenced with respect to the initialization of a variable defined in a different translation unit. Otherwise, the initialization of a variable is indeterminately sequenced with respect to the initialization of a variable defined in a different translation unit. If a program starts a thread, the subsequent unordered initialization of a variable is unsequenced with respect to every other dynamic initialization. \[\text{Note: This definition permits initialization of a sequence of ordered variables concurrently with another sequence. — end note}\] Variables with ordered initialization defined within a single translation unit shall be initialized in the order of their definitions in the translation unit. If a program starts a thread (30.3), the subsequent initialization of a variable is unsequenced with respect to the initialization of a variable defined in a different translation unit. Otherwise, the initialization of a variable is indeterminately sequenced with respect to the initialization of a variable defined in a different translation unit. If a program starts a thread, the subsequent unordered initialization of a variable is unsequenced with respect to every other dynamic initialization. Otherwise, the unordered initialization of a variable is indeterminately sequenced with respect to every other dynamic initialization. \[\text{Note: This definition permits initialization of a sequence of ordered variables concurrently with another sequence. — end note}\]

An implementation is permitted to perform the initialization of a non-local variable with static storage duration as a static initialization even if such initialization is not required to be done statically, provided that:

- the dynamic version of the initialization does not change the value of any other object of namespace scope prior to its initialization, and
- the static version of the initialization produces the same value in the initialized variable as would be produced by the dynamic initialization if all variables not required to be initialized statically were initialized dynamically.

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

\begin{verbatim}
inline double fd() { return 1.0; }
extern double d1;
double d2 = d1;  // unspecified:
    // may be statically initialized to 0.0 or
    // dynamically initialized to 0.0 if d1 is
    // dynamically initialized, or 1.0 otherwise
double d1 = fd();  // may be initialized statically or dynamically to 1.0
\end{verbatim}
It is implementation-defined whether the dynamic initialization of a non-local variable with static storage duration is done before the first statement of `main`. If the initialization is deferred to some point in time after the first statement of `main`, it shall occur before the first odr-use (3.2) of any function or variable defined in the same translation unit as the variable to be initialized.

```c
#include "a.h"
#include "b.h"
B b;
A::A(){
  b.Use();
}
```

```c
#include "a.h"
A a;
```

```c
#include "a.h"
#include "b.h"
extern A a;
extern B b;
```

```c
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`.

```c
A non-local variable with static storage duration having initialization with side-effects must be initialized even if it is not odr-used (3.2, 3.7.1).
```

It is implementation-defined whether the dynamic initialization of a non-local variable with static or thread storage duration is done before the first statement of the initial function of the thread. If the initialization is deferred to some point in time after the first statement of the initial function of the thread, it shall occur before the first odr-use (3.2) of any variable with thread storage duration defined in the same translation unit as the variable to be initialized.

If the initialization of a non-local variable with static or thread storage duration exits via an exception, `std::terminate` is called (15.5.1).

3.6.3 Termination

Destructors (12.4) for initialized objects (that is, objects whose lifetime (3.8) has begun) with static storage duration are called as a result of returning from `main` and as a result of calling `std::exit` (18.5). 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 are sequenced before the initiation of the destructors of any object with static storage duration. If the completion of the constructor or dynamic initialization of an object with thread storage duration is sequenced before that of

---

35) A non-local variable with static storage duration having initialization with side-effects must be initialized even if it is not odr-used (3.2, 3.7.1).
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 static 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. [Note: This definition permits concurrent destruction.
— end note] 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,
\texttt{std::terminate} is called (15.5.1).

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

If the completion of the initialization of an object with static storage duration is sequenced before a call
to \texttt{std::atexit} (see \texttt{<cstdlib>}, 18.5), the call to the function passed to \texttt{std::atexit} is sequenced before
the call to the destructor for the object. If a call to \texttt{std::atexit} is sequenced 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 \texttt{std::atexit}. If a call to \texttt{std::atexit} is sequenced before another
call to \texttt{std::atexit}, the call to the function passed to the second \texttt{std::atexit} call is sequenced before the
call to the function passed to the first \texttt{std::atexit} call.

If there is a use of a standard library object or function not permitted within signal handlers (18.10) that
does not happen before (1.10) completion of destruction of objects with static storage duration and execution
of \texttt{std::atexit} registered functions (18.5), the program has undefined behavior. [Note: If there is a use
of an object with static storage duration that does not happen before the object’s destruction, the program
has undefined behavior. Terminating every thread before a call to \texttt{std::exit} or the exit from \texttt{main} is
sufficient, but not necessary, to satisfy these requirements. These requirements permit thread managers as
static-storage-duration objects. — end note]

Calling the function \texttt{std::abort()} declared in \texttt{<cstdlib>} terminates the program without executing any
directors and without calling the functions passed to \texttt{std::atexit()} or \texttt{std::at_quick_exit()}.

### 3.7 Storage duration

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

- (1.1) static storage duration
- (1.2) thread storage duration
- (1.3) automatic storage duration
- (1.4) dynamic storage duration

Static, thread, and automatic storage durations are associated with objects introduced by declarations (3.1)
and implicitly created by the implementation (12.2). The dynamic storage duration is associated with objects
created with \texttt{operator new} (5.3.4).

The storage duration categories apply to references as well. The lifetime of a reference is its storage duration.
3.7.1 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 (3.6.2, 3.6.3).

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 12.8.

The keyword static can be used to declare a local variable with static storage duration. [Note: 6.7 describes the initialization of local static variables; 3.6.3 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.

3.7.2 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 (3.2) and, if constructed, shall be destroyed on thread exit.

3.7.3 Automatic storage duration

Block-scope variables explicitly declared register or not explicitly declared static 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 6.7. — end note]

If a variable with automatic storage duration has initialization or a destructor with side effects, it shall not be destroyed before the end of its block, nor shall it be eliminated 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 12.8.

3.7.4 Dynamic storage duration

Objects can be created dynamically during program execution (1.9), using new-expressions (5.3.4), and destroyed using delete-expressions (5.3.5). 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[].

The library provides default definitions for the global allocation and deallocation functions. Some global allocation and deallocation functions are replaceable (18.6.1). 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 (17.6.4.6). The following allocation and deallocation functions (18.6) are implicitly declared in global scope in each translation unit of a program.

```c
void* operator new(std::size_t);
void* operator new[](std::size_t);
void operator delete(void*) noexcept;
void operator delete[](void*) noexcept;
void operator delete(void*, std::size_t) noexcept;
void operator delete[](void*, std::size_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, 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 is ill-formed unless the name has been declared by including the appropriate header. — end note] Allocation and/or deallocation functions can also be declared and defined for any class (12.5).

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 3.7.4.1 and 3.7.4.2.

3.7.4.1 Allocation functions [basic.stc.dynamic.allocation]

1 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 (18.2). The first parameter shall not have an associated default argument (8.3.6). 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.

2 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 of any complete object type with a fundamental alignment requirement (3.11) 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 (4.10) 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 18.6.1.1 and 18.6.1.2, p0 shall point to 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.36

3 An allocation function that fails to allocate storage can invoke the currently installed new-handler function (18.6.2.3), 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 (18.6.2.4). — end note] If an allocation function that has a non-throwing exception specification (15.4) 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 (15.1) of a type that would match a handler (15.3) of type std::bad_alloc (18.6.2.1).

4 A global allocation function is only called as the result of a new expression (5.3.4), or called directly using the function call syntax (5.2.2), 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 (3.7.1), for objects or references with thread storage duration (3.7.2), for objects of type std::type_info (5.2.8), or for an exception object (15.1). — end note]

3.7.4.2 Deallocation functions [basic.stc.dynamic.deallocation]

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

36) The intent is to have operator new() implementable by calling std::malloc() or std::calloc(), so the rules are substantially the same. C++ differs from C in requiring a zero request to return a non-null pointer.
Each deallocation function shall return `void` and its first parameter shall be `void*`. A deallocation function can have more than one parameter. The global `operator delete` with exactly one parameter is a usual (non-placement) deallocation function. The global `operator delete` with exactly two parameters, the second of which has type `std::size_t`, is a usual deallocation function. Similarly, the global `operator delete[]` with exactly one parameter is a usual deallocation function. The global `operator delete[]` with exactly two parameters, the second of which has type `std::size_t`, is a usual deallocation function. If a class `T` has a member deallocation function named `operator delete` with exactly one parameter, then that function is a usual deallocation function. If class `T` does not declare such an `operator delete` but does declare a member deallocation function named `operator delete` with exactly two parameters, the second of which has type `std::size_t`, then this function is a usual deallocation function. Similarly, if a class `T` has a member deallocation function named `operator delete[]` with exactly one parameter, then that function is a usual (non-placement) deallocation function. If class `T` does not declare such an `operator delete[]` but does declare a member deallocation function named `operator delete[]` with exactly two parameters, the second of which has type `std::size_t`, then this function is a usual deallocation function. A deallocation function can 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.

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. Otherwise, the behavior is undefined if the value supplied to `operator delete(void*)` in the standard library is not one of the values returned by a previous invocation of either `operator new(std::size_t)` or `operator new(std::size_t, const std::nothrow_t&)` in the standard library, and the behavior is undefined if the value supplied to `operator delete[](void*)` in the standard library is not one of the values returned by a previous invocation of either `operator new[](std::size_t)` or `operator new[](std::size_t, const std::nothrow_t&)` in the standard library.

If the argument given to a deallocation function in the standard library is a pointer that is not the null pointer value (4.10), the deallocation function shall deallocate the storage referenced by the pointer, rendering invalid all pointers referring to any part of the deallocated storage. 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.\(^\text{38}\)

3.7.4.3 Safely-derived pointers

A `traceable pointer object` is

1. an object of an object pointer type (3.9.2), 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 (3.9.1), 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:

---

\(^{37}\) This deallocation function precludes use of an allocation function `void operator new(std::size_t, std::size_t)` as a placement allocation function (C.3.2).

\(^{38}\) Some implementations might define that copying an invalid pointer value causes a system-generated runtime fault.
(2.1)  — the value returned by a call to the C++ standard library implementation of `operator new(std::size_t);`\(^{39}\)

(2.2)  — the result of taking the address of an object (or one of its subobjects) designated by an lvalue resulting from indirection through a safely-derived pointer value;

(2.3)  — the result of well-defined pointer arithmetic (5.7) using a safely-derived pointer value;

(2.4)  — the result of a well-defined pointer conversion (4.10, 5.4) of a safely-derived pointer value;

(2.5)  — the result of a `reinterpret_cast` of a safely-derived pointer value;

(2.6)  — the result of a `reinterpret_cast` of an integer representation of a safely-derived pointer value;

(2.7)  — the value of an object whose value was copied from a traceable pointer object, where at the time of the copy the source object contained a copy of a safely-derived pointer value.

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

(3.1)  — the result of a `reinterpret_cast` of a safely-derived pointer value;

(3.2)  — the result of a valid conversion of an integer representation of a safely-derived pointer value;

(3.3)  — the value of an object whose value was copied from a traceable pointer object, where at the time of the copy the source object contained an integer representation of a safely-derived pointer value;

(3.4)  — the result of an additive or bitwise operation, one of whose operands is an integer representation of a safely-derived pointer value `P`, if that result converted by `reinterpret_cast<void*>(P)` would compare equal to a safely-derived pointer computable from `reinterpret_cast<void*>(P)`.

4 An implementation may have relaxed pointer safety, in which case the validity of a pointer value does not depend on whether it is a safely-derived pointer value. Alternatively, an implementation may have strict pointer safety, in which case a pointer value referring to an object with dynamic storage duration that is not a safely-derived pointer value is an invalid pointer value unless the referenced complete object has previously been declared reachable (20.7.4). [Note: the effect of using an invalid pointer value (including passing it to a deallocation function) is undefined, see 3.7.4.2. 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.

### 3.7.5 Duration of subobjects

The storage duration of member subobjects, base class subobjects and array elements is that of their complete object (1.8).

### 3.8 Object lifetime

The lifetime of an object is a runtime property of the object. An object is said to have non-vacuous initialization if it is of a class or aggregate type and it or one of its members 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

---

39) This section 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.
The lifetime of an object of type $T$ ends when:

- if the object has non-vacuous initialization, its initialization is complete.
- if $T$ is a class type with a non-trivial destructor (12.4), the destructor call starts, or
- the storage which the object occupies is reused or released.

[Note: The lifetime of an array object starts as soon as storage with proper size and alignment is obtained, and its lifetime ends when the storage which the array occupies is reused or released. 12.6.2 describes the lifetime of base and member subobjects. — end note]

The properties ascribed to objects throughout this International Standard apply for a given object 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 12.6.2 and in 12.7. 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. 12.6.2 and 12.7 describe the behavior of objects during the construction and destruction phases. — end note]

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 (5.3.5) 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.

Before the lifetime of an object has started but after the storage which the object will occupy has been allocated (40) or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that refers to 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 12.7. Otherwise, such a pointer refers to allocated storage (3.7.4.2), and using the pointer as if the pointer were of type void*, is well-defined. Indirection through such a pointer is permitted but the resulting 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 (4.10) to a pointer to a virtual base class, or
- the pointer is used as the operand of a static_cast (5.2.9), except when the conversion is to pointer to cv void, or to pointer to cv void and subsequently to pointer to either cv char or cv unsigned char, or
- the pointer is used as the operand of a dynamic_cast (5.2.7). [Example:

```c
#include <cstdlib>

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

40) For example, before the construction of a global object of non-POD class type (12.7).
struct D1 : B { void f(); }
struct D2 : B { void f(); }

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

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

—end example—

6 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 12.7. Otherwise, such a glvalue refers to allocated
storage (3.7.4.2), and using the properties of the glvalue that do not depend on its value is well-defined. The
program has undefined behavior if:

(6.1) — an lvalue-to-rvalue conversion (4.1) is applied to such a glvalue,
(6.2) — the glvalue is used to access a non-static data member or call a non-static member function of the
object, or
(6.3) — the glvalue is bound to a reference to a virtual base class (8.5.3), or
(6.4) — the glvalue is used as the operand of a dynamic_cast (5.2.7) or as the operand of typeid.

7 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:

(7.1) — the storage for the new object exactly overlays the storage location which the original object occupied,
and
(7.2) — the new object is of the same type as the original object (ignoring the top-level cv-qualifiers), and
(7.3) — 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
(7.4) — the original object was a most derived object (1.8) of type T and the new object is a most derived
object of type T (that is, they are not base class subobjects). [Example:

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

§ 3.8
constant C& C::operator=(const C& other) { 
   if (this != &other) {
      this->~C(); // lifetime of *this ends
      new (this) C(other); // new object of type C created
      f(); // well-defined
   }
   return *this;
}

C c1;
C c2;
c1 = c2; // well-defined
   c1.f(); // well-defined; c1 refers to a new object of type C

— end example ]

8 If a program ends the lifetime of an object of type T with static (3.7.1), thread (3.7.2), or automatic (3.7.3) storage duration and if T has a non-trivial destructor, the program must ensure that an object of the original type occupies that same storage location when the implicit destructor call takes place; otherwise the behavior of the program is undefined. This is true even if the block is exited with an exception. [ Example:

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

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

— end example ]

9 Creating a new object at the storage location that a const object with static, thread, or automatic storage duration occupies or, at the storage location that such a const object used to occupy before its lifetime ended results in undefined behavior. [ Example:

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

const B b;

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

— end example ]

41) 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.

§ 3.8
In this section, “before” and “after” refer to the “happens before” relation (1.10). [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]

3.9 Types

[Note: 3.9 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 (1.8), references (8.3.2), or functions (8.3.5). —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 (1.7) making up the object can be copied into an array of \( char \) or \( unsigned \ char \). If the content of the array of \( char \) or \( unsigned \ char \) is copied back into the object, the object shall subsequently hold its original value. [Example:

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

—end example]

For any trivially copyable type \( T \), if two pointers to \( T \) point to distinct \( T \) objects \( \text{obj1} \) and \( \text{obj2} \), where neither \( \text{obj1} \) nor \( \text{obj2} \) is a base-class subobject, if the underlying bytes (1.7) making up \( \text{obj1} \) are copied into \( \text{obj2} \), \( \text{obj2} \) shall subsequently hold the same value as \( \text{obj1} \). [Example:

```cpp
T* t1p;  
T* t2p;  
// provided that t2p points to an initialized object …  
std::memcpy(t1p, t2p, sizeof(T));  
// at this point, every subobject of trivially copyable type in *t1p contains  
// the same value as the corresponding subobject in *t2p
```

—end example]

The object representation of an object of type \( T \) is the sequence of \( N \) \( unsigned \ char \) objects taken up by the object of type \( T \), where \( N \) equals \( sizeof(T) \). The value representation of an object is the set of bits that hold the value of type \( T \). 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.

A class that has been declared but not defined, an enumeration type in certain contexts (7.2), or an array of unknown size or of incomplete element type, is an incompletely-defined object type. Incompletely-defined object types and the void types are incomplete types (3.9.1). Objects shall not be defined to have an incomplete type.

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

42) By using, for example, the library functions (17.6.1.2) `std::memcpy` or `std::memmove`.
43) By using, for example, the library functions (17.6.1.2) `std::memcpy` or `std::memmove`.
44) The intent is that the memory model of C++ is compatible with that of ISO/IEC 9899 Programming Language C.
45) The size and layout of an instance of an incompletely-defined object type is unknown.
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 \(N \times T\)"") are different types. The type of a pointer to array of unknown size, or of a type defined by a `typedef` declaration to be an array of unknown size, cannot be completed. [Example:

```c
class X; // X is an incomplete type
extern X* xp; // xp is a pointer to an incomplete type
extern int arr[]; // the type of arr is incomplete
typedef int UNKA[]; // UNKA is an incomplete type
UNKA* arrp; // arrp is a pointer to an incomplete type
UNKA** arrpp;

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

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

X x;
void bar() {
    xp = &x; // OK: type is "pointer to X"
    arrp = &arr; // ill-formed: different types
    xp++; // OK: X is complete
    arrp++; // ill-formed: UNKA can't be completed
}
```

— end example]

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

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

9 Arithmetic types (3.9.1), enumeration types, pointer types, pointer to member types (3.9.2), `std::nullptr_t`, and cv-qualified versions of these types (3.9.3) are collectively called scalar types. Scalar types, POD classes (Clause 9), arrays of such types and cv-qualified versions of these types (3.9.3) are collectively called POD types. Cv-qualified scalar types, trivially copyable class types (Clause 9), arrays of such types, and non-volatile const-qualified versions of these types (3.9.3) are collectively called trivially copyable types. Scalar types, trivial classes (Clause 9), arrays of such types and cv-qualified versions of these types (3.9.3) are collectively called trivial types. Scalar types, standard-layout class types (Clause 9), arrays of such types and cv-qualified versions of these types (3.9.3) are collectively called standard-layout types.

10 A type is a literal type if it is:

10.1 — void; or
10.2 — a scalar type; or
10.3 — a reference type; or
10.4 — an array of literal type; or
10.5 — a class type (Clause 9) that has all of the following properties:
— it has a trivial destructor,

— it is an aggregate type (8.5.1) or has at least one constexpr constructor or constructor template that is not a copy or move constructor, and

— all of its non-static data members and base classes are of non-volatile literal types.

Two types cev $T_1$ and cev $T_2$ are layout-compatible types if $T_1$ and $T_2$ are the same type, layout-compatible enumerations (7.2), or layout-compatible standard-layout class types (9.2).

3.9.1 Fundamental types

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 (3.11); that is, they have the same object representation. For narrow character types, all bits of the object representation participate in the value representation. 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 (4.7) 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; the other signed integer types are provided to meet special needs.

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 (3.11) as the corresponding signed integer type; 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, 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, section 5.2.4.2.1.

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. 

---

46) that is, large enough to contain any value in the range of INT_MIN and INT_MAX, as defined in the header <climits>.
47) See 7.1.6.2 regarding the correspondence between types and the sequences of type-specifiers that designate them.
48) 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.
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 (22.3.1). Type wchar_t shall have the same size, signedness, and alignment requirements (3.11) 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 <cstdint>, called the underlying types.

Values of type bool are either true or false. [Note: There are no signed, unsigned, short, or long bool types or values. —end note] Values of type bool participate in integral promotions (4.5).

Types bool, char, char16_t, char32_t, wchar_t, and the signed and unsigned integer types are collectively called integral types. A synonym for integral type is integer type. The representations of integral types shall define values by use of a pure binary numeration system. [Example: this International Standard permits 2’s complement, 1’s complement and signed magnitude representations for integral types. —end example]

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. Integral and floating types are collectively called arithmetic types. Specializations of the standard template std::numeric_limits (18.3) shall specify the maximum and minimum values of each arithmetic type for an implementation.

The void type has an empty set of values. The void type is an incomplete type that cannot be completed. 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 (5.4). An expression of type void shall be used only as an expression statement (6.2), as an operand of a comma expression (5.19), as the operand of a question mark operator (5.16), as the operand of typeid, noexcept, or decltype, as the expression in a return statement (6.6.3) for a function with the return type void, or as the operand of an explicit conversion to type cv void.

A value of type std::nullptr_t is a null pointer constant (4.10). Such values participate in the pointer and the pointer to member conversions (4.10, 4.11). sizeof(std::nullptr_t) shall be equal to sizeof(void*).

[Note: Even if the implementation defines two or more basic types to have the same value representation, they are nevertheless different types. —end note]

### 3.9.2 Compound types

Compound types can be constructed in the following ways:

1. **arrays** of objects of a given type, 8.3.4;
2. **functions**, which have parameters of given types and return void or references or objects of a given type, 8.3.5;
3. **pointers** to void or objects or functions (including static members of classes) of a given type, 8.3.1;
4. **references** to objects or functions of a given type, 8.3.2. There are two types of references:

   1. **lvalue reference**
   2. **rvalue reference**

---

49) Using a bool value in ways described by this International Standard as “undefined,” such as by examining the value of an uninitialzed automatic object, might cause it to behave as if it is neither true nor false.

50) Therefore, enumerations (7.2) are not integral; however, enumerations can be promoted to integral types as specified in 4.5.

51) 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.*)
— classes containing a sequence of objects of various types (Clause 9), a set of types, enumerations and functions for manipulating these objects (9.3), and a set of restrictions on the access to these entities (Clause 11);

— unions, which are classes capable of containing objects of different types at different times, 9.5;

— enumerations, which comprise a set of named constant values. Each distinct enumeration constitutes a different enumerated type, 7.2;

— pointers to non-static class members, which identify members of a given type within objects of a given class, 8.3.3.

These methods of constructing types can be applied recursively; restrictions are mentioned in 8.3.1, 8.3.4, 8.3.5, and 8.3.2. Constructing a type such that the number of bytes in its object representation exceeds the maximum value representable in the type `std::size_t` (18.2) is ill-formed.

The type of a pointer to `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 function 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 (3.11). A valid value of an object pointer type represents either the address of a byte in memory (1.7) or a null pointer (4.10). If an object of type `T` is located at an address `A`, a pointer of type `cv T*` whose value is the address `A` is said to point to that object, regardless of how the value was obtained. [Note: For instance, the address one past the end of an array (5.7) would be considered to point to an unrelated object of the array’s element type that might be located at that address. There are further restrictions on pointers to objects with dynamic storage duration; see 3.7.4.3. — end note] The value representation of pointer types is implementation-defined. Pointers to layout-compatible types shall have the same value representation and alignment requirements (3.11). [Note: Pointers to over-aligned types (3.11) have no special representation, but their range of valid values is restricted by the extended alignment requirement. This International Standard specifies only two ways of obtaining such a pointer: taking the address of a valid object with an over-aligned type, and using one of the runtime pointer alignment functions. An implementation may provide other means of obtaining a valid pointer value for an over-aligned type. — end note]

A pointer to `cv`-qualified (3.9.3) or `cv`-unqualified `void` can be used to point to objects of unknown type. Such a pointer shall be able to hold any object pointer. An object of type `cv void*` shall have the same representation and alignment requirements as `cv char*`.

### 3.9.3 CV-qualifiers

A type mentioned in 3.9.1 and 3.9.2 is a cv-unqualified type. Each type which is a cv-unqualified complete or incomplete object type or is `void` (3.9) has three corresponding cv-qualified versions of its type: a const-qualified version, a volatile-qualified version, and a const-volatile-qualified version. The term object type (1.8) includes the cv-qualifiers specified in the decl-specifier-seq (7.1), declarator (Clause 8), type-id (8.1), or newtype-id (5.3.4) when the object is created.

— A const object is an object of type `const T` or a non-mutable subobject of such an object.

— A volatile object is an object of type `volatile T`, a subobject of such an object, or a mutable subobject of a const volatile object.

— A const volatile object is an object of type `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.

§ 3.9.3

52) Static class members are objects or functions, and pointers to them are ordinary pointers to objects or functions.
The cv-qualified or cv-unqualified versions of a type are distinct types; however, they shall have the same representation and alignment requirements (3.11).\(^5\)

2 A compound type (3.9.2) is not cv-qualified by the cv-qualifiers (if any) of the types from which it is compounded. Any cv-qualifiers applied to an array type affect the array element type, not the array type (8.3.4).

3 See 8.3.5 and 9.3.2 regarding function types that have cv-qualifiers.

4 There is a partial ordering on cv-qualifiers, so that a type can be said to be more cv-qualified than another. Table 8 shows the relations that constitute this ordering.

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

5 In this International Standard, the notation \(\text{cv} \) (or \(\text{cv}1, \text{cv}2\), etc.), used in the description of types, represents an arbitrary set of cv-qualifiers, i.e., one of \{\text{const}\}, \{\text{volatile}\}, \{\text{const}, \text{volatile}\}, or the empty set. For a type \(\text{cv} \ T\), the top-level cv-qualifiers of that type are those denoted by \(\text{cv}\). \[\text{Example:}\] The type corresponding to the type-id int\& has no top-level cv-qualifiers. The type corresponding to the type-id volatile int \* const has the top-level cv-qualifier const. For a class type \(\text{C}\), the type corresponding to the type-id void (\(\text{C}::\*)\ volatile)(int) const has the top-level cv-qualifier volatile. \[-\text{end example}\]\n
6 Cv-qualifiers applied to an array type attach to the underlying element type, so the notation “\(\text{cv} \ T\),” where \(T\) is an array type, refers to an array whose elements are so-qualified. An array type whose elements are cv-qualified is also considered to have the same cv-qualifications as its elements. \[\text{Example:}\]

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

The type of both \(\text{arr1}\) and \(\text{arr2}\) is “array of 5 const char,” and the array type is considered to be const-qualified. \[-\text{end example}\]\n
### 3.10 Lvalues and rvalues

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

![Figure 1 — Expression category taxonomy](image-url)

---

\(^5\) 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.

§ 3.10
An lvalue (so called, historically, because lvalues could appear on the left-hand side of an assignment expression) designates a function or an object. [Example: If E is an expression of pointer type, then *E is an lvalue expression referring to the object or function to which E points. As another example, the result of calling a function whose return type is an lvalue reference is an lvalue. — end example]

An xvalue (an “eXpiring” value) also refers to an object, usually near the end of its lifetime (so that its resources may be moved, for example). Certain kinds of expressions involving rvalue references (8.3.2) yield xvalues. [Example: The result of calling a function whose return type is an rvalue reference to an object type is an xvalue (5.2.2). — end example]

A glvalue (“generalized” lvalue) is an lvalue or an xvalue.

An rvalue (so called, historically, because rvalues could appear on the right-hand side of an assignment expression) is an xvalue, a temporary object (12.2) or subobject thereof, or a value that is not associated with an object.

A prvalue (“pure” rvalue) is an rvalue that is not an xvalue. [Example: The result of calling a function whose return type is not a reference is a prvalue. The value of a literal such as 12, 7.3e5, or true is also a prvalue. — end example]

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 Clause 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]

Whenever a glvalue appears in a context where a prvalue is expected, the glvalue is converted to a prvalue; see 4.1, 4.2, and 4.3. [Note: An attempt to bind an rvalue reference to an lvalue is not such a context; see 8.5.3. — end note]

The discussion of reference initialization in 8.5.3 and of temporaries in 12.2 indicates the behavior of lvalues and rvalues in other significant contexts.

Unless otherwise indicated (5.2.2), prvalues shall always have complete types or the void type; in addition to these types, glvalues can also have incomplete types. [Note: class and array prvalues can have cv-qualified types; other prvalues always have cv-unqualified types. See Clause 5. — end note]

An lvalue for an object is necessary in order to modify the object except that an rvalue of class type can also be used to modify its referent under certain circumstances. [Example: a member function called for an object (9.3) can modify the object. — end example]

Functions cannot be modified, but pointers to functions can be modifiable.

A pointer to an incomplete type can be modifiable. At some point in the program when the pointed to type is complete, the object at which the pointer points can also be modified.

The referent of a const-qualified expression shall not be modified (through that expression), except that if it is of class type and has a mutable component, that component can be modified (7.1.6.1).

If an expression can be used to modify the object to which it refers, the expression is called modifiable. A program that attempts to modify an object through a nonmodifiable lvalue or rvalue expression is ill-formed.

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:

54 The intent of this list is to specify those circumstances in which an object may or may not be aliased.

§ 3.10

79
the dynamic type of the object,
— a cv-qualified version of the dynamic type of the object,
— a type similar (as defined in 4.4) to the dynamic type of the object,
— a type that is the signed or unsigned type corresponding 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 `char` or `unsigned char` type.

3.11 Alignment

Object types have alignment requirements (3.9.1, 3.9.2) 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 (7.6.2).

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)` (18.2). 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 (7.6.2). 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]

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 (5.3.6). Furthermore, the narrow character types (3.9.1) 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 (7.6.2). — end note]

7 Comparing alignments is meaningful and provides the obvious results:

(7.1) Two alignments are equal when their numeric values are equal.

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

(7.3) When an alignment is larger than another it represents a stricter alignment.

8 [Note: The runtime pointer alignment function (20.7.5) can be used to obtain an aligned pointer within a buffer; the aligned-storage templates in the library (20.10.7.6) can be used to obtain aligned storage. — end note]

9 If a request for a specific extended alignment in a specific context is not supported by an implementation, the program is ill-formed. Additionally, a request for runtime allocation of dynamic storage for which the requested alignment cannot be honored shall be treated as an allocation failure.
Standard conversions

1 Standard conversions are implicit conversions with built-in meaning. Clause 4 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 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 (Clause 5).

(2.2) — When used in the condition of an if statement or iteration statement (6.4, 6.5). The destination type is bool.

(2.3) — When used in the expression of a switch statement. The destination type is integral (6.4).

(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 8.5, 8.5.3.

—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 (8.5).

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 (8.5).

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 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 (8.3.2), 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 12.3. In general, an implicit conversion sequence (13.3.3.1) consists of a standard conversion sequence followed by a user-defined conversion followed by another standard conversion sequence. —end note]
4.1 Lvalue-to-rvalue conversion

A glvalue (3.10) of a non-function, non-array type \( T \) can be converted to a prvalue.\(^{55}\) 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 \( T \).\(^{56}\)

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

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

\((2.2)\) — 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 \( ex \) (3.2),

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

```c
struct S { int n; };
auto f() {
  S x { 1 };
  constexpr S y { 2 };
  return [&](bool b) { return (b ? y : x).n; };
}
auto g = f();
int m = g(false); // undefined behavior due to access of x.n outside its lifetime
int n = g(true);  // OK, does not access y.n
```
—end example] In all other cases, the result of the conversion is determined according to the following rules:

\((2.3)\) — If \( T \) is (possibly cv-qualified) \( \text{std}::\text{nullptr}_t \), the result is a null pointer constant (4.10).

\((2.4)\) — Otherwise, if \( T \) has a class type, the conversion copy-initializes a temporary of type \( T \) from the glvalue and the result of the conversion is a prvalue for the temporary.

\((2.5)\) — Otherwise, if the object to which the glvalue refers contains an invalid pointer value (3.7.4.2, 3.7.4.3), the behavior is implementation-defined.

\((2.6)\) — Otherwise, the value contained in the object indicated by the glvalue is the prvalue result.

\(^3\) [Note: See also 3.10. — end note]

4.2 Array-to-pointer conversion

An lvalue or rvalue of type “array of \( N \ T \)” or “array of unknown bound of \( T \)” can be converted to a prvalue of type “pointer to \( T \)” with the result being a pointer to the first element of the array.

4.3 Function-to-pointer conversion

An lvalue of function type \( T \) can be converted to a prvalue of type “pointer to \( T \)” with the result being a pointer to the function.\(^{57}\)

\(^2\) [Note: See 13.4 for additional rules for the case where the function is overloaded. — end note]

\(^{55}\) 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 3.10.

\(^{56}\) In C++ class prvalues can have cv-qualified types (because they are objects). This differs from ISO C, in which non-lvalues never have cv-qualified types.

\(^{57}\) This conversion never applies to non-static member functions because an lvalue that refers to a non-static member function cannot be obtained.
4.4 Qualification conversions

A cv-decomposition of a type $T$ is a sequence of $cv_i$ and $P_i$ such that

$$T = "cv_0 P_0 cv_1 P_1 \cdots cv_{n-1} P_{n-1} cv_n U"$$

for $n > 0$,

where each $cv_i$ is a set of cv-qualifiers (3.9.3), and each $P_i$ is “pointer to” (8.3.1), “pointer to member of class $C_i$ of type” (8.3.3), “array of $N_i$”, or “array of unknown bound of” (8.3.4). 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, $cv_1, cv_2, \cdots, cv_n$, is called the cv-qualification signature of $T$.

Two types $T_1$ and $T_2$ are \textit{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 $cv^j_i$ denotes the cv-qualifiers in the cv-qualification signature of $T_j$:

\begin{enumerate}
\item $T_1$ and $T_2$ are similar.
\item For every $i > 0$, if \texttt{const} is in $cv^1_i$ then \texttt{const} is in $cv^2_i$, and similarly for \texttt{volatile}.
\item If the $cv^1_1$ and $cv^2_1$ are different, then \texttt{const} is in every $cv^2_k$ for $0 < k < i$.
\end{enumerate}

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

4.5 Integral promotions

A prvalue of an integer type other than \texttt{bool}, \texttt{char16_t}, \texttt{char32_t}, or \texttt{wchar_t} whose integer conversion rank (4.13) is less than the rank of \texttt{int} can be converted to a prvalue of type \texttt{int} if \texttt{int} can represent all the values of the source type; otherwise, the source prvalue can be converted to a prvalue of type \texttt{unsigned int}.

A prvalue of type \texttt{char16_t}, \texttt{char32_t}, or \texttt{wchar_t} (3.9.1) can be converted to a prvalue of the first of the following types that can represent all the values of its underlying type: \texttt{int}, \texttt{unsigned int}, \texttt{long int}, \texttt{unsigned long int}, or \texttt{unsigned long long int}. If none of the types in that list can

\[Note: A prvalue of type “pointer to cv1 T” can be converted to a prvalue of type “pointer to cv2 T” if “cv2 T” is more cv-qualified than “cv1 T”. – end note\]

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

\[Note: Function types (including those used in pointer to member function types) are never cv-qualified (8.3.5). – end note\]
represent all the values of its underlying type, a prvalue of type char16_t, char32_t, or wchar_t can be converted to a prvalue of its underlying type.

3 A prvalue of an unscoped enumeration type whose underlying type is not fixed (7.2) 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 7.2): 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 (4.13) greater than the rank of long long in which all the values of the enumeration can be represented. If there are two such extended types, the signed one is chosen.

4 A prvalue of an unscoped enumeration type whose underlying type is fixed (7.2) can be converted to a prvalue of its underlying type. Moreover, if integral promotion can be applied to its underlying type, a prvalue of an unscoped enumeration type whose underlying type is fixed can also be converted to a prvalue of the promoted underlying type.

5 A prvalue for an integral bit-field (9.6) 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.

4.6 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.

4.7 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 4.12. 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.

4.8 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.
4.9 Floating-integral conversions

1 A prvalue of a floating point type can be converted to a prvalue of an integer type. The conversion truncates; that is, the fractional part is discarded. The behavior is undefined if the truncated value cannot be represented in the destination type. [Note: If the destination type is `bool`, see 4.12. — 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.

4.10 Pointer conversions

1 A null pointer constant is an integer literal (2.13.2) 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 (4.4). 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 result of converting a non-null pointer value of a pointer to object type to a “pointer to cv void” represents the address of the same byte in memory as the original pointer value. The null pointer value is converted to the null pointer value of the destination type.

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 10) of D. If B is an inaccessible (Clause 11) or ambiguous (10.2) 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.

4.11 Pointer to member conversions

1 A null pointer constant (4.10) 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 (4.4).

2 A prvalue of type “pointer to member of B of type cv T”, where B is a class type, can be converted to a prvalue of type “pointer to member of D of type cv T”, where D is a derived class (Clause 10) of B. If B is an inaccessible (Clause 11), ambiguous (10.2), or virtual (10.1) base class of D, or a base class of a virtual base class of D, a program that necessitates this conversion is ill-formed. The result of the conversion refers to the same member as the pointer to member before the conversion took place, but it refers to the base class member as if it were a member of the derived class. The result refers to the member in D’s instance of B. Since the result has type “pointer to member of D of type cv T”, indirection through it with a D object is valid. The result is the same as if indirecting through the pointer to member of B with the B subobject of D. The null member pointer value is converted to the null member pointer value of the destination type.  

59) The rule for conversion of pointers to members (from pointer to member of base to pointer to member of derived) appears
4.12 Boolean conversions

1 A prvalue of arithmetic, unscoped enumeration, pointer, or pointer to member type can be converted to a prvalue of type bool. A zero value, null pointer value, or null member pointer value is converted to false; any other value is converted to true. For direct-initialization (8.5), a prvalue of type std::nullptr_t can be converted to a prvalue of type bool; the resulting value is false.

4.13 Integer conversion 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 (3.9.1).

(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 (4.5) and the usual arithmetic conversions (Clause 5). — end note]
5 Expressions [expr]

1 [Note: Clause 5 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 9) or enumeration type (7.2). Uses of overloaded operators are transformed into function calls as described in 13.5. Overloaded operators obey the rules for syntax specified in Clause 5, but the requirements of operand type, value category, and evaluation order are replaced by the rules for function call. Relations between operators, such as ++a meaning a+=1, are not guaranteed for overloaded operators (13.5), and are not guaranteed for operands of type bool. —end note]

3 Clause 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 Clause 5; see 13.3.1.2, 13.6.

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: most existing implementations of C++ ignore integer overflows. Treatment of division by zero, forming a remainder using a zero divisor, and all floating point exceptions vary among machines, and is usually adjustable by a library function. —end note]

5 If an expression initially has the type “reference to T” (8.3.2, 8.5.3), the type is adjusted to T prior to any further analysis. The expression designates the object or function denoted by the reference, and the expression is an lvalue or an xvalue, depending on the expression.

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

7 [Note: An expression is an xvalue if it is:

(7.1) — the result of calling a function, whether implicitly or explicitly, whose return type is an rvalue reference to object type,

(7.2) — a cast to an rvalue reference to object type,

(7.3) — a class member access expression designating a non-static data member of non-reference type in which the object expression is an xvalue, or

(7.4) — 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]

[Example:]

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

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

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

--- end example

8 In some contexts, *unevaluated operands* appear (5.2.8, 5.3.3, 5.3.7, 7.1.6.2). An unevaluated operand is not evaluated. An unevaluated operand is considered a full-expression. [Note: In an unevaluated operand, a non-static class member may be named (5.1) and naming of objects or functions does not, by itself, require that a definition be provided (3.2). — end note]

9 Whenever a glvalue expression appears as an operand of an operator that expects a prvalue for that operand, the lvalue-to-rvalue (4.1), array-to-pointer (4.2), or function-to-pointer (4.3) standard conversions are applied to convert the expression to a prvalue. [Note: because cv-qualifiers are removed from the type of an expression of non-class type when the expression is converted to a prvalue, an lvalue expression of type const int can, for example, be used where a prvalue expression of type int is required. — end note]

10 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:

--- (10.1) ---
8 If either operand is of scoped enumeration type (7.2), no conversions are performed; if the other operand does not have the same type, the expression is ill-formed.

--- (10.2) ---
8 If either operand is of type long double, the other shall be converted to long double.

--- (10.3) ---
8 Otherwise, if either operand is double, the other shall be converted to double.

--- (10.4) ---
8 Otherwise, if either operand is float, the other shall be converted to float.

--- (10.5) ---
8 Otherwise, the integral promotions (4.5) shall be performed on both operands.61 Then the following rules shall be applied to the promoted operands:

--- (10.5.1) ---
8 If both operands have the same type, no further conversion is needed.

--- (10.5.2) ---
8 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.

--- (10.5.3) ---
8 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.

--- (10.5.4) ---
8 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.

--- (10.5.5) ---
8 Otherwise, both operands shall be converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

61) As a consequence, operands of type bool, char16_t, char32_t, wchar_t, or an enumerated type are converted to some integral type.

Expressions 89
In some contexts, an expression only appears for its side effects. Such an expression is called a **discarded-value expression**. The expression is evaluated and its value is discarded. The array-to-pointer (4.2) and function-to-pointer (4.3) standard conversions are not applied. The lvalue-to-rvalue conversion (4.1) is applied if and only if the expression is a glvalue of volatile-qualified type and it is one of the following:

1. \((\text{expression})\), where \text{expression} is one of these expressions,
2. \(\text{id-expression} \) (5.1.1),
3. \(\text{subscripting} \) (5.2.1),
4. \(\text{class member access} \) (5.2.5),
5. \(\text{indirection} \) (5.3.1),
6. \(\text{pointer-to-member operation} \) (5.5),
7. \(\text{conditional expression} \) (5.16) where both the second and the third operands are one of these expressions, or
8. \(\text{comma expression} \) (5.19) where the right operand is one of these expressions.

\[ \text{Note: Using an overloaded operator causes a function call; the above covers only operators with built-in meaning. If the lvalue is of class type, it must have a volatile copy constructor to initialize the temporary that is the result of the lvalue-to-rvalue conversion. — end note} \]

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

The **cv-combined type** of two types \(T_1\) and \(T_2\) is a type \(T_3\) similar to \(T_1\) whose cv-qualification signature (4.4) is:

1. \(\text{for every } j > 0, \text{cv}_{3,j}\) is the union of \(\text{cv}_{1,j}\) and \(\text{cv}_{2,j}\);
2. \(\text{if the resulting } \text{cv}_{3,j}\) is different from \(\text{cv}_{1,j}\) or \(\text{cv}_{2,j}\), then \text{const} is added to every \(\text{cv}_{3,k}\) for \(0 < k < j\).

\[ \text{Note: Given similar types } T_1 \text{ and } T_2, \text{ this construction ensures that both can be converted to } T_3. \text{ — end note] The composite pointer type of two operands } p_1 \text{ and } p_2 \text{ having types } T_1 \text{ and } T_2, \text{ respectively, where at least one is a pointer or pointer to member type or std::nullptr_t, is:} \]

1. \(\text{if both } p_1 \text{ and } p_2 \text{ are null pointer constants, std::nullptr_t;}
2. \(\text{if either } p_1 \text{ or } p_2 \text{ is a null pointer constant, } T_2 \text{ or } T_1, \text{ respectively;}
3. \(\text{if } T_1 \text{ or } T_2 \text{ is “pointer to cv1 void” and the other type is “pointer to cv2 T”, “pointer to cv12 void”, where cv12 is the union of cv1 and cv2;}
4. \(\text{if } T_1 \text{ is “pointer to cv1 C1” and } T_2 \text{ is “pointer to cv2 C2”, where } C_1 \text{ is reference-related to } C_2 \text{ or } C_2 \text{ is reference-related to } C_1 \text{(8.5.3), the cv-combined type of } T_1 \text{ and } T_2 \text{ or the cv-combined type of } T_2 \text{ and } T_1, \text{ respectively;}
5. \(\text{if } T_1 \text{ is “pointer to member of } C_1 \text{ of type cv1 U1” and } T_2 \text{ is “pointer to member of } C_2 \text{ of type cv2 U2” where } C_1 \text{ is reference-related to } C_2 \text{ or } C_2 \text{ is reference-related to } C_1 \text{(8.5.3), the cv-combined type of } T_2 \text{ and } T_1 \text{ or the cv-combined type of } T_1 \text{ and } T_2, \text{ respectively;}
6. \(\text{if } T_1 \text{ and } T_2 \text{ are similar types (4.4), the cv-combined type of } T_1 \text{ and } T_2;\)

---

\(^{62}\) The cast and assignment operators must still perform their specific conversions as described in 5.4, 5.2.9 and 5.18.
— otherwise, a program that necessitates the determination of a composite pointer type is ill-formed.

[Example:

typedef void *p;
typedef const int *q;
typedef int **pi;
typedef const int **pci;

The composite pointer type of \texttt{p} and \texttt{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]

5.1 Primary expressions \[expr.prim\]

5.1.1 General \[expr.prim.general\]

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

1 A literal is a primary expression. Its type depends on its form (2.13). A string literal is an lvalue; all other literals are prvalues.

2 The keyword \texttt{this} names a pointer to the object for which a non-static member function (9.3.2) is invoked or a non-static data member’s initializer (9.2) is evaluated.

3 If a declaration declares a member function or member function template of a class \texttt{X}, the expression \texttt{this} is a prvalue of type “pointer to \texttt{cv-qualifier-seq X}” between the optional \texttt{cv-qualifier-seq} and the end of the \texttt{function-definition}, \texttt{member-declarator}, or \texttt{declarator}. It shall not appear before the optional \texttt{cv-qualifier-seq} and it shall not appear within the declaration of a static member function (although its type and value category are defined within a static member function as they are within a non-static member function). [Note: this is because declaration matching does not occur until the complete declarator is known. — end note] Unlike the object expression in other contexts, \texttt{*this} is not required to be of complete type for purposes of class member access (5.2.5) outside the member function body. [Note: only class members declared prior to the declaration are visible. — end note] [Example:

struct A {
    char g();
    template<
        class T>
    auto f(T t) -> decltype(t + g())
        { return t + g(); }
};
template auto A::f(int t) -> decltype(t + g());
Otherwise, if a member-declarator declares a non-static data member (9.2) of a class X, the expression this is a prvalue of type “pointer to X” within the optional brace-or-equal-initializer. 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 brace-or-equal-initializer

    void f() {
        int b[sizeof(*this)];    // OK

        struct Inner {
            int c[sizeof(*this)];    // error: not inside a member function of Inner
        }
    }
};
```

— end example ]

A parenthesized expression is a primary expression whose type and value are identical to those of the enclosed expression. The presence of parentheses does not affect whether the expression is an lvalue. The parenthesized expression can be used in exactly the same contexts as those where the enclosed expression can be used, and with the same meaning, except as otherwise indicated.

An id-expression is a restricted form of a primary-expression. [ Note: an id-expression can appear after . and -> operators (5.2.5). — end note ]

An identifier is an id-expression provided it has been suitably declared (Clause 7). [ Note: for operator-function-ids, see 13.5; for conversion-function-ids, see 12.3.2; for literal-operator-ids, see 13.5.8; for template-ids, see 14.2. A class-name or decltype-specifier prefixed by ~ denotes a destructor; see 12.4. 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 (9.3.1). — end note ] The type of the expression is the type of the identifier. The result is the entity denoted by the identifier. The result is an lvalue if the entity is a function, variable, or data member and a prvalue otherwise.
qualified-id:
  nested-name-specifier template opt unqualified-id

nested-name-specifier:
  ::
  type-name ::
  namespace-name ::
  decltype-specifier ::
  nested-name-specifier identifier ::
  nested-name-specifier template opt simple-template-id ::

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

9 A nested-name-specifier that denotes a class, optionally followed by the keyword template (14.2), and then
followed by the name of a member of either that class (9.2) or one of its base classes (Clause 10), is a
qualified-id; 3.4.3.1 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 (3.3.7). — end note] Where class-name ::~ class-name is
used, the two class-names shall refer to the same class; this notation names the destructor (12.4). 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 (9.1). — end note]

10 The nested-name-specifier :: names the global namespace. A nested-name-specifier that names a
namespace (7.3), 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; 3.4.3.2 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.

11 A nested-name-specifier that denotes an enumeration (7.2), 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.

12 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.

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

(13.1) — as part of a class member access (5.2.5) in which the object expression refers to the member’s class63
or a class derived from that class, or

(13.2) — to form a pointer to member (5.3.1), or

(13.3) — if that id-expression denotes a non-static data member and it appears in an unevaluated operand.

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

  — end example]

63) This also applies when the object expression is an implicit (*this) (9.3.1).
5.1.2 Lambda expressions  

Lambda expressions provide a concise way to create simple function objects.  

```cpp
#include <algorithm>
#include <cmath>

void abssort(float* x, unsigned N) {
    std::sort(x, x + N, 
        [](float a, float b) {
            return std::abs(a) < std::abs(b);
        });
}
```

— end example ]

```
lambda-expression:
    lambda-introducer lambda-declarator_opt compound-statement
lambda-introducer:
    [ lambda-capture_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

init-capture:
    identifier initializer
    & identifier initializer

lambda-declarator:
    ( parameter-declaration-clause ) mutable_opt
    exception-specification_opt attribute-specifier-seq_opt trailing-return-type_opt
```

2 The evaluation of a lambda-expression results in a prvalue temporary (12.2). This temporary is called the closure object. A lambda-expression shall not appear in an unevaluated operand (Clause 5), in a template-argument, in an alias-declaration, in a typedef declaration, or in the declaration of a function or function template outside its function body and default arguments. [ Note: The intention is to prevent lambdas from appearing in a signature. — end note ] [ Note: A closure object behaves like a function object (20.9). — end note ]

3 The type of the 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. This class type is neither an aggregate (8.5.1) nor a literal type (3.9). 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 (3.4.2). The parameter types of a lambda-declarator do not affect these associated namespaces and classes. — end note ] 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:

(3.1) — the size and/or alignment of the closure type,
(3.2) — whether the closure type is trivially copyable (Clause 9),
(3.3) — whether the closure type is a standard-layout class (Clause 9), or
(3.4) — whether the closure type is a POD class (Clause 9).

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

4 If a lambda-expression does not include a lambda-declarator, it is as if the lambda-declarator were (). The lambda return type is auto, which is replaced by the trailing-return-type if provided and/or deduced from return statements as described in 7.1.6.4. [Example:

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

—end example] This function call operator or operator template is declared const (9.3.1) if and only if the lambda-expression’s parameter-declaration-clause is not followed by mutable. It is neither virtual nor declared volatile. Any exception-specification 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. [Note: Names referenced in the lambda-declarator are looked up in the context in which the lambda-expression appears. — end note]

5 The closure type for a non-generic lambda-expression has a public inline function call operator (13.5.4) 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 (14.5.2) whose template-parameter-list consists of 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 (8.3.5). 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. [Example:

```c
auto glambda = [](auto a, auto&& b) { return a < b; };  // OK
auto vglambda = [](auto printer) {
    return [=](auto&& ... ts) {
        printer(std::forward<decltype(ts)>(ts)...);
        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 closure type for a non-generic lambda-expression with no lambda-capture has a public non-virtual non-explicit const conversion function to pointer to function with C++ language linkage (7.5) having the same parameter and return types as the closure type’s function call operator. The value returned by this conversion function shall be the address of a function that, when invoked, has the same effect as invoking the closure type’s function call operator. For a generic lambda with no lambda-capture, the closure type has a public non-virtual non-explicit const conversion function template to pointer to function. The conversion function template has the same invented template-parameter-list, and the pointer to function has the same parameter types, as the function call operator template. The return type of the pointer to function shall behave as if it were a decltype-specifier denoting the return type of the corresponding function call operator template specialization. [Note: If the generic lambda has no trailing-return-type or the trailing-return-type contains a placeholder type, return type deduction of the corresponding function call operator template specialization has to be done. The corresponding specialization is that instantiation of the function call operator template with the same template arguments as those deduced for the conversion function template. Consider the following:

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

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

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

—end note] [Example:

```cpp
void f1(int (*)(int)) { }
void f2(char (*)(int)) { }
void g(int (*)(int)) { } // #1
void g(char (*)(char)) { } // #2
void h(int (*)(int)) { } // #3
void h(char (*)(int)) { } // #4

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

—end example] The value returned by any given specialization of this conversion function template shall be the address of a function that, when invoked, has the same effect as invoking the generic lambda’s corresponding function call operator template specialization. [Note: This will result in the implicit instantiation
of the generic lambda’s body. The instantiated generic lambda’s return type and parameter types shall match the return type and parameter types of the pointer to function. —end note] [Example:

```cpp
auto GL = [] (auto a) { std::cout << a; return a; };
int (*GL_int)(int) = GL; // OK: through conversion function template
GL_int(3); // OK: same as GL(3)
```

—end example]

7 The lambda-expression’s compound-statement yields the function-body (8.4) of the function call operator, but for purposes of name lookup (3.4), determining the type and value of this (9.3.2) and transforming id-expressions referring to non-static class members into class member access expressions using (*this) (9.3.1), the compound-statement is considered in the context of the lambda-expression. [Example:

```cpp
struct S1 {
  int x, y;
  int operator()(int);
  void f() {
    [=] ()->int {
      return operator()(this->x + y); // equivalent to S1::operator()(this->x + (*this).y)
                                // this has type S1*
    }
  }
};
```

—end example] Further, a variable __func__ is implicitly defined at the beginning of the compound-statement of the lambda-expression, with semantics as described in 8.4.1.

8 If a lambda-capture includes a capture-default that is & and 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”. 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) {
  [& , i]{}; // OK
  [& , &i]{}; // error: i preceded by & when & is the default
  [= , this]{}; // error: this when = is the default
  [i, i]{}; // error: i repeated
}
```

—end example]

9 A lambda-expression whose smallest enclosing scope is a block scope (3.3.3) is a local lambda expression; any other lambda-expression shall not have a capture-default or simple-capture in its lambda-introducer. The reaching scope of a local lambda expression is the set of enclosing scopes up to and including the innermost enclosing function and its parameters. [Note: This reaching scope includes any intervening lambda-expressions. —end note]

10 The identifier in a simple-capture is looked up using the usual rules for unqualified name lookup (3.4.1); each such lookup shall find an entity. An entity that is designated by a simple-capture is said to be explicitly captured, and shall be this or a variable with automatic storage duration declared in the reaching scope of the local lambda expression.

11 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:

§ 5.1.2
— 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:

```cpp
int x = 4;
auto y = [&r = x, x = x+1]()->int {
  r += 2;
  return x+2;
}(); // Updates x to 6, and initializes y to 7.

— end example]

A lambda-expression with an associated capture-default that does not explicitly capture this or a variable
with automatic storage duration (this excludes any id-expression that has been found to refer to an init-
capture’s associated non-static data member), is said to implicitly capture the entity (i.e., this or a variable)
if the compound-statement:

— odr-uses (3.2) the entity, or
— names the entity in a potentially-evaluated expression (3.2) where the enclosing full-expression depends
on a generic lambda parameter declared within the reaching scope of the lambda-expression.

[Example:

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

  auto g2 = [=](auto a) {
    int selector[sizeof(a) == 1 ? 1 : 2]{};
    f(x, selector); // OK: is a dependent expression, so captures x
  };
}

— end example] All such implicitly captured entities shall be declared within the reaching scope of the
lambda expression. [Note: The implicit capture of an entity by a nested lambda-expression can cause its
implicit capture by the containing lambda-expression (see below). Implicit odr-uses of this or a variable
in implicit capture. — end note]

An entity is captured if it is captured explicitly or implicitly. An entity captured by a lambda-expression
is odr-used (3.2) in the scope containing the lambda-expression. If this is captured by a local lambda
expression, its nearest enclosing function shall be a non-static member function. If a lambda-expression
or an instantiation of the function call operator template of a generic lambda odr-uses (3.2) this or a
variable with automatic storage duration from its reaching scope, that entity shall be captured by the
lambda-expression. If a lambda-expression captures an entity and that entity is not defined or captured in
the immediately enclosing lambda expression or function, the program is ill-formed. [Example:
A lambda-expression appearing in a default argument shall not implicitly or explicitly capture any entity.

Example:

```cpp
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
}
```

An entity is captured by copy if it is implicitly captured and the capture-default is = or if it is explicitly captured with a capture that is not of the form & 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 type of the corresponding captured entity if the entity is not a reference to an object, or the referenced type otherwise. [Note: If the captured entity is a reference to a function, the corresponding data member is also a reference to a function. —end note] A member of an anonymous union shall not be captured by copy.

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. A member of an anonymous union shall not be captured by reference.
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:

(17.1) — if \( m1 \) captures the entity by copy, \( m2 \) captures the corresponding non-static data member of \( m1 \)'s closure type;

(17.2) — 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]

Every id-expression within the compound-statement of a lambda-expression that is an odr-use (3.2) 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. Furthermore, such an id-expression does not cause the implicit capture of the entity. — end note] If this is captured, each odr-use of this is transformed into an access to the corresponding unnamed data member of the closure type, cast (5.4) to the type of this. [Note: The cast ensures that the transformed expression is a prvalue. — end note] [Example:

```cpp
void f(const int*);
void g() {
    const int N = 10;
    [=] {
        int arr[N];
        // OK: not an odr-use, refers to automatic variable
        f(&N);
        // OK: causes N to be captured; &N points to the
        // corresponding member of the closure type
    };
}
```

— end example]

Every occurrence of `decltype((x))` where \( x \) is a possibly parenthesized id-expression that names an entity of automatic storage duration is treated as if \( x \) were transformed into an access to a corresponding data member of the closure type that would have been declared if \( x \) were an odr-use of the denoted entity. [Example:

```cpp
void f3() {
    float x, &r = x;
    [=] {
        // x and r are not captured (appearance in a decltype operand is not an odr-use)
        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 & (transformation not considered)
        decltype((r)) r2 = y2;  // r2 has type float const &
    }
}```

§ 5.1.2
The closure type associated with a lambda-expression has no default constructor and a deleted copy assignment operator. It has a defaulted copy constructor and a defaulted move constructor (12.8). [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 (12.4).

A member of a closure type shall not be explicitly instantiated (14.7.1), explicitly specialized (14.7.2), or named in a friend declaration (11.3).

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 an 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 (14.5.3). 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]

### 5.1.3 Fold expressions

A fold expression performs a fold of a template parameter pack (14.5.3) 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 (14.5.3).

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:

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

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

5.2 Postfix expressions [expr.post]

1 Postfix expressions group left-to-right.

```cpp
postfix-expression:
  primary-expression
postfix-expression [ expression ]
postfix-expression [ 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 )
```

evaluation-list:
  initializer-list

```cpp
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 (14.2). — end note]`

5.2.1 Subscripting [expr.sub]

1 A postfix expression followed by an expression in square brackets is a postfix expression. One of the expressions shall have the type “array of `T`” or “pointer to `T`” and the other shall have unscoped enumeration or integral type. The result is of type “`T`.” The type “`T`” shall be a completely-defined object type.⁶⁴ The

⁶⁴ This is true even if the subscript operator is used in the following common idiom: `&x[0]`.

§ 5.2.1 102
expression \( E_1[E_2] \) is identical (by definition) to \(*((E_1)\ast(E_2))\) \[ Note: \text{see 5.3 and 5.7 for details of \(*\) and \+ and 8.3.4 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. 

2 A \textit{braced-init-list} shall not be used with the built-in subscript operator.

### 5.2.2 Function call \[expr.call\]

1 A function call is a postfix expression followed by parentheses containing a possibly empty, comma-separated list of \textit{initializer-clauses} which constitute the arguments to the function. The postfix expression shall have function type or pointer to function 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 \(4.3\) is suppressed on the postfix expression), or it shall have pointer to function type. Calling a function through an expression whose function type has a language linkage that is different from the language linkage of the function type of the called function’s definition is undefined \(7.5\). For a call to a non-static member function, the postfix expression shall be an implicit \(9.3.1, 9.4\) or explicit class member access \(5.2.5\) whose \textit{id-expression} is a function member name, or a pointer-to-member expression \(5.5\) 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 \texttt{this}. \[ Note: \text{a member function call of the form } f() \text{ is interpreted as } (*this).f() \text{ (see 9.3.1). — end note}\] If a function or member function name is used, the name can be overloaded \(\text{Clause 13}\), in which case the appropriate function shall be selected according to the rules in \(13.3\). If the selected function is non-virtual, or if the \textit{id-expression} in the class member access expression is a \textit{qualified-id}, that function is called. Otherwise, its final overrider \(10.3\) in the dynamic type of the object expression is called; such a call is referred to as a \textit{virtual function call}. \[ Note: \text{the dynamic type is the type of the object referred to by the current value of the object expression. 12.7 describes the behavior of virtual function calls when the object expression refers to an object under construction or destruction. — end note}\]

2 \[ Note: \text{If a function or member function name is used, and name lookup \(3.4\) 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 \textit{postfix-expression} designates a destructor \(12.4\), the type of the function call expression is \texttt{void}; otherwise, the type of the function call expression is the return type of the statically chosen function (i.e., ignoring the \texttt{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 \texttt{cv void}.

4 When a function is called, each parameter \(8.3.5\) shall be initialized \(8.5, 12.8, 12.1\) with its corresponding argument. \[ Note: \text{Such initializations are indeterminately sequenced with respect to each other \(1.9\) — end note}\] If the function is a non-static member function, the \texttt{this} parameter of the function \(9.3.2\) shall be initialized with a pointer to the object of the call, converted as if by an explicit type conversion \(5.4\). \[ Note: \text{There is no access or ambiguity checking on this conversion; the access checking and disambiguation are done as part of the (possibly implicit) class member access operator. See 10.2, 11.2, and 5.2.5. — end note}\] When a function is called, the parameters that have object type shall have completely-defined object type. \[ Note: \text{this still allows a parameter to be a pointer or reference to an incomplete class type. However, it prevents a passed-by-value parameter to have an incomplete class type. — end note}\]

During the initialization of a parameter, an implementation may avoid the construction of extra temporaries by combining the conversions on the associated argument and/or the construction of temporaries with the initialization of the parameter (see \(12.2\)). The lifetime of a parameter ends when the function in which it is defined returns. The initialization and destruction of each parameter occurs within the context of the calling function. \[ Example: \text{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 \(\text{Clause 15}\) with a handler that could handle the exception, this handler is not considered. — end example}\] The value of a function call is the value returned by the called function.
except in a virtual function call if the return type of the final overrider is different from the return type of
the statically chosen function, the value returned from the final overrider is converted to the return type of
the statically chosen function.

[Note: a function can change the values of its non-const parameters, but these changes cannot affect the
values of the arguments except where a parameter is of a reference type (8.3.2); if the reference is to a
const-qualified type, \texttt{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 \texttt{const} reference type a temporary object is introduced if
needed (7.1.6, 2.13, 2.13.5, 8.3.4, 12.2). In addition, it is possible to modify the values of nonconstant objects
through pointer parameters. —end note]

A function can be declared to accept fewer arguments (by declaring default arguments (8.3.6)) or more
arguments (by using the ellipsis, ..., or a function parameter pack (8.3.5)) than the number of parameters
in the function definition (8.4). [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 \texttt{va\_arg} (18.10). [Note: This paragraph does not
apply to arguments passed to a function parameter pack. Function parameter packs are expanded during
template instantiation (14.5.3), thus each such argument has a corresponding parameter when a function
template specialization is actually called. —end note] The lvalue-to-rvalue (4.1), array-to-pointer (4.2), and
function-to-pointer (4.3) standard conversions are performed on the argument expression. An argument that
has (possibly cv-qualified) type \texttt{std::nullptr\_t} is converted to type \texttt{void\*} (4.10). After these conversions,
if the argument does not have arithmetic, enumeration, pointer, pointer to member, or class type, the
program is ill-formed. Passing a potentially-evaluated argument of class type (Clause 9) 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 (4.5), or a floating point type that is subject
to the floating point promotion (4.6), the value of the argument is converted to the promoted type before
the call. These promotions are referred to as the default argument promotions.

[Note: The evaluations of the postfix expression and of the arguments are all unsequenced relative to one
another. All side effects of argument evaluations are sequenced before the function is entered (see 1.9).
—end note]

Recursive calls are permitted, except to the \texttt{main} function (3.6.1).

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.

If a function call is a prvalue of object type:

(11.1) — if the function call is either

(11.1.1) — the operand of a \texttt{decltype\_specifier} or

(11.1.2) — the right operand of a comma operator that is the operand of a \texttt{decltype\_specifier},

a temporary object is not introduced 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. This is true regardless of
whether the expression uses function call notation or operator notation (13.3.1.2). —end note] [Note:
unlike the rule for a \texttt{decltype\_specifier} that considers whether an \texttt{id\_expression} is parenthesized (7.1.6.2),
parentheses have no special meaning in this context. —end note]

(11.2) — otherwise, the type of the prvalue shall be complete.
§ 5.2.3  Explicit type conversion (functional notation)  

A *simple-type-specifier* (7.1.6.2) or *typename-specifier* (14.6) followed by a parenthesized *expression-list* constructs a value of the specified type given the expression list. If the expression list is a single expression, the type conversion expression is equivalent (in definedness, and if defined in meaning) to the corresponding cast expression (5.4). If the type specified is a class type, the class type shall be complete. If the expression list specifies more than a single value, the type shall be a class with a suitably declared constructor (8.5, 12.1), and the expression \( T(x_1, x_2, \ldots) \) is equivalent in effect to the declaration \( T(t(x_1, x_2, \ldots)) \); for some invented temporary variable \( t \), with the result being the value of \( t \) as a prvalue.

The expression \( T() \), where \( T \) is a *simple-type-specifier* or *typename-specifier* for a non-array complete object type or the (possibly cv-qualified) \( \text{void} \) type, creates a prvalue of the specified type, whose value is that produced by value-initializing (8.5) an object of type \( T \); no initialization is done for the \( \text{void()} \) case. [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 (Clause 5). — end note]

Similarly, a *simple-type-specifier* or *typename-specifier* followed by a *braced-init-list* creates a temporary object of the specified type direct-list-initialized (8.5.4) with the specified *braced-init-list*, and its value is that temporary object as a prvalue.

§ 5.2.4  Pseudo destructor call  

The use of a *pseudo-destructor-name* after a dot . or arrow \( \rightarrow \) 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 \( \text{void} \). 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

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

shall designate the same scalar type.

§ 5.2.5  Class member access  

A postfix expression followed by a dot . or an arrow \( \rightarrow \), optionally followed by the keyword *template* (14.2), 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.

For the first option (dot) the first expression shall have complete class type. For the second option (arrow) the first expression shall have pointer to complete class type. The expression \( E_1\rightarrow E_2 \) is converted to the equivalent form \( (*)E_1).E_2 \); the remainder of 5.2.5 will address only the first option (dot).66 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 9), the name of a class is also considered a nested member of that class. — end note] [Note: 3.4.5 describes how names are looked up after the . and \( \rightarrow \) operators. — end note]

Abbreviating *postfix-expression.id-expression* as \( E_1.E_2 \), \( E_1 \) is called the *object expression*. The type and value category of \( E_1.E_2 \) are determined as follows. In the remainder of 5.2.5, \( cq \) represents either \( \text{const} \) or the absence of \( \text{const} \) and \( vq \) represents either \( \text{volatile} \) or the absence of \( \text{volatile} \). \( cv \) represents an arbitrary set of cv-qualifiers, as defined in 3.9.3.

65) If the class member access expression is evaluated, the subexpression evaluation happens even if the result is unnecessary to determine the value of the entire postfix expression, for example if the *id-expression* denotes a static member.

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

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

(4.2) — If \(E_2\) is a non-static data member and the type of \(E_1\) is “\(\text{cq1} \text{vq1} X\)”, and the type of \(E_2\) is “\(\text{cq2} \text{vq2} T\)”, the expression designates the named member of the object designated by the first expression. If \(E_1\) is an lvalue, then \(E_1.E_2\) is an lvalue; otherwise \(E_1.E_2\) is an xvalue. Let the notation \(\text{vq1}\) stand for the “union” of \(\text{vq1}\) and \(\text{vq2}\); that is, if \(\text{vq1}\) or \(\text{vq2}\) is \(\text{volatile}\), then \(\text{vq1}\) is \(\text{volatile}\). Similarly, let the notation \(\text{cq12}\) stand for the “union” of \(\text{cq1}\) and \(\text{cq2}\); that is, if \(\text{cq1}\) or \(\text{cq2}\) is \(\text{const}\), then \(\text{cq12}\) is \(\text{const}\). If \(E_2\) is declared to be a \textit{mutable} member, then the type of \(E_1.E_2\) is “\(\text{cq12} \text{vq12} T\)". If \(E_2\) is not declared to be a \textit{mutable} member, then the type of \(E_1.E_2\) is “\(\text{cq12} \text{vq12} T\)".

(4.3) — If \(E_2\) is a (possibly overloaded) member function, function overload resolution (13.3) is used to determine whether \(E_1.E_2\) 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 \(E_2\) is “function of parameter-type-list returning \(T\)”, then \(E_1.E_2\) is an lvalue; the expression designates the static member function. The type of \(E_1.E_2\) is the same type as that of \(E_2\), namely “function of parameter-type-list returning \(T\)".

(4.3.2) — Otherwise, if \(E_1.E_2\) refers to a non-static member function and the type of \(E_2\) is “function of parameter-type-list \(cv\) ref-qualifier_opt returning \(T\)”, then \(E_1.E_2\) is a prvalue. The expression designates a non-static member function. The expression can be used only as the left-hand operand of a member function call (9.3). \[Note: Any redundant set of parentheses surrounding the expression is ignored (5.1). \—end note\] The type of \(E_1.E_2\) is “function of parameter-type-list \(cv\) returning \(T\)".

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

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

If \(E_2\) is a non-static data member or a non-static member function, the program is ill-formed if the class of which \(E_2\) is directly a member is an ambiguous base (10.2) of the naming class (11.2) of \(E_2\). \[Note: The program is also ill-formed if the naming class is an ambiguous base of the class type of the object expression; see 11.2. \—end note\]

### 5.2.6 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 or a pointer to a complete object type. The value of the operand object is modified by adding 1 to it, unless the object is of type \texttt{bool}, in which case it is set to \texttt{true}. \[Note: this use is deprecated, see Annex D. \—end note\] 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 5.7 and 5.18.

The operand of postfix -- is decremented analogously to the postfix ++ operator, except that the operand shall not be of type \texttt{bool}. \[Note: For prefix increment and decrement, see 5.3.2. \—end note\]
5.2.7 Dynamic cast

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 (5.2.11).

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 an expression having a complete class type, and the result is an xvalue of the type referred to by `T`.

If the type of `v` is the same as `T`, 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).

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

If `T` is “pointer to `cv1` `B`” and `v` has type “pointer to `cv2` `D`” such that `B` is a base class of `D`, the result is a pointer to the unique `B` subobject of the `D` object pointed to by `v`. 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`. The result is an lvalue if `T` is an lvalue reference, or an xvalue if `T` is an rvalue reference. 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:

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

—end example]

Otherwise, `v` shall be a pointer to or a glvalue of a polymorphic type (10.3).

If `T` is “pointer to `cv` void,” then the result is a pointer to the most derived object pointed to by `v`. Otherwise, a run-time check is applied to see if the object pointed or referred to by `v` can be converted to the type pointed or referred to by `T`.

If `C` is the class type to which `T` points or refers, the run-time check logically executes as follows:

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

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

3. Otherwise, the run-time check fails.

The value of a failed cast to pointer type is the null pointer value of the required result type. A failed cast to reference type throws an exception (15.1) of a type that would match a handler (15.3) of type `std::bad_cast` (18.7.2).

[Example:

```
class A { virtual void f(); };  
class B { virtual void g(); };  
class D : public virtual A, private B { };  
void g() {
```

67) The most derived object (1.8) pointed or referred to by `v` can contain other `B` objects as base classes, but these are ignored.
D d;
B* bp = (B*)&d;  // cast needed to break protection
A* ap = &d;    // public derivation, no cast needed
D& dr = dynamic_cast<D&>(*bp);  // fails
ap = dynamic_cast<A*>(bp);    // fails
bp = dynamic_cast<B*>(ap);    // succeeds
ap = dynamic_cast<A*>(&d);    // succeeds
bp = dynamic_cast<B*>(&d);    // ill-formed (not a run-time check)
}

class E : public D, public B { }
class F : public E, public D { }
void h() {
    F f;
    A* ap = &f;    // succeeds: finds unique A
    D* dp = dynamic_cast<D*>(ap);  // fails: yields 0
    // f has two D subobjects
    E* ep = (E*)ap;  // ill-formed: cast from virtual base
    E* ep1 = dynamic_cast<E*>(ap);  // succeeds
}

—end example] [Note: 12.7 describes the behavior of a dynamic_cast applied to an object under construction or destruction. —end note]

5.2.8 Type identification [expr.typeid]

1 The result of a typeid expression is an lvalue of static type const std::type_info (18.7.1) 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 18.7.1.68 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 (10.3), the result refers to a std::type_info object representing the type of the most derived object (1.8) (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 (4.10), the typeid expression throws an exception (15.1) of a type that would match a handler of type std::bad_typeid exception (18.7.3).

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 (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) conversions are not applied to the expression. If the type of the expression is a class type, the class shall be completely-defined. The expression is an unevaluated operand (Clause 5).

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:

    class D { /* ... */ }
    D d1;

68) The recommended name for such a class is extended_type_info.
69) If p is an expression of pointer type, then *p, (**p), *(p), (**(p)), *((p)), and so on all meet this requirement.
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>` (18.7.1) is not included prior to a use of typeid, the program is ill-formed.

7 [ Note: 12.7 describes the behavior of typeid applied to an object under construction or destruction. — end note ]

5.2.9 Static cast

1 The result of the expression `static_cast<T>(v)` is the result of converting the expression v to type T. If T is an lvalue reference type or an rvalue reference to function type, the result is an lvalue; if T is an rvalue reference to type object type, the result is an xvalue; otherwise, the result is a prvalue. The static_cast operator shall not cast away constness (5.2.11).

2 An lvalue of type “cv1 B,” where B is a class type, can be cast to type “reference to cv2 D,” where D is a class derived (Clause 10) from B, if a valid standard conversion from “pointer to D” to “pointer to B” exists (4.10), cv2 is the same cv-qualification as, or greater cv-qualification than, cv1, and D is neither a virtual base class of D nor a base class of a virtual base class of D. The result has type “cv2 D.” An xvalue of type “cv1 B” may 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 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 ]

3 A glvalue, class prvalue, or array prvalue of type “cv1 T1” can be cast to type “rvalue reference to cv2 T2” if “cv2 T2” is reference-compatible with “cv1 T1” (8.5.3). 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 (4.1) is applied to the bit-field and the resulting prvalue is used as the expression of the static_cast for the remainder of this section. If T2 is an inaccessible (Clause 11) or ambiguous (10.2) base class of T1, a program that necessitates such a cast is ill-formed.

4 An expression e can be explicitly converted to a type T using a static_cast of the form `static_cast<T>(e)` if the declaration `T t(e);` is well-formed, for some invented temporary variable t (8.5). The effect of such an explicit conversion is the same as performing the declaration and initialization and then using the temporary variable as the result of the conversion. The expression e is used as a glvalue if and only if the initialization uses it as a glvalue.

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

6 Any expression can be explicitly converted to type cv void, in which case it becomes a discarded-value expression (Clause 5). [ Note: however, if the value is in a temporary object (12.2), 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 ]

§ 5.2.9
The inverse of any standard conversion sequence (Clause 4) not containing an lvalue-to-rvalue (4.1), array-to-pointer (4.2), function-to-pointer (4.3), null pointer (4.10), null member pointer (4.11), or boolean (4.12) conversion, can be performed explicitly using `static_cast`. A program is ill-formed if it uses `static_cast` to perform the inverse of an ill-formed standard conversion sequence. [Example:

```cpp
struct B { 
};
struct D : private B { 
};
void f() {
    static_cast<D*>(0); // Error: B is a private base of D.
    static_cast<int B::*>(0); // Error: B is a private base of D.
}
```
—end example]

The lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) conversions are applied to the operand. Such a `static_cast` is subject to the restriction that the explicit conversion does not cast away constness (5.2.11), and the following additional rules for specific cases:

A value of a scoped enumeration type (7.2) can be explicitly converted to an integral type. When that type is `cv bool`, the resulting value is `false` if the original value is zero and `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. The value is unchanged if the original value is within the range of the enumeration values (7.2). 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 (4.9), 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 10) from `B`, if a valid standard conversion from “pointer to `D`” to “pointer to `B`” exists (4.10), `cv2` is the same cv-qualification as, or greater cv-qualification than, `cv1`, and `B` is neither a virtual base class of `D` nor a base class of a virtual base class of `D`. The null pointer value (4.10) 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 10) of `D`, if a valid standard conversion from “pointer to member of `B` of type `T`” to “pointer to member of `D` of type `T`” exists (4.11), and `cv2` is the same cv-qualification as, or greater cv-qualification than, `cv1`. The null member pointer value (4.11) is converted to the null member pointer value of the destination type. If the prvalue of type “pointer to `cv1 B`” points to a `B` that actually contains the original member, or is a base or derived class of the class containing the original member, the resulting pointer points to the enclosing object of type `D`. 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 5.5. — 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`. The null pointer value is converted to the null pointer value of the destination type. If the original pointer value represents the address `A` of a byte in memory and `A` satisfies the alignment requirement of `T`, then the resulting pointer value represents the same address as the original pointer value, that is, `A`. The result of any other such

### § 5.2.9

(4.11) Function types (including those used in pointer to member function types) are never cv-qualified; see 8.3.5.
pointer conversion is unspecified. A value of type pointer to object converted to “pointer to cv void” and back, possibly with different cv-qualification, shall have its original value. [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]

5.2.10 Reinterpret cast [expr.reinterpret.cast]

1 The result of the expression `reinterpret_cast<T>(v)` is the result of converting the expression `v` to type `T`. If `T` is an lvalue reference type or an rvalue reference to function type, the result is an lvalue; if `T` is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue and the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the expression `v`. Conversions that can be performed explicitly using `reinterpret_cast` are listed below. No other conversion can be performed explicitly using `reinterpret_cast`.

2 The `reinterpret_cast` operator shall not cast away constness (5.2.11). 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.

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

4 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 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 3.7.4.3, 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. The effect of calling a function through a pointer to a function type (8.3.5) that is not the same as the type used in the definition of the function is undefined. 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 4.10 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*>(static_cast<cv void*>(v))`. 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. 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.

71) The types may have different cv-qualifiers, subject to the overall restriction that a `reinterpret_cast` cannot cast away constness.
The null pointer value (4.10) is converted to the null pointer value of the destination type. [Note: A null pointer constant of type std::nullptr_t cannot be converted to a pointer type, and a null pointer constant of integral type is not necessarily converted to a null pointer value. — end note]

A prvalue of type “pointer to member of X of type T1” can be explicitly converted to a prvalue of a different type “pointer to member of Y of type T2” if T1 and T2 are both function types or both object types. The null member pointer value (4.11) 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.

A glvalue expression of type T1 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 refers to the same object as the source glvalue, but with the specified type. [Note: That is, for lvalues, a reference cast reinterpret_cast<T&>(x) has the same effect as the conversion *reinterpret_cast<T*>(&x) with the built-in & and * operators (and similarly for reinterpret_cast<T&&>(x)). — end note] No temporary is created, no copy is made, and constructors (12.1) or conversion functions (12.3) are not called.

5.2.11 Const cast [expr.const.cast]

The result of the expression const_cast<T>(v) is of type T. If T is an lvalue reference to object type, the result is an lvalue; if T is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue and the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) 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 section, an expression may be cast to its own type using a const_cast operator. — end note]

For two similar types T1 and T2 (4.4), a prvalue of type T1 may be explicitly converted to the type T2 using a const_cast. The result of a const_cast refers to the original entity. [Example:

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

CA &&r = A{};           // 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 T1 and T2, if a pointer to T1 can be explicitly converted to the type “pointer to T2” using a const_cast, then the following conversions can also be made:

(4.1) — an lvalue of type T1 can be explicitly converted to an lvalue of type T2 using the cast const_cast<T2&>;

(4.2) — a glvalue of type T1 can be explicitly converted to an xvalue of type T2 using the cast const_cast<T2&&> and

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

73) This is sometimes referred to as a type pun.
— if \( T_1 \) is a class type, a prvalue of type \( T_1 \) can be explicitly converted to an xvalue of type \( T_2 \) using the cast `\texttt{const\_cast}<T2&&>`.

The result of a reference `\texttt{const\_cast}` refers to the original object.

A null pointer value (4.10) is converted to the null pointer value of the destination type. The null member pointer value (4.11) 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 `\texttt{const\_cast}` that casts away a const-qualifier\(^{74}\) may produce undefined behavior (7.1.6.1). — 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 (4.4) 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 `\texttt{const\_cast}`. For instance, conversions between pointers to functions are not covered because such conversions lead to values whose use causes undefined behavior. For the same reasons, conversions between pointers to member functions, and in particular, the conversion from a pointer to a const member function to a pointer to a non-const member function, are not covered. — end note]

5.3 Unary expressions

Expressions with unary operators group right-to-left.

\[
\text{unary-expression:}
\begin{align*}
\text{postfix-expression} \\
++ \text{cast-expression} \\
-- \text{cast-expression} \\
\text{unary-operator cast-expression} \\
\text{sizeof unary-expression} \\
\text{sizeof ( type-id )} \\
\text{sizeof ... ( identifier )} \\
\text{alignof ( type-id )} \\
\text{noexcept-expression} \\
\text{new-expression} \\
\text{delete-expression}
\end{align*}
\]

\text{unary-operator: one of}

\[
\begin{align*}
* & \\
+ & \\
- & \\
! & \\
~ & \\
\end{align*}
\]

5.3.1 Unary operators

The unary * operator performs \textit{indirection}: the expression to which it is applied shall be a pointer to an object type, or a pointer to a function type and the result is an lvalue referring to the object or function to which the expression points. If the type of the expression is “pointer to \( T \),” the type of the result is “\( T \)” [Note: indirection through a pointer to an incomplete type (other than \texttt{cv void}) is valid. The lvalue thus obtained can be used in limited ways (to initialize a reference, for example); this lvalue must not be converted to a prvalue, see 4.1. — end note]

\(^{74}\) \texttt{const\_cast} is not limited to conversions that cast away a const-qualifier.
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 (1.7) 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 (5.7) and comparison (5.9, 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:

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

—end example] [Note: a pointer to member formed from a mutable non-static data member (7.1.1) 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” (4.3). 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 13) can be taken only in a context that uniquely determines which version of the overloaded function is referred to (see 13.4). [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 4); its value is true if the converted operand is false and false otherwise. The type of the result is bool.

10 The operand of the complement operator ~ shall have integral or unscoped enumeration type; the result is the one’s 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 unary-expression &X(), where X is a class-name or decltype-specifier. The ambiguity is resolved in favor of treating ~ as a unary complement rather than treating ~X as referring to a destructor.

§ 5.3.1
5.3.2 Increment and decrement [expr.pre.incr]

The operand of prefix ++ is modified by adding 1, or set to true if it is bool (this use is deprecated). The operand shall be a modifiable lvalue. The type of the operand shall be an arithmetic type 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. If x is not of type bool, the expression ++x is equivalent to x+=1 [Note: See the discussions of addition (5.7) and assignment operators (5.18) for information on conversions. —end note]

The operand of prefix -- is modified by subtracting 1. The operand shall not be of type bool. 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 5.2.6. —end note]

5.3.3 Sizeof [expr.sizeof]

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 (Clause 5), or a parenthesized type-id. The sizeof operator shall not be applied to an expression that has function or incomplete type, to an enumeration type whose underlying type is not fixed before all its enumerators have been declared, 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 (3.9.1) is implementation-defined. [Note: In particular, sizeof(bool), sizeof(char16_t), sizeof(char32_t), and sizeof(wchar_t) are implementation-defined. —end note] [Note: See 1.7 for the definition of byte and 3.9 for the definition of object representation. —end note]

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 (1.8). The result of applying sizeof to a base class subobject is the size of the base class type.75 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.

The sizeof operator can be applied to a pointer to a function, but shall not be applied directly to a function.

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 (14.5.3). [Example:

```c

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

    — end example]
```

The result of sizeof and sizeof... is a constant of type std::size_t. [Note: std::size_t is defined in the standard header <cstddef> (18.2). —end note]

5.3.4 New [expr.new]

The new-expression attempts to create an object of the type-id (8.1) 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 (1.8, 3.9, 10.4). It is implementation-defined whether over-aligned types

75) sizeof(bool) is not required to be 1.
76) 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.
are supported (3.11). [Note: because references are not objects, references cannot be created by new-expressions. — end note] [Note: the type-id may be a cv-qualified type, in which case the object created by the new-expression has a cv-qualified type. — end note]

new-expression:
::opt new new-placement opt new-type-id new-initializer opt
::opt new new-placement opt (type-id) new-initializer opt

new-placement:
(expression-list)

new-type-id:
type-specifier-seq new-declarator opt

new-declarator:
ptr-operator new-declarator opt
noptr-new-declarator

noptr-new-declarator:
[expression] 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 (3.7.4). [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 new-expression returns a pointer to the object created. If it is an array, the new-expression returns a pointer to the initial element of the array.

2 If a placeholder type (7.1.6.4) appears in the type-specifier-seq of a new-type-id or type-id of a new-expression, the new-expression shall contain a new-initializer of the form

(assignment-expression)

The allocated type is deduced from the new-initializer as follows: Let e be the assignment-expression in the new-initializer 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 (7.1.6.4):

T x(e);

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

§ 5.3.4
Instead, the explicitly parenthesized version of the `new` operator can be used to create objects of compound types (3.9.2):

```c
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 (5.20) 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]

7 The `expression` in a `noptr-new-declarator` is erroneous if:

   7.1 — the `expression` is of non-class type and its value before converting to `std::size_t` is less than zero;

   7.2 — the `expression` is of class type and its value before application of the second standard conversion (13.3.3.1.2) is less than zero;

   7.3 — its value is such that the size of the allocated object would exceed the implementation-defined limit (annex B); or

   7.4 — 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 (2.13.5)) exceeds the number of elements to initialize.

If the `expression`, after converting to `std::size_t`, is a core constant expression and the expression is erroneous, the program is ill-formed. Otherwise, a `new-expression` with an erroneous expression does not call an allocation function and terminates by throwing an exception of a type that would match a handler (15.3) of type `std::bad_array_new_length` (18.6.2.2). When the value of the `expression` is zero, the allocation function is called to allocate an array with no elements.

8 A `new-expression` may obtain storage for the object by calling an `allocation function` (3.7.4.1). If the `new-expression` terminates by throwing an exception, it may release storage by calling a deallocation function (3.7.4.2). If the allocated type is a non-array type, the allocation function’s name is `operator new` and the deallocation function’s name is `operator delete`. If the allocated type is an array type, the allocation function’s name is `operator new[]` and the deallocation function’s name is `operator delete[]`. [Note: an implementation shall provide default definitions for the global allocation functions (3.7.4, 18.6.1.1, 18.6.1.2). A C++ program can provide alternative definitions of these functions (17.6.4.6) and/or class-specific versions (12.5). — end note]

9 If the `new-expression` begins with a unary :: operator, the allocation function’s name is looked up in the global scope. Otherwise, if the allocated type is a class type `T` or array thereof, the allocation function’s name is looked up in the scope of `T`. If this lookup fails to find the name, or if the allocated type is not a class type, the allocation function’s name is looked up in the global scope.

10 An implementation is allowed to omit a call to a replaceable global allocation function (18.6.1.1, 18.6.1.2). When it does so, the storage is instead provided by the implementation or provided by extending the `std::size_t` and thus thwarts any attempt to detect a negative value afterwards.

§ 5.3.4
allocation of another *new-expression*. The implementation may extend the allocation of a *new-expression* \(e_1\) to provide storage for a *new-expression* \(e_2\) if the following would be true were the allocation not extended:

- (10.1) The evaluation of \(e_1\) is sequenced before the evaluation of \(e_2\), and
- (10.2) \(e_2\) is evaluated whenever \(e_1\) obtains storage, and
- (10.3) both \(e_1\) and \(e_2\) invoke the same replaceable global allocation function, and
- (10.4) if the allocation function invoked by \(e_1\) and \(e_2\) is throwing, any exceptions thrown in the evaluation of either \(e_1\) or \(e_2\) would be first caught in the same handler, and
- (10.5) the pointer values produced by \(e_1\) and \(e_2\) are operands to evaluated *delete-expressions*, and
- (10.6) the evaluation of \(e_2\) is sequenced before the evaluation of the *delete-expression* whose operand is the pointer value produced by \(e_1\).

**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]};
    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 —

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` and `unsigned char`, 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 (3.11) of any object type whose size is no greater than the size of the array being created. [Note: Because allocation functions are assumed to return pointers to storage that is appropriately aligned for objects of any type with fundamental alignment, this constraint on array allocation overhead permits the common idiom of allocating character arrays into which objects of other types will later be placed. — end note]

When a *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.

§ 5.3.4
The `new-placement` syntax is used to supply additional arguments to an allocation function. If used, overload resolution is performed on a function call created by assembling an argument list consisting of the amount of space requested (the first argument) and the expressions in the `new-placement` part of the `new-expression` (the second and succeeding arguments). The first of these arguments has type `std::size_t` and the remaining arguments have the corresponding types of the expressions in the `new-placement`; such an expression is called a placement `new-expression`.

[Example:

(14.1) new T results in a call of `operator new(sizeof(T))`,
(14.2) new(2,f) T results in a call of `operator new(sizeof(T),2,f)`,
(14.3) new T[5] results in a call of `operator new[](sizeof(T)*5+x)`, and
(14.4) new(2,f) T[5] results in a call of `operator new[](sizeof(T)*5+y,2,f)`.

Here, x and y are non-negative unspecified values 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 (15.4), it indicates failure to allocate storage by throwing a `std::bad_alloc` exception (3.7.4.1, Clause 15, 18.6.2.1); 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 reserved placement allocation function (18.6.1.3) 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.

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

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

(17.1) If the `new-initializer` is omitted, the object is default-initialized (8.5). [Note: If no initialization is performed, the object has an indeterminate value. —end note]
(17.2) Otherwise, the `new-initializer` is interpreted according to the initialization rules of 8.5 for direct-initialization.

The invocation of the allocation function is indeterminately sequenced with respect to the evaluations of expressions in the `new-initializer`. Initialization of the allocated object is sequenced before the value computation of the `new-expression`. It is unspecified whether expressions in the `new-initializer` are evaluated if the allocation function returns the null pointer or exits using an exception.

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 (12.5), and the constructor (12.1). If the `new-expression` creates an array of objects of class type, the destructor is potentially invoked (12.4).

If any part of the object initialization described above terminates by throwing an exception, storage has been obtained for the object, 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

---

78) This may include evaluating a `new-initializer` and/or calling a constructor.
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]

21 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.

22 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 (8.3.5), 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 the two-parameter form of a usual deallocation function (3.7.4.2) 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 (5.3.5)  

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

23 If a new-expression calls a deallocation function, it passes the value returned from the allocation function call as the first argument of type void*. If a placement deallocation function is called, it is passed the same additional arguments as were passed to the placement allocation function, that is, the same arguments as those specified with the new-placement syntax. If the implementation is allowed to make a copy of any argument as part of the call to the allocation function, it is allowed to make a copy (of the same original value) as part of the call to the deallocation function or to reuse the copy made as part of the call to the allocation function. If the copy is elided in one place, it need not be elided in the other.

5.3.5 Delete  

The delete-expression operator destroys a most derived object (1.8) or array created by a new-expression.

delete-expression:

```
::opt delete cast-expression
::opt delete [ ] cast-expression
```

The first alternative is for non-array objects, and the second is for arrays. Whenever the delete keyword is immediately followed by empty square brackets, it shall be interpreted as the second alternative.\(^{79}\) The operand shall be of pointer to object type or of class type. If of class type, the operand is contextually implicitly converted (Clause 4) to a pointer to object type.\(^{80}\) The delete-expression’s result has type void.

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

\(^{79}\) 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.

\(^{80}\) This implies that an object cannot be deleted using a pointer of type void* because void is not an object type.
this section. In the first alternative (delete object), 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 (1.8) representing a base class of such an object (Clause 10). If not, the behavior is undefined. In the second alternative (delete array), the value of the operand of delete may be a null pointer value or a pointer value that resulted from a previous array new-expression. If not, the behavior is undefined. [Note: this means that the syntax of the delete-expression must match the type of the object allocated by new, not the syntax of the new-expression. —end note] [Note: a pointer to a const type can be the operand of a delete-expression; it is not necessary to cast away the constness (5.2.11) of the pointer expression before it is used as the operand of the delete-expression. —end note]

3 In the first alternative (delete object), 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 the second alternative (delete array) 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 12.6.2).

7 If the value of the operand of the delete-expression is not a null pointer value, then:

(7.1) — If the allocation call for the new-expression for the object to be deleted was not omitted and the allocation was not extended (5.3.4), the delete-expression shall call a deallocation function (3.7.4.2). The value returned from the allocation call of the new-expression shall be passed as the first argument to the deallocation function.

(7.2) — Otherwise, if the allocation was extended or was provided by extending the allocation of another new-expression, and the delete-expression for every other pointer value produced by a new-expression that had storage provided by the extended new-expression has been evaluated, the delete-expression shall call a deallocation function. The value returned from the allocation call of the extended new-expression shall be passed as the first argument to the deallocation function.

(7.3) — Otherwise, the delete-expression will not call a deallocation function (3.7.4.2).

Otherwise, it is unspecified whether the deallocation function will be called. [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]

8 [Note: An implementation provides default definitions of the global deallocation functions operator delete() for non-arrays (18.6.1.1) and operator delete[]() for arrays (18.6.1.2). A C++ program can provide alternative definitions of these functions (17.6.4.6), and/or class-specific versions (12.5). —end note]

9 When the keyword delete in a delete-expression is preceded by the unary :: operator, the deallocation function’s name is looked up in global scope. Otherwise, the lookup considers class-specific deallocation functions (12.5). If no class-specific deallocation function is found, the deallocation function’s name is looked up in global scope.

10 If deallocation function lookup finds both a usual deallocation function with only a pointer parameter and a usual deallocation function with both a pointer parameter and a size parameter, the function to be called is selected as follows:

81) For non-zero-length arrays, this is the same as a pointer to the first element of the array created by that new-expression. Zero-length arrays do not have a first element.
If 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 two parameters is selected.

Otherwise, it is unspecified which of the two deallocation functions is selected.

When a delete-expression is executed, the selected deallocation function shall be called with the address of the block of storage to be reclaimed as its first argument and (if the two-parameter deallocation function is used) the size of the block as its second argument.

Access and ambiguity control are done for both the deallocation function and the destructor (12.4, 12.5).

5.3.6 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.

5.3.7 noexcept operator

The noexcept operator determines whether the evaluation of its operand, which is an unevaluated operand (Clause 5), can throw an exception (15.1).

The result of the noexcept operator is a constant of type bool and is a prvalue.

The result of the noexcept operator is true if the set of potential exceptions of the expression (15.4) is empty, and false otherwise.

5.4 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 Clause 5. — end note]

An explicit type conversion can be expressed using functional notation (5.2.3), a type conversion operator (dynamic_cast, static_cast, reinterpret_cast, const_cast), or the cast notation.

Any type conversion not mentioned below and not explicitly defined by the user (12.3) is ill-formed.

The conversions performed by

(4.1) — a const_cast (5.2.11),
(4.2) — a static_cast (5.2.9),
(4.3) — a static_cast followed by a const_cast,
(4.4) — a reinterpret_cast (5.2.10), or

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.
(4.5) — a reinterpret_cast followed by a const_cast,

   can be performed using the cast notation of explicit type conversion. The same semantic restrictions and behaviors apply, with the exception that in performing a static_cast in the following situations the conversion is valid even if the base class is inaccessible:

(4.6) — a pointer to an object of derived class type or an lvalue or rvalue of derived class type may be explicitly converted to a pointer or reference to an unambiguous base class type, respectively;

(4.7) — a pointer to member of derived class type may be explicitly converted to a pointer to member of an unambiguous non-virtual base class type;

(4.8) — a pointer to an object of an unambiguous non-virtual base class type, a glvalue of an unambiguous non-virtual base class type, or a pointer to member of an unambiguous non-virtual base class type may be explicitly converted to a pointer, a reference, or a pointer to member of a derived class type, respectively.

If a conversion can be interpreted in more than one of the ways listed above, the interpretation that appears first in the list is used, even if a cast resulting from that interpretation is ill-formed. If a conversion can be interpreted in more than one way as a static_cast followed by a const_cast, the conversion is ill-formed.

[ Example:
  
  struct A { };
  struct I1 : A { };
  struct I2 : A { };
  struct D : I1, I2 { };
  A* foo( D* p ) {
    return (A*)( p ); // ill-formed static_cast interpretation
  }

  —end example]

5 The operand of a cast using the cast notation can be a prvalue of type “pointer to incomplete class type”. The destination type of a cast using the cast notation can be “pointer to incomplete class type”. If both the operand and destination types are class types and one or both are incomplete, it is unspecified whether the static_cast or the reinterpret_cast interpretation is used, even if there is an inheritance relationship between the two classes. [Note: For example, if the classes were defined later in the translation unit, a multi-pass compiler would be permitted to interpret a cast between pointers to the classes as if the class types were complete at the point of the cast. —end note]

5.5 Pointer-to-member operators [expr.mptr.oper]

1 The pointer-to-member operators ->* and .* group left-to-right.

   pm-expression:
   
   cast-expression
   pm-expression .* cast-expression
   pm-expression ->* cast-expression

2 The binary operator .* binds its second operand, which shall be of type “pointer to member of T” to its first operand, which shall be of class T or of a class of which T is an unambiguous and accessible base class. The result is an object or a function of the type specified by the second operand.

3 The binary operator ->* binds its second operand, which shall be of type “pointer to member of T” to its first operand, which shall be of type “pointer to T” or “pointer to a class of which T is an unambiguous and accessible base class.” The expression E1->*E2 is converted into the equivalent form (*E1).*E2. 

4 Abbreviating pm-expression.*cast-expression as E1.*E2, E1 is called the object expression. If the dynamic type of E1 does not contain the member to which E2 refers, the behavior is undefined.

§ 5.5
The restrictions on `cv`-qualification, and the manner in which the `cv`-qualifiers of the operands are combined to produce the `cv`-qualifiers of the result, are the same as the rules for `E1`:`E2` given in 5.2.5. [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 `.*` or `->*` 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 `.*` expression whose object expression is an rvalue, the program is ill-formed if the second operand is a pointer to member function with `ref-qualifier` &. In a `.*` expression whose object expression is an lvalue, the program is ill-formed if the second operand is a pointer to member function with `ref-qualifier` &&. The result of a `.*` expression whose second operand is a pointer to a data member is an lvalue if the first operand is an lvalue and an xvalue otherwise. The result of a `.*` expression whose second operand is a pointer to a member function is a prvalue. If the second operand is the null pointer to member value (4.11), the behavior is undefined.

### 5.6 Multiplicative operators

The multiplicative operators `*`, `/`, and `%` group left-to-right.

```
multiplicative-expression:
  pm-expression
  pm-expression * pm-expression
  pm-expression / pm-expression
  pm-expression % pm-expression
```

The operands of `*` and `/` shall have arithmetic or unscoped enumeration type; the operands of `%` shall have integral or unscoped enumeration type. The usual arithmetic conversions are performed on the operands and determine the type of the result.

The binary `*` operator indicates multiplication.

The binary `/` operator yields the quotient, and the binary `%` operator yields the remainder from the division of the first expression by the second. If the second operand of `/` or `%` is zero the behavior is undefined. For integral operands the `/` operator yields the algebraic quotient with any fractional part discarded;\(^{83}\) if the quotient `a/b` is representable in the type of the result, `(a/b)*b + a%b` is equal to `a`; otherwise, the behavior of both `a/b` and `a%b` is undefined.

### 5.7 Additive operators

The additive operators `+` and `-` group left-to-right. The usual arithmetic conversions are performed for operands of arithmetic or enumeration type.

\(^{83}\) 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 pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integral expression. In other words, if the expression P points to the i-th element of an array object, the expressions (P)+N (equivalently, N+(P)) and (P)-N (where N has the value n) point to, respectively, the i + n-th and i − n-th elements of the array object, provided they exist. Moreover, if the expression P points to the last element of an array object, the expression (P)+1 points one past the last element of the array object, and if the expression Q points one past the last element of an array object, the expression (Q)-1 points to the last element of the array object. If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined.

5 When two pointers to elements of the same array object are subtracted, the result is the difference of the subscripts of the two array elements. The type of the result is an implementation-defined signed integral type; this type shall be the same type that is defined as std::ptrdiff_t in the <cstddef> header (18.2). As with any other arithmetic overflow, if the result does not fit in the space provided, the behavior is undefined. In other words, if the expressions P and Q point to, respectively, the i-th and j-th elements of an array object, the expression (P)-(Q) has the value i − j provided the value fits in an object of type std::ptrdiff_t. Moreover, if the expression P points either to an element of an array object or one past the last element of an array object, and the expression Q points to the last element of the same array object, the expression ((Q)+1)-(P) has the same value as ((Q)-(P))+1 and as −((P)-(((Q)+1))), and has the value zero if the expression P points one past the last element of the array object, even though the expression (Q)+1 does not point to an element of the array object. Unless both pointers point to elements of the same array object, or one past the last element of the array object, the behavior is undefined. 85

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 (4.4), 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]  

84) An object that is not an array element is considered to belong to a single-element array for this purpose; see 5.3.1.
85) Another way to approach pointer arithmetic is first to convert the pointer(s) to character pointer(s): In this scheme the integral value of the expression added to or subtracted from the converted pointer is first multiplied by the size of the object originally pointed to, and the resulting pointer is converted back to the original type. For pointer subtraction, the result of the difference between the character pointers is similarly divided by the size of the object originally pointed to.

When viewed in this way, an implementation need only provide one extra byte (which might overlap another object in the program) just after the end of the object in order to satisfy the “one past the last element” requirements.
7 If the value 0 is added to or subtracted from a pointer value, the result compares equal to the original pointer value. If two pointers point to the same object or both point one past the end of the same array or both are null, and the two pointers are subtracted, the result compares equal to the value 0 converted to the type `std::ptrdiff_t`.

5.8 Shift operators

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 × 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 × 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.

5.9 Relational operators

1 The relational operators group left-to-right. [Example: `a<b<c` means `(a<b)<c` and not `(a<b)&&(b<c)`.

```
relational-expression:
  shift-expression
  relational-expression < shift-expression
  relational-expression > shift-expression
  relational-expression <= shift-expression
  relational-expression >= shift-expression
```

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`.

2 The usual arithmetic conversions are performed on operands of arithmetic or enumeration type. If both operands are pointers, pointer conversions (4.10) and qualification conversions (4.4) are performed to bring them to their composite pointer type (Clause 5). After conversions, the operands shall have the same type.

3 Comparing pointers to objects\(^\text{86}\) 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 one pointer points to an element of an array, or to a subobject thereof, and another pointer points one past the last element of the array, the latter pointer compares greater.

3. 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 11) and provided their class is not a union.

\(^{86}\) An object that is not an array element is considered to belong to a single-element array for this purpose; see 5.3.1.
If two operands \( p \) and \( q \) compare equal (5.10), \( p \leq q \) and \( p \geq q \) both yield \textit{true} and \( p < q \) and \( p > q \) both yield \textit{false}. Otherwise, if a pointer \( p \) compares greater than a pointer \( q \), \( p \geq q \), \( p > q \), \( q \leq p \), and \( q < p \) all yield \textit{true} and \( p < q \), \( p < q \), \( q > p \), and \( q > p \) all yield \textit{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 \textit{true} if the specified relationship is true and \textit{false} if it is false.

### 5.10 Equality operators

[expr.eq]

\[
\begin{align*}
\text{equality-expression:} & \\
\text{relational-expression} & \\
\text{equality-expression} & \text{== relational-expression} \\
\text{equality-expression} & \text{!= relational-expression}
\end{align*}
\]

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 \texttt{std::nullptr_t}. The operators \( == \) and \( != \) both yield \textit{true} or \textit{false}, i.e., a result of type \texttt{bool}. In each case below, the operands shall have the same type after the specified conversions have been applied.

2. If at least one of the operands is a pointer, pointer conversions (4.10) and qualification conversions (4.4) are performed on both operands to bring them to their composite pointer type (Clause 5). Comparing pointers is defined as follows: Two pointers compare equal if they are both null, both point to the same function, or both represent the same address (3.9.2), otherwise they compare unequal.

3. If at least one of the operands is a pointer to member, pointer to member conversions (4.11) and qualification conversions (4.4) are performed on both operands to bring them to their composite pointer type (Clause 5). Comparing pointers to members is defined as follows:

\begin{enumerate}
\item If two pointers to members are both the null member pointer value, they compare equal.
\item If only one of two pointers to members is the null member pointer value, they compare unequal.
\item If either is a pointer to a virtual member function, the result is unspecified.
\item If one refers to a member of class \( C_1 \) and the other refers to a member of a different class \( C_2 \), where neither is a base class of the other, the result is unspecified. \[ \text{Example:} \]
\begin{verbatim}
struct A {}; 
struct B : A { int x; }; 
struct C : A { int x; }; 

int A::*bx = (int(A::*))&B::x; 
int A::*cx = (int(A::*))&C::x; 

bool b1 = (bx == cx); // unspecified
\end{verbatim}
\[ \text{— end example} \]
\item If both refer to (possibly different) members of the same union (9.5), they compare equal.
\item Otherwise, two pointers to members compare equal if they would refer to the same member of the same most derived object (1.8) or the same subobject if indirection with a hypothetical object of the associated class type were performed, otherwise they compare unequal. \[ \text{Example:} \]
\begin{verbatim}
struct B { 
   int f(); 
}; 
struct L : B { }; 
struct R : B { }; 
\end{verbatim}
\end{enumerate}
struct D : L, R { }

int (B::*pb)() = &B::f;
int (L::*pl)() = pb;
int (R::*pr)() = pb;
int (D::*pdl)() = pl;  // false
int (D::*pdr)() = pr;  // true

— end example —

Two operands of type `std::nullptr_t` or one operand of type `std::nullptr_t` and the other a null pointer constant compare equal.

If two operands compare equal, the result is `true` for the `==` operator and `false` for the `!=` operator. If two operands compare unequal, the result is `false` for the `==` operator and `true` for the `!=` operator. Otherwise, the result of each of the operators is unspecified.

If both operands are of arithmetic or enumeration type, the usual arithmetic conversions are performed on both operands; each of the operators shall yield `true` if the specified relationship is true and `false` if it is false.

### 5.11 Bitwise AND operator [expr.bit.and]

```c
and-expression:
  equality-expression
  and-expression & equality-expression
```

The usual arithmetic conversions are performed; the result is the bitwise AND function of the operands. The operator applies only to integral or unscoped enumeration operands.

### 5.12 Bitwise exclusive OR operator [expr.xor]

```c
exclusive-or-expression:
  and-expression
  exclusive-or-expression ^ and-expression
```

The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral or unscoped enumeration operands.

### 5.13 Bitwise inclusive OR operator [expr.or]

```c
inclusive-or-expression:
  exclusive-or-expression
  inclusive-or-expression | exclusive-or-expression
```

The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral or unscoped enumeration operands.

### 5.14 Logical AND operator [expr.log.and]

```c
logical-and-expression:
  inclusive-or-expression
  logical-and-expression && inclusive-or-expression
```

The `&&` operator groups left-to-right. The operands are both contextually converted to `bool` (Clause 4). The result is `true` if both operands are `true` and `false` otherwise. Unlike `&`, `&&` guarantees left-to-right evaluation: the second operand is not evaluated if the first operand is `false`.

The result is a `bool`. If the second expression is evaluated, every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second expression.

§ 5.14
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 4). It returns 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.

5.16 Conditional operator

conditional-expression:
  logical-or-expression
  logical-or-expression ? expression : assignment-expression

1 Conditional expressions group right-to-left. The first expression is contextually converted to bool (Clause 4). 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 (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.

(2.2) — Both the second and the third operands have type void; the result is of type void and is a prvalue.

[Note: This includes the case where both operands are throw-expressions. — end note]

3 Otherwise, if the second and third operand 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 convert each of those operands to the type of the other. The process for determining whether an operand expression E1 of type T1 can be converted to match an operand expression E2 of type T2 is defined as follows:

(3.1) — If E2 is an lvalue: E1 can be converted to match E2 if E1 can be implicitly converted (Clause 4) to the type “value reference to T2”, subject to the constraint that in the conversion the reference must bind directly (8.5.3) to an lvalue.

(3.2) — If E2 is an xvalue: E1 can be converted to match E2 if E1 can be implicitly converted to the type “rvalue reference to T2”, subject to the constraint that the reference must bind directly.

(3.3) — If E2 is a prvalue or if neither of the conversions above can be done and at least one of the operands has (possibly cv-qualified) class type:

(3.3.1) — if E1 and E2 have class type, and the underlying class types are the same or one is a base class of the other: E1 can be converted to match E2 if the class of T2 is the same type as, or a base class of, the class of T1, and the cv-qualification of T2 is the same cv-qualification as, or a greater cv-qualification than, the cv-qualification of T1. If the conversion is applied, E1 is changed to a prvalue of type T2 by copy-initializing a temporary of type T2 from E1 and using that temporary as the converted operand.

(3.3.2) — Otherwise (if E1 or E2 has a non-class type, or if they both have class types but the underlying classes are not the same and neither is a base class of the other): E1 can be converted to match
E2 if E1 can be implicitly converted to the type that E2 would have after applying the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions.

Using this process, it is determined whether the second operand can be converted to match the third operand, and whether the third operand can be converted to match the second operand. If both can be converted, or one can be converted but the conversion is ambiguous, the program is ill-formed. If neither can be converted, the operands are left unchanged and further checking is performed as described below. If exactly one conversion is possible, 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 section.

4 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.

5 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 (13.3.1.2, 13.6). 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 section.

6 Lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the second and third operands. After those conversions, one of the following shall hold:

(6.1) The second and third operands have the same type; the result is of that type. If the operands have class type, the result is a prvalue temporary of the result type, which is copy-initialized from either the second operand or the third operand depending on the value of the first operand.

(6.2) The second and third operands have arithmetic or enumeration type; the usual arithmetic conversions are performed to bring them to a common type, and the result is of that type.

(6.3) One or both of the second and third operands have pointer type; pointer conversions (4.10) and qualification conversions (4.4) are performed to bring them to their composite pointer type (Clause 5). The result is of the composite pointer type.

(6.4) One or both of the second and third operands have pointer to member type; pointer to member conversions (4.11) and qualification conversions (4.4) are performed to bring them to their composite pointer type (Clause 5). The result is of the composite pointer type.

(6.5) Both the second and third operands have type std::nullptr_t or one has that type and the other is a null pointer constant. The result is of type std::nullptr_t.

5.17 Throwing an exception [expr.throw]

A throw-expression is of type void.

1 A throw-expression with an operand throws an exception (15.1); 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 returning T” to “pointer to T” or “pointer to function returning T,” respectively.

2 Evaluating a throw-expression with no operand rethrows the currently handled exception (15.3). 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:
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() (15.5.1).

5.18 Assignment and compound assignment operators [expr.ass]

The assignment operator (=) and the compound assignment operators all group right-to-left. All require a
modifiable lvalue as their left operand and return 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.
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
  = *= /= %= += -= >>= <<= &= ˆ= |=

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 4) 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 (12.8, 13.5.3).

5 [ Note: For class objects, assignment is not in general the same as initialization (8.5, 12.1, 12.6, 12.8). — 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 accessed from 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 3.10. — 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{}.
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 (13.5.3, 13.3).

[Example:

```cpp
complex<double> z;
z = { 1, 2 }; // meaning z.operator={1,2}
z += { 1, 2 }; // meaning z.operator++={1,2}
int a, b;
a = b = { 1 }; // meaning a=b=1;
a = { 1 } = b; // syntax error
```
—end example]

5.19 Comma operator
[expr.comma]

The comma operator groups left-to-right.

expression:

```
assignment-expression
expression , assignment-expression
```

A pair of expressions separated by a comma is evaluated left-to-right; the left expression is a discarded-value expression (Clause 5). 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 glvalue and a bit-field. If the value of the right operand is a temporary (12.2), the result is that temporary.

In contexts where comma is given a special meaning, [Example: in lists of arguments to functions (5.2.2) and lists of initializers (8.5) — end example] the comma operator as described in Clause 5 can appear only in parentheses. [Example:

```cpp
f(a, (t=3, t+2), c);
```
has three arguments, the second of which has the value 5. —end example]

5.20 Constant expressions
[expr.const]

Certain contexts require expressions that satisfy additional requirements as detailed in this sub-clause; other contexts have different semantics depending on whether or not an expression satisfies these requirements. Expressions that satisfy these requirements are called constant expressions. [Note: Constant expressions can be evaluated during translation. —end note]

```
constant-expression:
conditional-expression
```

A conditional-expression e is a core constant expression unless the evaluation of e, following the rules of the abstract machine (1.9), would evaluate one of the following expressions:

(2.1) — this (5.1.1), except in a constexpr function or a constexpr constructor that is being evaluated as part of e;

(2.2) — 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 (12.4) [Note: Overload resolution (13.3) is applied as usual — end note];

(2.3) — an invocation of an undefined constexpr function or an undefined constexpr constructor;

87) However, an invocation of an overloaded comma operator is an ordinary function call; hence, the evaluations of its argument expressions are unsequenced relative to one another (see 1.9).
— an expression that would exceed the implementation-defined limits (see Annex B);

— an operation that would have undefined behavior [Note: including, for example, signed integer overflow (Clause 5), certain pointer arithmetic (5.7), division by zero (5.6), or certain shift operations (5.8) — end note];

— a lambda-expression (5.1.2);

— an lvalue-to-rvalue conversion (4.1) unless it is applied to
  — 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
  — a non-volatile glvalue that refers to a subobject of a string literal (2.13.5), or
  — a non-volatile glvalue that refers to a non-volatile object defined with constexpr, or that refers to a non-mutable sub-object of such an object, or
  — a non-volatile glvalue of literal type that refers to a non-volatile object whose lifetime began within the evaluation of e;

— an lvalue-to-rvalue conversion (4.1) or modification (5.18, 5.2.6, 5.3.2) that is applied to a glvalue that refers to a non-active member of a union or a subobject thereof;

— 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
  — it is a non-static data member of an object whose 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 (3.2, 5.1.2);

— a conversion from type cv void * to a pointer-to-object type;

— a dynamic cast (5.2.7);

— a reinterpret_cast (5.2.10);

— a pseudo-destructor call (5.2.4);

— modification of an object (5.18, 5.2.6, 5.3.2) 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 (5.2.8) whose operand is a glvalue of a polymorphic class type;

— a new-expression (5.3.4);

— a delete-expression (5.3.5);

— a relational (5.9) or equality (5.10) operator where the result is unspecified; or

— a throw-expression (5.17).

[Example:
int x; "not constant"
struct A {
    constexpr A(bool b) : m(b?42:x) {} // m with the value 42
    int m;
};
constexpr int v = A(true).m; // OK: constructor call initializes
constexpr int w = A(false).m; // error: initializer for m is x, which is non-constant

constexpr int f1(int k) {
    constexpr int x = k; // error: x is not initialized by a constant expression because lifetime of k began outside the initializer of x
    return x;
}
constexpr int f2(int k) {
    int x = k; // OK: not required to be a constant expression because x is not constexpr
    return x;
}
constexpr int incr(int &n) {
    return ++n;
}
constexpr int g(int k) {
    constexpr int x = incr(k); // error: incr(k) is not a core constant expression because lifetime of k began outside the expression incr(k)
    return x;
}
constexpr int h(int k) {
    int x = incr(k); // OK: incr(k) is not required to be a core constant expression
    return x;
}
constexpr int y = h(1); // OK: initializes y with the value 2 because 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 array bounds (8.3.4, 5.3.4), as bit-field lengths (9.6), as enumerator initializers if the underlying type is not fixed (7.2), and as alignments (7.6.2). — end note]

4 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

(4.1) — user-defined conversions,
(4.2) — lvalue-to-rvalue conversions (4.1),
(4.3) — array-to-pointer conversions (4.2),
(4.4) — function-to-pointer conversions (4.3),

§ 5.20
5 A **constant expression** is either a glvalue core constant expression whose value refers to an entity that is a permitted result of a constant expression (as defined below), or a prvalue core constant expression whose value is an object where, for that object and its subobjects:

- each non-static data member of reference type refers to an entity that is a permitted result of a constant expression, and
- if the object or subobject is of pointer type, it contains the address of an object with static storage duration, the address past the end of such an object (5.7), the address of a function, or a null pointer 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.

6 [Note: Since this International Standard imposes no restrictions on the accuracy of floating-point operations, it is unspecified whether the evaluation of a floating-point expression during translation yields the same result as the evaluation of the same expression (or the same operations on the same values) during program execution.](#) **Example:**

```cpp
bool f() {
    char array[1 + int(1 + 0.2 - 0.1 - 0.1)]; // Must be evaluated during translation
    int size = 1 + int(1 + 0.2 - 0.1 - 0.1); // May be evaluated at runtime
    return sizeof(array) == size;
}
```

It is unspecified whether the value of `f()` will be `true` or `false`. — end example] — end note]

7 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 4) 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 43; }
private:
    int val;
};
template<int> struct X { };    
constexpr A a = 42;
X<a> x; // OK: unique conversion to int
int ary[a]; // error: ambiguous conversion
```

88) Nonetheless, implementations are encouraged to provide consistent results, irrespective of whether the evaluation was performed during translation and/or during program execution.
— end example]
6 Statements

1 Except as indicated, statements are executed in sequence.

\[
\text{statement:}
\]
\[
\begin{align*}
&\text{labeled-statement} \\
&\quad \text{attribute-specifier-seqopt expression-statement} \\
&\quad \text{attribute-specifier-seqopt compound-statement} \\
&\quad \text{attribute-specifier-seqopt selection-statement} \\
&\quad \text{attribute-specifier-seqopt iteration-statement} \\
&\quad \text{attribute-specifier-seqopt jump-statement} \\
&\quad \text{declaration-statement} \\
&\quad \text{attribute-specifier-seqopt try-block}
\end{align*}
\]

The optional attribute-specifier-seq appertains to the respective statement.

6.1 Labeled statement

1 A statement can be labeled.

\[
\text{labeled-statement:}
\]
\[
\begin{align*}
&\quad \text{attribute-specifier-seqopt identifier : statement} \\
&\quad \text{attribute-specifier-seqopt case constant-expression : statement} \\
&\quad \text{attribute-specifier-seqopt default : statement}
\end{align*}
\]

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 definition. Labels have their own name space and do not interfere with other identifiers.

2 Case labels and default labels shall occur only in switch statements.

6.2 Expression statement

1 Expression statements have the form

\[
\text{expression-statement:}
\]
\[
\begin{align*}
&\quad \text{expressionopt ;}
\end{align*}
\]

The expression is a discarded-value expression (Clause 5). 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 (6.5.1). — end note]

6.3 Compound statement or block

1 So that several statements can be used where one is expected, the compound statement (also, and equivalently, called “block”) is provided.

\[
\text{compound-statement:}
\]
\[
\begin{align*}
&\quad \{ \text{statement-seqopt}\} \\
&\quad \text{statement-seq:} \\
&\quad \text{statement} \\
&\quad \text{statement-seq statement}
\end{align*}
\]

A compound statement defines a block scope (3.3). [Note: A declaration is a statement (6.7). — end note]
### 6.4 Selection statements

Selection statements choose one of several flows of control.

```plaintext
selection-statement:
    if ( condition ) statement
    if ( condition ) statement else statement
    switch ( condition ) statement

condition:
    expression
    attribute-specifier-seqopt decl-specifier-seq declarator = initializer-clause
    attribute-specifier-seqopt decl-specifier-seq declarator braced-init-list
```

See 8.3 for the optional `attribute-specifier-seq` in a condition. In Clause 6, 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 (3.3). 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:

```plaintext
if (x)
    int i;
```

can be equivalently rewritten as

```plaintext
if (x) {
    int i;
}
```

Thus after the `if` statement, `i` is no longer in scope. — end example]

2 The rules for conditions apply both to `selection-statements` and to the `for` and `while` statements (6.5). 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 7.1.6.4.

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 re-declared in the outermost block of a substatement controlled by the condition, the declaration that re-declares 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 4). 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.

§ 6.4
In the `decl-specifier-seq` of a `condition`, each `decl-specifier` shall be either a `type-specifier` or `constexpr`.

### 6.4.1 The `if` statement

If the condition (6.4) 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."^89"

### 6.4.2 The `switch` statement

The `switch` statement causes control to be transferred to one of several statements depending on the value of a condition.

The condition shall be of integral type, enumeration type, or class type. If of class type, the condition is contextually implicitly converted (Clause 4) to an integral or enumeration type. If the (possibly converted) type is subject to integral promotions (4.5), 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:

```plaintext
  case constant-expression :
```

where the `constant-expression` shall be a converted constant expression (5.20) 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.

There shall be at most one label of the form

```plaintext
  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`, 6.6.1. [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]

### 6.5 Iteration statements

Iteration statements specify looping.

```plaintext
  iteration-statement:
    while ( condition ) statement
    do statement while ( expression ) ;
    for ( for-init-statement condition_opt; expression_opt) statement
    for ( for-range-declaration : for-range-initializer) statement
```

89) In other words, the `else` is associated with the nearest un-elsed `if`. 
for-range-declaration:
  attribute-specifier-seq_opt decl-specifier-seq declarator

for-range-initializer:
  expression
  braced-init-list
See 8.3 for the optional attribute-specifier-seq in a for-range-declaration. [Note: A for-init-statement ends with a semicolon. — end note]

The substatement in an iteration-statement implicitly defines a block scope (3.3) 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]

[Note: The requirements on conditions in iteration statements are described in 6.4. — end note]

6.5.1 The while statement [stmt.while]

1 In the while statement the substatement is executed repeatedly until the value of the condition (6.4) 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 (3.3.2) 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;
}
```

§ 6.5.1 140
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 |

6.5.2 The do statement

1 The expression is contextually converted to bool (Clause 4); 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.

6.5.3 The for statement

1 The for statement

   for ( for-init-statement condition\_opt; expression\_opt) statement

is equivalent to

   {
   for-init-statement
   while ( condition ) {
   statement
   expression ;
   }
   }

except that names declared in the for-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 (6.4) specifies a test, made before each iteration, such that the loop is exited when the condition becomes false; the expression often specifies incrementing that is done 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 for-init-statement is a declaration, the scope of the name(s) declared extends to the end of the for statement. [ Example:

   int i = 42;
   int a[10];

   for ( int i = 0; i < 10; i++ )
   a[i] = i;

   int j = i;   // j = 42

   — end example ]

6.5.4 The range-based for statement

1 For a range-based for statement of the form

   for ( for-range-declaration : expression ) statement

let range-init be equivalent to the expression surrounded by parentheses\(^ {90} \)

   ( expression )

\(^ {90} \) this ensures that a top-level comma operator cannot be reinterpreted as a delimiter between init-declarators in the declaration of __range.

§ 6.5.4
and for a range-based for statement of the form

```c
for ( for-range-declaration : braced-init-list ) statement
```

let range-init be equivalent to the braced-init-list. In each case, a range-based for statement is equivalent to

```c
{ 
  auto && __range = range-init;
  for ( auto __begin = begin-expr, 
       __end = end-expr;
       __begin != __end;
       ++__begin ) {
    for-range-declaration = *__begin;
    statement
  }
}
```

where __range, __begin, and __end are variables defined for exposition only, and _RangeT is the type of the expression, and begin-expr and end-expr are determined as follows:

1. If _RangeT is an array type, begin-expr and end-expr are __range and __range + __bound, respectively, where __bound is the array bound. If _RangeT is an array of unknown size or an array of incomplete type, the program is ill-formed;

2. If _RangeT is a class type, the unqualified-ids begin and end are looked up in the scope of class _RangeT as if by class member access lookup (3.4.5), and if either (or both) finds at least one declaration, begin-expr and end-expr are __range.begin() and __range.end(), respectively;

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 (3.4.2). [ Note: Ordinary unqualified lookup (3.4.1) 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.

### 6.6 Jump statements

Jump statements unconditionally transfer control.

```c
jump-statement:
  break ;
  continue ;
  return expression_opt ;
  return braced-init-list ;
  goto identifier ;
```

2 On exit from a scope (however accomplished), objects with automatic storage duration (3.7.3) that have been constructed in that scope are destroyed in the reverse order of their construction. [ Note: For temporaries, see 12.2. — 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 6.7 for transfers into blocks).
Note: However, the program can be terminated (by calling `std::exit()` or `std::abort()` (18.5), for example) without destroying class objects with automatic storage duration. — end note

6.6.1 The break statement

The `break` statement shall occur only in an `iteration-statement` or a `switch` statement and causes termination of the smallest enclosing `iteration-statement` or `switch` statement; control passes to the statement following the terminated statement, if any.

6.6.2 The continue statement

The `continue` statement shall occur only in an `iteration-statement` and causes control to pass to the loop-continuation portion of the smallest enclosing `iteration-statement`, that is, to the end of the loop. More precisely, in each of the statements

```c
while (foo) {
    do {
        for (;;) {
            // ...
        }
        // ...
    }
}
contin: ;
```

a `continue` not contained in an enclosed iteration statement is equivalent to `goto contin`.

6.6.3 The return statement

A function returns to its caller by the `return` statement.

The `expression` 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 (12.1), or a destructor (12.4). 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 object or reference to be returned by copy-initialization (8.5) from the operand. [Note: A return statement can involve the construction and copy or move of a temporary object (12.2). A copy or move operation associated with a return statement may be elided or considered as an rvalue for the purpose of overload resolution in selecting a constructor (12.8). — end note] [Example:

```c
std::pair<std::string,int> f(const char* p, int x) {
    return {p,x};
}
```

— end example] Flowing off the end of a function is equivalent to a `return` with no value; this results in undefined behavior in a value-returning function.

The copy-initialization of the returned entity 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 (6.6) of the block enclosing the return statement.

6.6.4 The goto statement

The `goto` statement unconditionally transfers control to the statement labeled by the identifier. The identifier shall be a label (6.1) located in the current function.

6.7 Declaration statement

A declaration statement introduces one or more new identifiers into a block; it has the form

```c
declaration-statement:
    block-declaration
```
If an identifier introduced by a declaration was previously declared in an outer block, the outer declaration is hidden for the remainder of the block, after which it resumes its force.

Variables with automatic storage duration (3.7.3) 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 (6.6).

It is possible to transfer into a block, but not in a way that bypasses declarations with initialization. A program that jumps\(^1\) 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 (8.5). [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]

The zero-initialization (8.5) of all block-scope variables with static storage duration (3.7.1) or thread storage duration (3.7.2) is performed before any other initialization takes place. Constant initialization (3.6.2) of a block-scope entity with static storage duration, if applicable, is performed before its block is first entered. An implementation is permitted to perform early initialization of other block-scope variables with static or thread storage duration under the same conditions that an implementation is permitted to statically initialize a variable with static or thread storage duration in namespace scope (3.6.2). Otherwise such a variable is initialized 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.\(^2\) 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: 3.6.3 describes the order in which block-scope objects with static and thread storage duration are destroyed. —end note]

\(^1\) The transfer from the condition of a switch statement to a case label is considered a jump in this respect.  
\(^2\) The implementation must not introduce any deadlock around execution of the initializer.
6.8 Ambiguity resolution

There is an ambiguity in the grammar involving expression-statements and declarations: An expression-statement with a function-style explicit type conversion (5.2.3) 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 (7.1.6),]

T(a)->m = 7; // expression-statement
T(a)++; // expression-statement
T(a,5)<<c; // expression-statement

T(*d)(int); // declaration
T(e)[5]; // declaration
T(f) = { 1, 2 }; // declaration
T(*g)(double(3)); // declaration

In the last example above, g, which is a pointer to T, is initialized to double(3). This is of course ill-formed for semantic reasons, but that does not affect the syntactic analysis. —end example]

The remaining cases are declarations. [Example:

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:

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() {

§ 6.8

145
```
// 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 ]
```
7 Declarations

1 Declarations generally specify how names are to be interpreted. Declarations have the form

\[
\text{declaration-seq:} \\
\quad \text{declaration} \\
\quad \text{declaration-seq declaration}
\]

\[
\text{declaration:} \\
\quad \text{block-declaration} \\
\quad \text{function-definition} \\
\quad \text{template-declaration} \\
\quad \text{explicit-instantiation} \\
\quad \text{explicit-specialization} \\
\quad \text{linkage-specification} \\
\quad \text{namespace-definition} \\
\quad \text{empty-declaration} \\
\quad \text{attribute-declaration}
\]

\[
\text{block-declaration:} \\
\quad \text{simple-declaration} \\
\quad \text{asm-definition} \\
\quad \text{namespace-alias-definition} \\
\quad \text{using-declaration} \\
\quad \text{using-directive} \\
\quad \text{static_assert-declaration} \\
\quad \text{alias-declaration} \\
\quad \text{opaque-enum-declaration}
\]

\[
\text{alias-declaration:} \\
\quad \text{using} \ \text{identifier attribute-specifier-seq}_{\text{opt}}= \text{type-id} ;
\]

\[
\text{simple-declaration:} \\
\quad \text{decl-specifier-seq}_{\text{opt}} \ \text{init-declarator-list}_{\text{opt}} ; \\
\quad \text{attribute-specifier-seq} \ \text{decl-specifier-seq}_{\text{opt}} \ \text{init-declarator-list} ;
\]

\[
\text{static_assert-declaration:} \\
\quad \text{static_assert} \ ( \text{constant-expression} \) ; \\
\quad \text{static_assert} \ ( \text{constant-expression} , \text{string-literal} \) ;
\]

\[
\text{empty-declaration:} \\
\quad ;
\]

\[
\text{attribute-declaration:} \\
\quad \text{attribute-specifier-seq} ;
\]

[Note: \text{asm-definitions} are described in 7.4, and \text{linkage-specifications} are described in 7.5. \text{Function-definitions} are described in 8.4 and \text{template-declarations} are described in Clause 14. \text{Namespace-definitions} are described in 7.3.1, \text{using-declarations} are described in 7.3.3 and \text{using-directives} are described in 7.3.4. — end note]

2 The \text{simple-declaration}

\[
\text{attribute-specifier-seq}_{\text{opt}} \ \text{decl-specifier-seq}_{\text{opt}} \ \text{init-declarator-list}_{\text{opt}} ;
\]

is divided into three parts. Attributes are described in 7.6. \text{decl-specifiers}, the principal components of a \text{decl-specifier-seq}, are described in 7.1. \text{declarators}, the components of an \text{init-declarator-list}, are described in Clause 8. The \text{attribute-specifier-seq} in a \text{simple-declaration} appertains to each of the entities declared by the \text{declarators} of the \text{init-declarator-list}. [Note: In the declaration for an entity, attributes appertaining
to that entity may appear at the start of the declaration and after the 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 (3.3); the scope rules are summarized in 3.4. 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 7 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 9) or enumeration (7.2), that is, when the decl-specifier-seq contains either a class-specifier, an elaborated-type-specifier with a class-key (9.1), 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 constant expression (5.20) that can be contextually converted to bool (Clause 4). 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 (1.4) shall include the text of the string-literal, if one is supplied, except that characters not in the basic source character set (2.3) are not required to appear in the diagnostic message. [Example:

static_assert(sizeof(long) >= 8, "64-bit code generation required for this library.");

— end example]

7 An empty-declaration has no effect.

8 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 type-specifiers (7.1.6) in the decl-specifier-seq and the recursive declarator structure of the init-declarator describe a type (8.3), which is then associated with the name being declared by the init-declarator.

9 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 (7.1.3). 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 (8.3.5) and an object declaration otherwise.

10 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 (3.1). A definition causes the appropriate amount of storage to be reserved and any appropriate initialization (8.5) to be done.

11 Only in function declarations for constructors, destructors, and type conversions can the decl-specifier-seq be omitted.  

93) The “implicit int” rule of C is no longer supported.
7.1 Specifiers

The specifiers that can be used in a declaration are

\[
\text{decl-specifier:}
\]

\[
\begin{align*}
\text{storage-class-specifier} \\
\text{type-specifier} \\
\text{function-specifier} \\
\text{friend} \\
\text{typedef} \\
\text{constexpr}
\end{align*}
\]

\text{decl-specifier-seq:}

\[
\text{decl-specifier attribute-specifier-seq opt}
\]

\[
\text{decl-specifier decl-specifier-seq}
\]

The optional attribute-specifier-seq in a decl-specifier-seq appertains to the type determined by the preceding decl-specifiers (8.3). The attribute-specifier-seq affects the type only for the declaration it appears in, not other declarations involving the same type.

Each decl-specifier shall appear at most once in the complete decl-specifier-seq of a declaration, except that long may appear twice.

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 type-specifier other than a cv-qualifier in the decl-specifier-seq. The sequence shall be self-consistent as described below. [Example:

\[
\begin{align*}
typedef \text{char* Pc}; \\
\text{static Pc;} & \quad // \text{error: name missing}
\end{align*}
\]

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,

\[
\begin{align*}
\text{void f(const Pc);} & \quad // \text{void f(char* const) (not const char*)} \\
\text{void g(const int Pc);} & \quad // \text{void g(const int)}
\end{align*}
\]

—end example]

[Note: Since signed, unsigned, long, and short by default imply int, a type-name appearing after one of those specifiers is treated as the name being (re)declared. [Example:

\[
\begin{align*}
\text{void h(unsigned Pc);} & \quad // \text{void h(unsigned int)} \\
\text{void k(unsigned int Pc);} & \quad // \text{void k(unsigned int)}
\end{align*}
\]

—end example] — end note]

7.1.1 Storage class specifiers

The storage class specifiers are

\[
\text{storage-class-specifier:}
\]

\[
\begin{align*}
\text{register} \\
\text{static} \\
\text{thread_local} \\
\text{extern} \\
\text{mutable}
\end{align*}
\]

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** 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** (9.5)). The **storage-class-specifier** applies to the name declared by each **init-declator** 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 (14.7.3) or an explicit instantiation (14.7.2) directive.

2 The **register** specifier shall be applied only to names of variables declared in a block (6.3) or to function parameters (8.4). It specifies that the named variable has automatic storage duration (3.7.3). A variable declared without a **storage-class-specifier** at block scope or declared as a function parameter has automatic storage duration by default.

3 A **register** specifier is a hint to the implementation that the variable so declared will be heavily used. [Note: The hint can be ignored and in most implementations it will be ignored if the address of the variable is taken. This use is deprecated (see D.2). — end note]

4 The **thread_local** specifier indicates that the named entity has thread storage duration (3.7.2). 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**.

5 The **static** specifier can be applied only to names of variables and functions and to anonymous unions (9.5). 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 (3.7.1), unless accompanied by the **thread_local** specifier, which declares the variable to have thread storage duration (3.7.2). A **static** specifier can be used in declarations of class members; 9.4 describes its effect. For the linkage of a name declared with a **static** specifier, see 3.5.

6 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 3.5. [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]

7 The linkages implied by successive declarations for a given entity shall agree. That is, within a given scope, each declaration declaring the same variable name or the same overloading of a function name shall imply the same linkage. Each function in a given set of overloaded functions can have a different linkage, however. [Example:

```c

static char* f();        // f() has internal linkage
char* f()                // f() still has internal linkage
{ /* ... */ }

char* g();               // g() has external linkage
static char* g()         // error: inconsistent linkage
{ /* ... */ }

void h();               // external linkage
inline void h();        // external linkage

inline void l();        // external linkage
void l();               // external linkage

inline void m();        // external linkage
extern void m();        // external linkage

static void n();        // internal linkage
inline void n();        // internal linkage
```
```
static int a; // a has internal linkage
int a; // error: two definitions
static int b; // b has internal linkage
extern int b; // b still has internal linkage
int c; // c has external linkage
static int c; // error: inconsistent linkage
extern int d; // d has external linkage
static int d; // error: inconsistent linkage
```

— end example]

8 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
}
```

— end example]

9 The mutable specifier shall appear only in the declaration of a non-static data member (9.2) whose type is neither const-qualified nor a reference type. [Example:

```c
class X {
  mutable const int* p; // OK
  mutable int* const q; // ill-formed
};
```

— end example]

10 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 (7.1.6.1).

### 7.1.2 Function specifiers

1 Function-specifiers can be used only in function declarations.

```c
function-specifier:
  inline
  virtual
  explicit
```

2 A function declaration (8.3.5, 9.3, 11.3) 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 defined by 7.1.2 shall still be respected.
A function defined within a class definition is an inline function. The *inline* specifier shall not appear on a block scope function declaration.⁹⁴ If the *inline* specifier is used in a friend declaration, that declaration shall be a definition or the function shall have previously been declared inline.

An inline function shall be defined in every translation unit in which it is odr-used and shall have exactly the same definition in every case (3.2). [Note: A call to the inline function may be encountered before its definition appears in the translation unit. — end note] If the definition of a function appears in a translation unit before its first declaration as inline, the program is ill-formed. If a function 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 with external linkage shall have the same address in all translation units. A static local variable in an extern inline function always refers to the same object. A type defined within the body of an extern inline function is the same type in every translation unit.

The *virtual* specifier shall be used only in the initial declaration of a non-static class member function; see 10.3.

The *explicit* specifier shall be used only in the declaration of a constructor or conversion function within its class definition; see 12.3.1 and 12.3.2.

### 7.1.3 The *typedef* specifier

Declarations containing the *decl-specifier* typedef declare identifiers that can be used later for naming fundamental (3.9.1) or compound (3.9.2) types. The typedef specifier shall not be combined in a *decl-specifier-seq* with any other kind of specifier except a *type-specifier*, and it shall not be used in the *decl-specifier-seq* of a parameter-declaration (8.3.5) nor in the *decl-specifier-seq* of a function-definition (8.4).

```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 8. A `typedef-name` is thus a synonym for another type. A `typedef-name` does not introduce a new type the way a class declaration (9.1) or enum declaration does. [Example: after

```cpp
typedef int MILES, *KLICKSP;
```

the constructions

```cpp
MILES distance;
extern KLICKSP metricp;
```

are all correct declarations; the type of `distance` is `int` and that of `metricp` is “pointer to `int`.” — end example]

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`. It has the same semantics as if it were introduced by the typedef specifier. In particular, it does not define a new type and it shall not appear in the type-id. [Example:

```cpp
using handler_t = void (*)(int);
extern handler_t ignore;
extern void (*ignore)(int); // redeclare ignore
using cell = pair<void*, cell*>; // ill-formed
```

— end example]

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. [Example:

```

⁹⁴ The inline keyword has no effect on the linkage of a function.

§ 7.1.3 152
typedef struct s { /* ... */ } s;
typedef int I;
typedef int I;
typedef I I;

— end example]

4 In a given class scope, a typedef specifier can be used to 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;
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 { /* ... */};
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 (9.1). If a typedef-name is used to identify the subject of an elaborated-type-specifier (7.1.6.3), a class definition (Clause 9), a constructor declaration (12.1), or a destructor declaration (12.4), 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
```

§ 7.1.3
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 (3.5). [Example:

```c
typedef struct { } *ps, S;  // S is the class name for linkage purposes
```

— end example]

7.1.4 The friend specifier

The friend specifier is used to specify access to class members; see 11.3.

7.1.5 The constexpr specifier

The constexpr specifier shall be applied only to the definition of a variable or variable template, the declaration of a function or function template, or the declaration of a static data member of a literal type (3.9). 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:

```c
constexpr void square(int &x); // OK: declaration
custom void square(int &x);  // OK: definition
```

constexpr void square(int &x); // OK: declaration
custom void square(int &x); // OK: definition
constexpr void square(int &x) { // OK: declaration
    x *= x;
}

constexpr pixel large(4);    // OK: square defined
int next(constexpr int x) {  // error: not for parameters
    return x + 1;
}

extern constexpr int memsz;  // error: not a definition

— end example]

A constexpr 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. constexpr functions and constexpr constructors are implicitly inline (7.1.2).

The definition of a constexpr function shall satisfy the following constraints:

(3.1) it shall not be virtual (10.3);
(3.2) its return type shall be a literal type;
(3.3) each of its parameter types shall be a literal type;
— its \textit{function-body} shall be \texttt{= delete}, \texttt{= default}, or a \textit{compound-statement} that does not contain
\begin{itemize}
\item an \texttt{asm-definition},
\item a \texttt{goto} statement,
\item a \texttt{try-block}, or
\item a definition of a variable of non-literal type or of static or thread storage duration or for which no initialization is performed.
\end{itemize}

\begin{example}

\begin{verbatim}
constexpr int square(int x) {
    return x * x; }
// OK
constexpr long long_max() {
    return 2147483647; }
// OK
constexpr int abs(int x) {
    if (x < 0)
        x = -x;
    return x;  // OK
}
constexpr int first(int n) {
    static int value = n;
    // error: variable has static storage duration
    return value;
}
constexpr int uninit() {
    int a;
    // error: variable is uninitialized
    return a;
}
constexpr int prev(int x) {
    return --x; }
// OK
constexpr int g(int x, int n) {
    int r = 1;
    while (--n > 0) r *= x;
    return r;
}
\end{verbatim}
\end{example}

— end example

\begin{itemize}
\item The definition of a \texttt{constexpr} constructor shall satisfy the following constraints:
\begin{itemize}
\item the class shall not have any virtual base classes;
\item each of the parameter types shall be a literal type;
\item its \textit{function-body} shall not be a \texttt{function-try-block};
\end{itemize}
\end{itemize}

In addition, either its \textit{function-body} shall be \texttt{= delete}, or it shall satisfy the following constraints:
\begin{itemize}
\item either its \textit{function-body} shall be \texttt{= default}, or the \textit{compound-statement} of its \textit{function-body} shall satisfy the constraints for a \texttt{function-body} of a \texttt{constexpr} function;
\item every non-variant non-static data member and base class sub-object shall be initialized (12.6.2);
\item if the class is a union having variant members (9.5), exactly one of them shall be initialized;
\item 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;
\item for a non-delegating constructor, every constructor selected to initialize non-static data members and base class sub-objects shall be a \texttt{constexpr} constructor;
\end{itemize}
— for a delegating constructor, the target constructor shall be a constexpr constructor.

[Example:

```cpp
struct Length {
  constexpr explicit Length(int i = 0) : val(i) { }
private:
  int val;
};
```

— end example]

5 For a non-template, non-defaulted constexpr function or a non-template, non-defaulted, non-inheriting constexpr constructor, if no argument values exist such that an invocation of the function or constructor could be an evaluated subexpression of a core constant expression (5.20), or, for a constructor, a constant initializer for some object (3.6.2), the program is ill-formed; no diagnostic required. [Example:

```cpp
class int f(bool b) {
  return b ? throw 0 : 0 ; }
constexpr int f() { return f(true); } // ill-formed, no diagnostic required
struct B {
  constexpr B(int x) : i(0) { }
  int i;
};
int global;
struct D : B {
  constexpr D() : B(global) { }
  // ill-formed, no diagnostic required
  // lvalue-to-rvalue conversion on non-constant global
};
```

— end example]

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 a call to a constexpr function can appear in a constant expression.

8 The constexpr specifier has no effect on the type of a constexpr function or a constexpr constructor. [Example:

```cpp
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. If it is initialized by a constructor call, that call shall be a constant
expression (5.20). Otherwise, or if a constexpr specifier is used in a reference declaration, every full-expression that appears in its initializer shall be a constant expression. [Note: Each implicit conversion used in converting the initializer expressions and each constructor call used for the initialization is part of such a full-expression. — end note] [Example:

```cpp
struct pixel {
    int x, y;
};
constexpr pixel ur = { 1294, 1024 }; // OK
constexpr pixel origin; // error: initializer missing
```

— end example]

### 7.1.6 Type specifiers

The type-specifiers are

```plaintext
type-specifier:
    trailing-type-specifier
    class-specifier
    enum-specifier

trailing-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

trailing-type-specifier-seq:
    trailing-type-specifier attribute-specifier-seq_opt
    trailing-type-specifier trailing-type-specifier-seq
```

The optional attribute-specifier-seq in a type-specifier-seq or a trailing-type-specifier-seq appertains to the type denoted by the preceding type-specifiers (8.3). The attribute-specifier-seq affects the type only for the declaration it appears in, not other declarations involving the same type.

As a general rule, at most one type-specifier is allowed in the complete decl-specifier-seq of a declaration or in a type-specifier-seq or trailing-type-specifier-seq. The only exceptions to this rule are the following:

1. **const** can be combined with any type specifier except itself.
2. **volatile** can be combined with any type specifier except itself.
3. **signed** or **unsigned** can be combined with **char**, **long**, **short**, or **int**.
4. **short** or **long** can be combined with **int**.
5. **long** can be combined with **double**.
6. **long** can be combined with **long**.

Except in a declaration of a constructor, destructor, or conversion function, at least one type-specifier that is not a cv-qualifier shall appear in a complete type-specifier-seq or a complete decl-specifier-seq. A type-specifier-seq shall not define a class or enumeration unless it appears in the type-id of an alias-declaration (7.1.3) that is not the declaration of a template-declaration.

95) 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.
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 of the declaration shall not be empty. [Note: 3.9.3 and 8.3.5 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]

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 (5.2.11). — end note]

Except that any class member declared mutable (7.1.1) can be modified, any attempt to modify a const object during its lifetime (3.8) results in undefined behavior. [Example:

```c
const int ci = 3; // cv-qualified (initialized as required)
ci = 4; // ill-formed: attempt to modify const

int i = 2; // not cv-qualified
const int* cip; // pointer to const int
cip = &i; // OK: cv-qualified access path to unqualified
*cip = 4; // ill-formed: attempt to modify through ptr to const

int* ip;
ip = const_cast<int*>(cip); // cast needed to convert const int* to int*
*ip = 4; // defined: *ip points to i, a non-const object

const int* ciq = new const int (3); // initialized as required
int* iq = const_cast<int*>(ciq); // cast required
*iq = 4; // undefined: modifies a const object
```

For another example

```c
struct X {
    mutable int i;
    int j;
};
struct Y {
    X x;
    Y();
};

const Y y;
y.x.i++; // well-formed: mutable member can be modified
y.x.j++; // ill-formed: const-qualified member modified
Y* p = const_cast<Y*>(&y); // cast away const-ness of y
p->x.i = 99; // well-formed: mutable member can be modified
p->x.j = 99; // undefined: modifies a const member
```

— end example]
What constitutes an access to an object that has volatile-qualified type is implementation-defined. If an attempt is made to refer to an object defined with a volatile-qualified type through the use of a glvalue with a non-volatile-qualified type, the program 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 1.9 for detailed semantics. In general, the semantics of volatile are intended to be the same in C++ as they are in C. — end note]

7.1.6.2 Simple type specifiers [dcl.type.simple]

The simple type specifiers are

\[
\text{simple-type-specifier:} \\
\text{\hspace{1em} nested-name-specifier_opt type-name} \\
\text{\hspace{1em} nested-name-specifier template simple-template-id} \\
\text{\hspace{1em} char} \\
\text{\hspace{1em} char16_t} \\
\text{\hspace{1em} char32_t} \\
\text{\hspace{1em} wchar_t} \\
\text{\hspace{1em} bool} \\
\text{\hspace{1em} short} \\
\text{\hspace{1em} int} \\
\text{\hspace{1em} long} \\
\text{\hspace{1em} signed} \\
\text{\hspace{1em} unsigned} \\
\text{\hspace{1em} float} \\
\text{\hspace{1em} double} \\
\text{\hspace{1em} void} \\
\text{\hspace{1em} auto} \\
\text{\hspace{1em} decltype-specifier} \\
\text{\hspace{1em} type-name:} \\
\text{\hspace{1.5em} class-name} \\
\text{\hspace{1.5em} enum-name} \\
\text{\hspace{1.5em} typedef-name} \\
\text{\hspace{1.5em} simple-template-id} \\
\text{\hspace{1em} decltype-specifier:} \\
\text{\hspace{1.5em} decltype ( expression )} \\
\text{\hspace{1.5em} decltype ( auto )}
\]

1 The simple type specifiers are

2 The simple-type-specifier auto is a placeholder for a type to be deduced (7.1.6.4). The other simple-type-specifiers specify either a previously-declared type, a type determined from an expression, or one of the fundamental types (3.9.1). Table 9 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]
Table 9 — *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 14.2</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</code></td>
<td>“unsigned long long int”</td>
</tr>
<tr>
<td><code>unsigned long long</code></td>
<td>“unsigned long long int”</td>
</tr>
<tr>
<td><code>signed long int</code></td>
<td>“long int”</td>
</tr>
<tr>
<td><code>signed long</code></td>
<td>“long int”</td>
</tr>
<tr>
<td><code>signed long long</code></td>
<td>“long long int”</td>
</tr>
<tr>
<td><code>signed long long</code></td>
<td>“long long int”</td>
</tr>
<tr>
<td><code>long long int</code></td>
<td>“long long int”</td>
</tr>
<tr>
<td><code>long long</code></td>
<td>“long long int”</td>
</tr>
<tr>
<td><code>long int</code></td>
<td>“long int”</td>
</tr>
<tr>
<td><code>long</code></td>
<td>“long int”</td>
</tr>
<tr>
<td><code>signed short int</code></td>
<td>“short int”</td>
</tr>
<tr>
<td><code>signed short</code></td>
<td>“short int”</td>
</tr>
<tr>
<td><code>short int</code></td>
<td>“short int”</td>
</tr>
<tr>
<td><code>short</code></td>
<td>“short int”</td>
</tr>
<tr>
<td><code>wchar_t</code></td>
<td>“wchar_t”</td>
</tr>
<tr>
<td><code>float</code></td>
<td>“float”</td>
</tr>
<tr>
<td><code>double</code></td>
<td>“double”</td>
</tr>
<tr>
<td><code>long double</code></td>
<td>“long double”</td>
</tr>
<tr>
<td><code>void</code></td>
<td>“void”</td>
</tr>
<tr>
<td><code>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>
For an expression e, the type denoted by decltype(e) is defined as follows:

(4.1) — if e is an unparenthesized id-expression or an unparenthesized class member access (5.2.5), decltype(e) is the type of the entity named by e. If there is no such entity, or if e names a set of overloaded functions, the program is ill-formed;

(4.2) — otherwise, if e is an xvalue, decltype(e) is T&&, where T is the type of e;

(4.3) — otherwise, if e is an lvalue, decltype(e) is T&, where T is the type of e;

(4.4) — otherwise, decltype(e) is the type of e.

The operand of the decltype specifier is an unevaluated operand (Clause 5).

[Example:

```cpp
class A {
    int x;
    public:
    A() { x = 42; }
    A(int&& x) : x_(x) {}
};

int main() {
    static A a;
    a.x = 7;
    int x = 17;
    std::cout << x + a.x;  // OK. (a.x is an lvalue)
    return 0;
}
```
— end example] [Note: The rules for determining types involving decltype(auto) are specified in 7.1.6.4.
— end note]

[Note: in the case where the operand of a decltype-specifier is a function call and the return type of the function is a class type, a special rule (5.2.2) ensures that the return type is not required to be complete (as it would be if the call appeared in a sub-expression or outside of a decltype-specifier). 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. [Example:

```cpp
#include <iostream>

template<class T> struct A {
    ~A() = delete;
};

int main() {
    template<class T> auto h() -> A<T>;
    template<class T> auto i(T) -> T;
    template<class T> auto f(T) -> decltype(i(h<T>()));
    template<class T> auto f(T) -> void;
    auto g() -> void {
        f(42);
    }
    template<class T> auto q(T) -> decltype((h<T>()));
    void r() {
    }
```
q(42);        // Error: deduction against q succeeds, so overload resolution
            // selects the specialization “q(T) -> decltype((h<T>())) [with T=int] ”.
            // The return type is A<int>, so a temporary is introduced and its
            // destructor is used, so the program is ill-formed.
}

--- end example] — end note"

### 7.1.6.3 Elaborated type specifiers

elaborated-type-specifier:
   - class-key attribute-specifier-seq\textsubscript{opt} nested-name-specifier\textsubscript{opt} identifier
   - class-key simple-template-id
   - class-key nested-name-specifier template\textsubscript{opt} simple-template-id

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 (14.7.3), an explicit instantiation (14.7.2) or it has one of the following forms:

   - class-key attribute-specifier-seq\textsubscript{opt} identifier ;
   - friend class-key ::\textsubscript{opt} identifier ;
   - friend class-key ::\textsubscript{opt} simple-template-id ;
   - friend class-key nested-name-specifier identifier ;
   - friend class-key nested-name-specifier template\textsubscript{opt} simple-template-id ;

In the first case, the *attribute-specifier-seq*, if any, appertains to the class being declared; the attributes in the *attribute-specifier-seq* are thereafter considered attributes of the class whenever it is named.

2 3.4.4 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 (11.3). — 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 (7.2), the *union* *class-key* shall be used to refer to a union (Clause 9), and either the *class* or *struct* *class-key* shall be used to refer to a class (Clause 9) declared using the *class* or *struct* *class-key*.

[ Example:

enum class E { a, b };
enum E x = E::a;         // OK

— end example ]

### 7.1.6.4 auto specifier

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.
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` (8.3.5), that 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 `return` statements in the body of the function, if any.

If the `auto` type-specifier appears as one of the `decl-specifiers` in the `decl-specifier-seq` of a parameter-declaration of a `lambda-expression`, the lambda is a `generic lambda` (5.1.2). [Example:

```
auto glambda = [] (int i, auto a) { return i; }; // OK: a generic lambda
```
—end example]

The type of a variable declared using `auto` or `decltype(auto)` is deduced from its initializer. This use is allowed when declaring variables in a block (6.3), in namespace scope (3.3.6), and in a `for-init-statement` (6.5.3). `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 `init-declarators`, each of which shall have a non-empty `initializer`. In an `initializer` of the form

```
(expression-list)
```

the `expression-list` shall be a single `assignment-expression`.

[Example:

```
auto x = 5;     // OK: x has type int
const auto *v = &x, u = 6; // OK: v has type const int*, u has type const int
static auto y = 0.0;  // OK: y has type double
auto int r;         // error: auto is not a storage-class-specifier
auto f() -> int;    // OK: f returns int
auto g() { return 0.0; } // OK: g returns double
auto h();           // OK: h’s return type will be deduced when it is defined
```
—end example]

A placeholder type can also be used in declaring a variable in the `condition` of a selection statement (6.4) or an iteration statement (6.5), in the `type-specifier-seq` in the `new-type-id` or `type-id` of a `new-expression` (5.3.4), in a `for-range-declaration`, and in declaring a static data member with a `brace-or-equal-initializer` that appears within the `member-specification` of a class definition (9.4.2).

A program that uses `auto` or `decltype(auto)` in a context not explicitly allowed in this section is ill-formed.

When a variable declared using a placeholder type is initialized, or a `return` statement occurs in a function declared with a return type that contains a placeholder type, the deduced return type or variable type is determined from the type of its initializer. In the case of a `return` with no operand or with an operand of type `void`, the declared return type shall be `auto` and the deduced return type is `auto` and the `return` statement is `void`. Otherwise, let `T` be the declared type of the variable or return type of the function. If the placeholder is the `auto` type-specifier, the deduced type is determined using the rules for template argument deduction. If the initialization is direct-list-initialization then the `braced-init-list` shall contain only a single `initializer-clause` `L`. If the deduction is for a `return` statement and the initializer is a `braced-init-list` (8.5.4), the program is ill-formed. Otherwise, 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 (14.8.2.1), where `P` is a function template parameter type and the corresponding argument is the initializer, or `L` in the case of direct-list-initialization. If the deduction fails, the declaration is ill-formed. Otherwise, the type deduced for the variable or return type is obtained by substituting the deduced `U` into `P`. [Example:
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:
  const auto &i = expr;

The type of i is the deduced type of the parameter u in the call f(expr) of the following invented function template:

  template <class U> void f(const U& u);

— end example]

If the placeholder is the decltype(auto) type-specifier, the declared type of the variable or return type of the function shall be the placeholder alone. The type deduced for the variable or return type is determined as described in 7.1.6.2, as though the initializer had been the operand of the decltype. [Example:

  int i;
  int&& f();
  auto x3a = i;    // decltype(x3a) is int
decalltype(auto) x3d = i;  // decltype(x3d) is int
  auto x4a = (i);  // decltype(x4a) is int
decalltype(auto) x4d = (i);  // decltype(x4d) is int&
  auto x5a = f();  // decltype(x5a) is int
decalltype(auto) x5d = f();  // decltype(x5d) is int&
  auto x6a = { 1, 2 };  // decltype(x6a) is std::initializer_list<int>
decalltype(auto) x6d = { 1, 2 };  // error, { 1, 2 } is not an expression
  auto *x7a = &i;  // decltype(x7a) is int*
decalltype(auto)*x7d = &i;  // error, declared type is not plain decltype(auto)

— end example]

8 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 as described above, and if the type that replaces the placeholder type is not the same in each deduction, the program is ill-formed.

[Example:
  auto x = 5, *y = &x;    // OK: auto is int
  auto a = 5, b = { 1, 2 };  // error: different types for auto

— end example]

9 If a function with a declared return type that contains a placeholder type has multiple return statements, the return type is deduced for each return statement. If the type deduced is not the same in each deduction, the program is ill-formed.

10 If a function with a declared return type that uses a placeholder type has no 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:

  auto f() {}  // OK, return type is void
  auto* g() {}  // error, cannot deduce auto* from void()

§ 7.1.6.4
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 `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.

```cpp
auto n = n; // error, n’s type is unknown
auto f();
void g() { &f; } // error, f’s return type is unknown
auto sum(int i) {
    if (i == 1)
        return i; // sum’s return type is int
    else
        return sum(i-1)+i; // OK, sum’s return type has been deduced
}
```

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 `return` statement with a non-type-dependent operand. [Note: Therefore, any use of a specialization of the function template will cause an implicit instantiation. Any errors that arise from this instantiation are not in the immediate context of the function type and can result in the program being ill-formed. — end note] [Example:

```cpp
template <class T> auto f(T t) { return t; } // return type deduced at instantiation time
typedef decltype(f(1)) fint_t; // instantiates f<int> to deduce return type
template<class T> auto f(T* t) { return *t; } // instantiates both f’s to determine return types,
// chooses second
```

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. [Example:

```cpp
auto f();
auto f() { return 42; } // return type is int
auto f(); // OK
int f(); // error, cannot be overloaded with auto f()
decltype(auto) f(); // error, auto and decltype(auto) don’t match
```

```cpp
template <typename T> auto g(T t) { return t; } // #1
template auto g(int); // OK, return type is int
template char g(char); // error, no matching template
template<> auto g(double); // OK, forward declaration with unknown return type
```

```cpp
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
```

```cpp
void h() { return g(42); } // error, ambiguous
```

```cpp
template <typename T> struct A {
    friend T frf(T);
};
auto frf(int i) { return i; } // not a friend of A<int>
```
14 A function declared with a return type that uses a placeholder type shall not be virtual (10.3).

15 An explicit instantiation declaration (14.7.2) does not cause the instantiation of an entity declared using a placeholder type, but it also does not prevent that entity from being instantiated as needed to determine its type. [Example:

```cpp
template <typename T> auto f(T t) { return t; }
extern template auto f(int); // does not instantiate f<int>
int (*p)(int) = f; // instantiates f<int> to determine its return type, but an explicit
                      // instantiation definition is still required somewhere in the program
```

—end example—

7.2 Enumeration declarations [dcl.enum]

1 An enumeration is a distinct type (3.9.2) 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 identifier_opt enum-base_opt
  enum-key attribute-specifier-seq_opt nested-name-specifier identifier
  enum-base_opt
opaque-enum-declaration:
  enum-key attribute-specifier-seq_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 identifier” 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:

```cpp
struct S {
  enum E : int {}; // error: redeclaration of enumeration
};
```

—end example—

§ 7.2 — end note]
The enumeration type declared with an enum-key of only enum is an unscoped enumeration, and its enumerators are unscoped enumerators. The enum-keys enum class and enum struct are semantically equivalent; an enumeration type declared with one of these is a scoped enumeration, and its enumerators are scoped enumerators. The optional 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 enum-list are declared as constants, and can appear wherever constants are required. An enum-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 enum-definition without an initializer gives the enumerator the value obtained by increasing the value of the previous enumerator by one. [Example:

```
enum { a, b, c=0 };
enum { d, e, f=e+2 };
```
]

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 (5.20). 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 (5.20). 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 (3.3.2) 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 \( e_{\text{min}} \) is the smallest enumerator and \( e_{\text{max}} \) is the largest, the values of the enumeration are the values in the range \( b_{\text{min}} \) to \( b_{\text{max}} \), defined as follows: Let \( K \) be 1 for a two's complement representation and 0 for a one's complement or sign-magnitude representation. \( b_{\text{max}} \) is the smallest value greater than or equal to \( \max(\lvert e_{\text{min}} \rvert - K, \lvert e_{\text{max}} \rvert) \) and equal to \( 2^M - 1 \), where \( M \) is a non-negative integer. \( b_{\text{min}} \) is zero if \( e_{\text{min}} \) is non-negative and \( -(b_{\text{max}} + K) \) otherwise. The size of the smallest bit-field large enough to hold all the values of the enumeration type is \( \max(M, 1) \) if \( b_{\text{min}} \) 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 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 layout-compatible 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 (4.5). Example:

```
enum color { red, yellow, green=20, blue };
color col = red;
color* cp = &col;
if (*cp == blue) // ...
```

makes color a type describing various colors, and then declares col as an object of that type, and cp as a pointer to an object of that type. The possible values of an object of type color are red, yellow, green, blue; these values can be converted to the integral values 0, 1, 20, and 21. Since enumerations are distinct types, objects of type color can be assigned only values of type color.

```
color c = 1;  // error: type mismatch,
              // no conversion from int to color

int i = yellow;  // OK: yellow converted to integral value 1
                 // integral promotion
```

Note that this implicit enum to int conversion is not provided for a scoped enumeration:

```
enum class Col { red, yellow, green };
int x = Col::red;  // error: no Col to int conversion
Col y = Col::red;
if (y) { }  // error: no Col to bool conversion
```

— end example] [96] This set of values is used to define promotion and conversion semantics for the enumeration type. It does not preclude an expression of enumeration type from having a value that falls outside this range.

§ 7.2
d = direction::right;    // OK
}

enum class altitude { high='h', low='l' }

void h() {
    altitude a;         // OK
    a = high;           // error: high not in scope
    a = altitude::low;  // OK
}

— end example —

An enumerator declared in class scope can be referred to using the class member access operators (::, . (dot) and -> (arrow)), see 5.2.5. [Example:

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 —

7.3 Namespaces

A namespace is an optionally-named declarative region. The name of a namespace can be used to access entities declared in that namespace; that is, the members of the namespace. Unlike other declarative regions, the definition of a namespace can be split over several parts of one or more translation units.

The outermost declarative region of a translation unit is a namespace; see 3.3.6.

7.3.1 Namespace definition

The grammar for a namespace-definition is

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-namescope-specifier :: identifier { namespace-body }
    enclosing-namescope-specifier:
        identifier enclosing-namescope-specifier :: identifier

§ 7.3.1
namespace-body:
  declaration-seq

2 Every namespace-definition shall appear in the global scope or in a namespace scope (3.3.6).

3 In a named-namespace-definition, the identifier is the name of the namespace. If the identifier, when looked up (3.4.1), refers to a namespace-name (but not a namespace-alias) introduced in the declarative region in which the named-namespace-definition appears, 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.

4 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:

```c
namespace Outer {
  int i;
  namespace Inner {
    void f() { i++; }  // Outer::i
    int i;
    void g() { i++; }  // Inner::i
  }
}
```
—end example]

5 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 7.3.1.2). Such a redeclaration has the same enclosing namespaces as the original declaration. [Example:

```c
namespace Q {
  namespace V {
    void f();  // enclosing namespaces are the global namespace, Q, and Q::V
    class C { void m(); };
  }
  void V::f() { // enclosing namespaces are the global namespace, Q, and Q::V
    extern void h();  // ... so this declares Q::V::h
  }
  void V::C::m() { // enclosing namespaces are the global namespace, Q, and Q::V
  }
}
```
—end example]

6 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.

7 The optional attribute-specifier-seq in a named-namespace-definition appertains to the namespace being defined or extended.

8 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 (3.4.2) whenever one of them is, and a using-directive (7.3.4) that names the inline namespace is implicitly inserted into the enclosing namespace as for an unnamed namespace (7.3.1.1). Furthermore, each member of the inline namespace can subsequently be partially specialized (14.5.5), explicitly instantiated (14.7.2), or explicitly specialized (14.7.3) as though it were a member of the enclosing namespace. Finally, looking up a name in the enclosing namespace via
explicit qualification (3.4.3.2) 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 } }
```

[Example:
```cpp
namespace A::B::C {
    int i;
}
```

The above has the same effect as:

```cpp
namespace A {
    namespace B {
        namespace C {
            int i;
        }
    }
}
```

— end example]

7.3.1.1 Unnamed namespaces [namespace.unnamed]

An unnamed-namespace-definition behaves as if it were replaced by

```cpp
inline opt namespace unique { /* empty body */ }
using namespace unique;
namespace unique { namespace-body }
```

where inline appears if and only if it appears in the unnamed-namespace-definition and all occurrences of unique in a translation unit are replaced by the same identifier, and this identifier differs from all other identifiers in the translation unit. The optional attribute-specifier-seq in the unnamed-namespace-definition appertains to unique. [Example:
```cpp
namespace { int i; } // unique ::i
void f() { i++; } // unique ::i++
```

namespace A {
```cpp
namespace {
    int i; // A:: unique ::i
    int j; // A:: unique ::j
}
void g() { i++; } // A:: unique ::i++
```

using namespace A;
```cpp
void h() {
    i++; // error: unique ::i or h:: unique ::i
    A::i++; // h:: unique ::i
}
```
7.3.1.2 Namespace member definitions

A declaration in a namespace \( \mathcal{N} \) (excluding declarations in nested scopes) whose *declarator-id* is an *unqualified-id* declares (or redeclares) a member of \( \mathcal{N} \), and may be a definition. \( \text{[Note: An explicit instantiation (14.7.2) or explicit specialization (14.7.3) of a template does not introduce a name and thus may be declared using an unqualified-id in a member of the enclosing namespace set, if the primary template is declared in an inline namespace. — end note] [Example:]}

```cpp
namespace X {
    void f() { /* ... */ } // OK: introduces X::f()

    namespace M {
        void g(); // OK: introduces X::M::g()
    }

    using M::g;
    void g(); // error: conflicts with X::M::g()
}
```

Members of a named namespace can also be defined outside that namespace by explicit qualification (3.4.3.2) 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. \( \text{[Example:]}

```cpp
namespace Q {
    namespace V {
        void f();
    }

    void V::f() { /* ... */ } // OK
    void V::g() { /* ... */ } // error: g() is not yet a member of V

    namespace V {
        void g();
    }
}
```

If a *friend* declaration in a non-local class first declares a class, function, class template or function template\( ^{97} \) the friend is a member of the innermost enclosing namespace. The *friend* declaration does not by itself make the name visible to unqualified lookup (3.4.1) or qualified lookup (3.4.3). \( \text{[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 (3.4.2). 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}

---

\( ^{97} \) this implies that the name of the class or function is unqualified.
determine whether the entity has been previously declared shall not consider any scopes outside the innermost enclosing namespace. [Note: The other forms of friend declarations cannot declare a new member of the innermost enclosing namespace and thus follow the usual lookup rules. — end note] [Example:

```cpp
// Assume f and g have not yet been declared.
void h(int);
template <class T> void f2(T);
namespace A {
    class X {
        friend void f(X); // A::f(X) is a friend
    class Y {
        friend void g(); // A::g is a friend
        friend void h(int); // A::h is a friend
            // ::h not considered
        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]

### 7.3.2 Namespace alias

A namespace-alias-definition declares an alternate name for a namespace according to the following grammar:

```plaintext
namespace-alias-definition:
    namespace identifier = qualified-name-specifier ;

qualified-name-specifier:
    nested-name-specifier_opt namespace-name
```

The identifier in a namespace-alias-definition is a synonym for the name of the namespace denoted by the qualified-name-specifier and becomes a namespace-alias. [Note: When looking up a namespace-name in a namespace-alias-definition, only namespace names are considered, see 3.4.6. — 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:

```cpp
namespace Company_with_very_long_name { /* ... */ }
namespace GWVLN = Company_with_very_long_name;
namespace GWVLN = Company_with_very_long_name; // OK: duplicate
```
7.3.3 The using declaration

A `using-declaration` introduces a name into the declarative region in which the `using-declaration` appears. `using-declaration`:

```
using typenameopt nested-name-specifier unqualified-id;
```

The member name specified in a `using-declaration` is declared in the declarative region in which the `using-declaration` appears. [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 a `using-declaration` names a constructor (3.4.3.1), it implicitly declares a set of constructors in the class in which the `using-declaration` appears (12.9); otherwise the name specified in a `using-declaration` is a synonym for a set of declarations in another namespace or class.

Every `using-declaration` is a declaration and a member-declaration and so can 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, the `nested-name-specifier` shall name a base class of the class being defined. If such a `using-declaration` names a constructor, the `nested-name-specifier` shall name a direct base class of the class being defined; otherwise it introduces the set of declarations found by member name lookup (10.2, 3.4.3.1). [Example:

```
class C {
    int g();
};

class D2 : public B {
    using B::f; // OK: B is a base of D2
    using B::e; // OK: e is an enumerator of base B
    using B::x; // OK: x is a union member of base B
    using C::g; // error: C isn't a base of D2
};
```

— end example]

[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 (14.5.2). — end note] If an assignment operator brought from a base class into a derived class scope has the signature of a copy/move...
assignment operator for the derived class (12.8), the using-declaration does not by itself suppress the implicit declaration of the derived class assignment operator; the copy/move assignment operator from the base class is hidden or overridden by the implicitly-declared copy/move assignment operator of the derived class, as described below.

5 A using-declaration shall not name a template-id. [Example:

```cpp
struct A {
  template <class T> void f(T);
  template <class T> struct X {};
};
struct B : A {
  using A::f<double>;        // ill-formed
  using A::X<int>;           // ill-formed
};
```

—end example]

6 A using-declaration shall not name a namespace.

7 A using-declaration shall not name a scoped enumerator.

8 A using-declaration for a class member shall be a member-declaration. [Example:

```cpp
struct X {
  int i;
  static int s;
};

void f() {
  using X::i;         // error: X::i is a class member
  // and this is not a member declaration.
  using X::s;         // error: X::s is a class member
  // and this is not a member declaration.
}
```

—end example]

9 Members declared by a using-declaration can be referred to by explicit qualification just like other member names (3.4.3.2). [Example:

```cpp
void f();

namespace A {
  void g();
}

namespace X {
  using ::f;         // global f
  using A::g;       // A's g
}

void h()
{
  X::f();           // calls ::f
  X::g();           // calls A::g
}
```

—end example]

§ 7.3.3
A *using-declaration* is a *declaration* and can therefore be used repeatedly where (and only where) multiple declarations are allowed.  

```c
namespace A {
  int i;
}

namespace A1 {
  using A::i;
  using A::i; // OK: double declaration
}

void f() {
  using A::i;
  using A::i; // error: double declaration
}

struct B {
  int i;
};

struct X : B {
  using B::i;
  using B::i; // error: double member declaration
};
```

— end example —

Members added to the namespace after the *using-declaration* are not considered when a use of the name is made.  

```c
namespace A {
  void f(int);
}

using A::f; // f is a synonym for A::f;

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 (14.5.5). —end note]

13 Since a using-declaration is a declaration, the restrictions on declarations of the same name in the same declarative region (3.3) also apply to using-declarations. [Example:

```c
namespace A {
    int x;
}

namespace B {
    int i;
    struct g {}; struct x {}; void f(int); void f(double); void g(char); // OK: hides struct g
}

void func() {
    int i;
    using B::i; // error: i declared twice
    void f(char);
    using B::f; // OK: each f is a function
    f(3.5); // calls B::f(double)
    using B::g;
    g('a'); // calls B::g(char)
    struct g gl; // gl has class type B::g
    using B::x;
    using A::x; // OK: hides struct B::x
    x = 99; // assigns to A::x
    struct x xl; // xl has class type B::x
}
```

— end example]

14 If a function declaration in namespace scope or block scope has the same name and the same parameter-type-list (8.3.5) 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:

```c
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)
}```

§ 7.3.3
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-declaration* brings names from a base class into a derived class scope, 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 (8.3.5), cv-qualification, and ref-qualifier (if any) in a base class (rather than conflicting). [Note: For *using-declarations* that name a constructor, see 12.9. — end note] [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)
}

— end example ]

16 For the purpose of overload resolution, the functions which are introduced by a *using-declaration* into a derived class will be treated as though they were members of the derived class. In particular, the implicit *this* parameter shall be treated as if it were a pointer to the derived class rather than to the base class. This has no effect on the type of the function, and in all other respects the function remains a member of the base class.

17 The access rules for inheriting constructors are specified in 12.9; otherwise all instances of the name mentioned in a *using-declaration* shall be accessible. In particular, if a derived class uses a *using-declaration* to access a member of a base class, the member name shall be accessible. If the name is that of an overloaded member function, then all functions named shall be accessible. The base class members mentioned by a *using-declaration* shall be visible in the scope of at least one of the direct base classes of the class where the *using-declaration* is specified. [Note: Because a *using-declaration* designates a base class member (and not a member subobject or a member function of a base class subobject), a *using-declaration* cannot be used to resolve inherited member ambiguities. For example,
struct A { int x(); };  
struct B : A { };  
struct C : A {  
    using A::x;  
    int x(int);  
};  
struct D : B, C {  
    using C::x;  
    int x(double);  
};  
int f(D* d) {  
    return d->x();  // ambiguous: B::x or C::x  
}  

—end note]  

18 The alias created by the using-declaration has the usual accessibility for a member-declaration. [Note: A using-declaration that names a constructor does not create aliases; see 12.9 for the pertinent accessibility rules. —end note] [Example:

class A {  
    private:  
        void f(char);  
    public:  
        void f(int);  
    protected:  
        void g();  
};  

class B : public A {  
    using A::f;  // error: A::f(char) is inaccessible  
    public:  
        using A::g;  // B::g is a public synonym for A::g  
};  

—end example]

19 If a using-declaration uses the keyword typename and specifies a dependent name (14.6.2), the name introduced by the using-declaration is treated as a typedef-name (7.1.3).

7.3.4 Using directive [namespace.udir]

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 3.4.6. —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 (3.4.1), 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:

namespace A {

§ 7.3.4

179
int i;
namespace B {
    namespace C {
        int i;
    }
    using namespace A::B::C;
    void f1() {
        i = 5;       // OK, C::i visible in B and hides A::i
    }
}
namespace D {
    using namespace B;
    using namespace C;
    void f2() {
        i = 5;       // ambiguous, B::C::i or A::i?
    }
    void f3() {
        i = 5;       // uses A::i
    }
    void f4() {
        i = 5;       // ill-formed; neither i is visible
    }
}

— end example]

For unqualified lookup (3.4.1), 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 3.4.3.2. — 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;

§ 7.3.4
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]

5 If a namespace is extended (7.3.1) 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.

6 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,

    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();    // okay: name g refers to the same entity
        h();    // okay: overload resolution selects A::h
    }

    — end note]  

7 During overload resolution, all functions from the transitive search are considered for argument matching. The set of declarations found by the transitive search is unordered. [Note: In particular, the order in which namespaces were considered and the relationships among the namespaces implied by the using-directives do not cause preference to be given to any of the declarations found by the search. — end note] An ambiguity exists if the best match finds two functions with the same signature, even if one is in a namespace reachable through using-directives in the namespace of the other.98 [Example:

    namespace D {
        int d1;

98) During name lookup in a class hierarchy, some ambiguities may be resolved by considering whether one member hides the other along some paths (10.2). There is no such disambiguation when considering the set of names found as a result of following using-directives.

§ 7.3.4
void f(char);
}  
using namespace D;

int d1;  // OK: no conflict with D::d1

namespace E {
    int e;
    void f(int);
}

namespace D {  // namespace extension
    int d2;
    using namespace E;
    void f(int);
}

void f() {
    d1++;
    // error: ambiguous ::d1 or D::d1?
    ::d1++;  // OK
    D::d1++;  // OK
    d2++;
    // OK: D::d2
    e++;
    // OK: E::e
    f(1);
    // error: ambiguous: D::f(int) or E::f(int)?
    f('a');  // OK: D::f(char)
}

—end example]  

7.4 The asm declaration  [dcl.asm]

An asm declaration has the form

\[
\text{asm-definition:} \quad \text{asm (string-literal)};
\]

The asm declaration is conditionally-supported; its meaning is implementation-defined. [Note: Typically it is used to pass information through the implementation to an assembler. —end note]

7.5 Linkage specifications  [dcl.link]

All function types, function names with external linkage, and variable names with external linkage have a language linkage. [Note: Some of the properties associated with an entity with language linkage are specific to each implementation and are not described here. For example, a particular language linkage may be associated with a particular form of representing names of objects and functions with external linkage, or with a particular calling convention, etc. —end note] The default language linkage of all function types, function names, and variable names is C++ language linkage. Two function types with different language linkages are distinct types even if they are otherwise identical.

Linkage (3.5) between C++ and non-C++ code fragments can be achieved using a linkage-specification:

\[
\begin{align*}
\text{linkage-specification:} \\
\text{extern string-literal (declaration-seq\textopt)} \\
\text{extern string-literal declaration}
\end{align*}
\]

The string-literal indicates the required language linkage. This International Standard specifies the semantics for the string-literals "C" and "C++": Use of a string-literal other than "C" or "C++" is conditionally-supported, with implementation-defined semantics. [Note: Therefore, a linkage-specification with a string-literal that is unknown to the implementation requires a diagnostic. —end note] [Note: It is recommended
that the spelling of the *string-literal* be taken from the document defining that language. For example, *Ada* (not *ADA*) and *Fortran* or *FORTRAN*, depending on the vintage. —end note]

3 Every implementation shall provide for linkage to functions written in the C programming language, "C", and linkage to C++ functions, "C++". [Example:

```c
complex sqrt(complex); // C++ linkage by default
extern "C" {
    double sqrt(double); // C linkage
}
```
—end example]

4 Linkage specifications nest. When linkage specifications nest, the innermost one determines the language linkage. A linkage specification does not establish a scope. A *linkage-specification* shall occur only in namespace scope (3.3). In a *linkage-specification*, the specified language linkage applies to the function types of all function declarators, function names with external linkage, and variable names with external linkage declared within the *linkage-specification*. [Example:

```c
extern "C" void f1(void(*pf)(int)); // the name f1 and its function type have C language linkage
    // pf is a pointer to a C function
extern "C" typedef void FUNC();
    FUNC f2; // the name f2 has C++ language linkage and the
    // function's type has C language linkage
extern "C" FUNC f3;
    // the name of function f3 and the function's type
    // have C language linkage
void (*pf2)(FUNC*);
    // the name of the variable pf2 has C++ linkage and
    // the type of pf2 is pointer to C function
    // takes one parameter of type pointer to C function
extern "C" {
    static void f4(); // the name of the function f4 has
    // internal linkage (not C language
    // linkage) and the function's type
    // has C language linkage.
}

extern "C" void f5() {
    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.
}
```
A C language linkage is ignored in determining the language linkage of the names of class members and the function type of class member functions. [Example:

```c
extern "C" typedef void FUNC_c();

class C {
  void mf1(FUNC_c*);
  // the name of the function mf1 and the member
  // function's type have C++ language linkage; the
  // parameter has type pointer to C function
  FUNC_c mf2;
  // the name of the function mf2 and the member
  // function's type have C++ language linkage
  static FUNC_c* q;
  // the name of the data member q has C++ language
  // linkage and the data member's type is pointer to
  // C function
};

extern "C" {
  class X {
    void mf();
    // the name of the function mf and the member
    // function's type have C++ language linkage
    void mf2(void(*)());
    // the name of the function mf2 has C++ language
    // linkage; the parameter has type pointer to
    // C function
  }
};
}
```

— end example —

If two declarations declare functions with the same name and parameter-type-list (8.3.5) to be members of the same namespace or declare objects with the same name to be members of the same namespace and the declarations give the names different language linkages, the program is ill-formed; no diagnostic is required if the declarations appear in different translation units. Except for functions with C++ linkage, a function declaration without a linkage specification shall not precede the first linkage specification for that function. A function can be declared without a linkage specification after an explicit linkage specification has been seen; the linkage explicitly specified in the earlier declaration is not affected by such a function declaration.

At most one function with a particular name can have C language linkage. Two declarations for a function with C language linkage with the same function name (ignoring the namespace names that qualify it) that appear in different namespace scopes refer to the same function. Two declarations for a variable with C language linkage with the same name (ignoring the namespace names that qualify it) that appear in different namespace scopes refer to the same variable. An entity with C language linkage shall not be declared with the same name as a variable in global scope, unless both declarations denote the same entity; no diagnostic is required if the declarations appear in different translation units. A variable with C language linkage shall not be declared with the same name as a function with C language linkage (ignoring the namespace names that qualify the respective names); no diagnostic is required if the declarations appear in different translation units. [Note: Only one definition for an entity with a given name with C language linkage may appear in the program (see 3.2); this implies that such an entity must not be defined in more than one namespace scope. — end note] [Example:

```c
int x;

namespace A {
  extern "C" int f();
  extern "C" int g() { return 1; }
  extern "C" int h();
  extern "C" int x();       // ill-formed: same name as global-space object x
}
```

§ 7.5
namespace B {
extern "C" int f(); // A::f and B::f refer to the same function
extern "C" int g() { return 1; } // ill-formed, the function g
  // with C language linkage has two definitions
}

int A::f() { return 98; } // definition for the function f with C language linkage
extern "C" int h() { return 97; } // definition for the function h with C language linkage
  // A::h and ::h refer to the same function

— end example ]

7 A declaration directly contained in a linkage-specification is treated as if it contains the extern specifier (7.1.1) 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 ]

8 [ Note: Because the language linkage is part of a function type, when indirecting through a pointer to C function, the function to which the resulting lvalue refers is considered a C function. — end note ]

9 Linkage from C++ to objects defined in other languages and to objects defined in C++ from other languages is implementation-defined and language-dependent. Only where the object layout strategies of two language implementations are similar enough can such linkage be achieved.

7.6 Attributes [dcl.attr]

7.6.1 Attribute syntax and semantics [dcl.attr.grammar]

1 Attributes specify additional information for various source constructs such as types, variables, names, blocks, or translation units.

attribute-specifier-seq:
  attribute-specifier-seq opt attribute-specifier
attribute-specifier:
  [ [ attribute-list ] ]
alignment-specifier:
  alignas ( type-id ... opt)
  alignas ( constant-expression ... opt)
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

§ 7.6.1
attribute-scoped-token:
  attribute-namespace : : identifier
attribute-namespace:
  identifier
attribute-argument-clause:
  ( balanced-token-seq )
balanced-token-seq:
  balanced-token_opt
  balanced-token-seq balanced-token
balanced-token:
  ( balanced-token-seq )
  [ balanced-token-seq ]
  { balanced-token-seq }
  any token other than a parenthesis, a bracket, or a brace

2 [ Note: For each individual attribute, the form of the balanced-token-seq will be specified. — end note ]

3 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 (14.5.3). 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 (2.11) or an alternative token (2.5) that satisfies the syntactic requirements of an identifier (2.10) is contained in an attribute-token, it is considered an identifier. No name lookup (3.4) 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). The use of an attribute-scoped-token is conditionally-supported, with implementation-defined behavior. [ Note: Each implementation should choose a distinctive name for the attribute-namespace in an attribute-scoped-token. — end note ]

4 Each attribute-specifier-seq is said to appertain to some entity or statement, identified by the syntactic context where it appears (Clause 6, Clause 7, Clause 8). If an attribute-specifier-seq that appertains to some entity or statement contains an attribute that is not allowed to apply to that entity or statement, the program is ill-formed. If an attribute-specifier-seq appertains to a friend declaration (11.3), that declaration shall be a definition. No attribute-specifier-seq shall appertain to an explicit instantiation (14.7.2).

5 For an attribute-token not specified in this International Standard, the behavior is implementation-defined.

6 Two consecutive left square bracket tokens shall appear only when introducing an attribute-specifier. [ 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:

        int p[10];
        void f() {
            int x = 42, y[5];
            int(p[x] { return x; }()); // error: invalid attribute on a nested
            // declarator-id and not a function-style cast of
            // an element of p.
            y[] { return 2; }() = 2; // error even though attributes are not allowed
            // in this context.
        }
    — end example ]

7.6.2 Alignment specifier [dcl.align]

1 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, an exception-declaration (15.3), or a variable declared with the register storage class specifier. An alignment-specifier may also be applied to the declaration or definition of a class (in an elaborated-type-specifier (7.1.6.3) or class-head (Clause 9), respectively) and to the declaration

§ 7.6.2 186
or definition of an enumeration (in an opaque-enum-declaration or enum-head, respectively (7.2)). An alignment-specifier with an ellipsis is a pack expansion (14.5.3).

2 When the alignment-specifier is of the form alignas( constant-expression ):
   (2.1) — the constant-expression shall be an integral constant expression
   (2.2) — if the constant expression evaluates to a fundamental alignment, the alignment requirement of the declared entity shall be the specified fundamental alignment
   (2.3) — if the constant expression evaluates to an extended alignment and the implementation supports that alignment in the context of the declaration, the alignment of the declared entity shall be that alignment
   (2.4) — if the constant expression evaluates to an extended alignment and the implementation does not support that alignment in the context of the declaration, the program is ill-formed
   (2.5) — if the constant expression evaluates to zero, the alignment specifier shall have no effect
   (2.6) — otherwise, the program is ill-formed.

3 When the alignment-specifier is of the form alignas( type-id ), it shall have the same effect as alignas( alignof( type-id ) ) (5.3.6).

4 When multiple alignment-specifiers are specified for an entity, the alignment requirement shall be set to the strictest specified alignment.

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
extern S* p; // diagnostic required
```
— end example ]

7 [ Example: An aligned buffer with an alignment requirement of A and holding N elements of type T other than char, signed char, or unsigned char 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]

7.6.3 Noreturn attribute [dcl.attr.noreturn]

1 The attribute-token `noreturn` specifies that a function does not return. It shall appear at most once in each `attribute-list` and no `attribute-argument-clause` shall be present. The attribute may be applied to the `declarator-id` in a function declaration. The first declaration of a function shall specify the `noreturn` attribute if any declaration of that function specifies the `noreturn` attribute. If a function is declared with the `noreturn` attribute in one translation unit and the same function is declared without the `noreturn` attribute in another translation unit, the program is ill-formed; no diagnostic required.

2 If a function `f` is called where `f` was previously declared with the `noreturn` attribute and `f` eventually returns, the behavior is undefined. [Note: The function may terminate by throwing an exception. —end note] [Note: Implementations are encouraged to issue a warning if a function marked `[[noreturn]]` might return. —end note]

3 [Example:

```c
[[ 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]

7.6.4 Carries dependency attribute [dcl.attr.depend]

1 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 (1.10) each lvalue-to-rvalue conversion (4.1) of that object. The attribute may also be applied to the `declarator-id` of a function declaration, in which case it specifies that the return value, if any, carries a dependency to the evaluation of the function call expression.

2 The first declaration of a function shall specify the `carries_dependency` attribute for its `declarator-id` if any declaration of the function specifies the `carries_dependency` attribute. Furthermore, the first declaration of a function shall specify the `carries_dependency` attribute for a parameter if any declaration of that function specifies the `carries_dependency` attribute for that parameter. If a function or one of its parameters is declared with the `carries_dependency` attribute in its first declaration in one translation unit and the same function or one of its parameters is declared without the `carries_dependency` attribute in its first declaration in another translation unit, the program is ill-formed; no diagnostic required.
[Note: The carries_dependency attribute does not change the meaning of the program, but may result in generation of more efficient code. — end note]

Example:

```c
/* Translation unit A. */

struct foo { int* a; int* b; }
std::atomic<struct foo *> foo_head[10];
int foo_array[10][10];

[[carries_dependency]] struct foo* f(int i) {
    return foo_head[i].load(memory_order_consume);
}

int g(int* x, int* y [[carries_dependency]]) {
    return kill_dependency(foo_array[*x][*y]);
}

/* Translation unit B. */

[[carries_dependency]] struct foo* f(int i);
int g(int* x, int* y [[carries_dependency]]);

int c = 3;

void h(int i) {
    struct foo* p;
    p = f(i);
    do_something_with(g(&c, p->a));
    do_something_with(g(p->a, &c));
}
```

The carries_dependency attribute on function f means that the return value carries a dependency out of f, so that the implementation need not constrain ordering upon return from f. Implementations of f and its caller may choose to preserve dependencies instead of emitting hardware memory ordering instructions (a.k.a. fences).

Function g’s second parameter has a carries_dependency attribute, but its first parameter does not. Therefore, function h’s first call to g carries a dependency into g, but its second call does not. The implementation might need to insert a fence prior to the second call to g.

— end example

7.6.5 Deprecated attribute

1 The attribute-token deprecated can be used to mark names and entities whose use is still allowed, but is discouraged for some reason. [Note: in particular, deprecated is appropriate for names and entities that are deemed obsolescent or unsafe. — end note] It shall appear at most once in each attribute-list. An attribute-argument-clause may be present and, if present, it shall have the form:

( string-literal )

[Note: the string-literal in the attribute-argument-clause could be used to explain the rationale for deprecation and/or to suggest a replacing entity. — end note]
The attribute may be applied to the declaration of a class, a typedef-name, a variable, a non-static data member, a function, a namespace, an enumeration, an enumerator, or a template specialization.

A name or entity declared without the deprecated attribute can later be re-declared 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]
8 Declarators

A declarator declares a single variable, function, or type, within a declaration. The *init-declarator-list* appearing in a declaration is a comma-separated sequence of declarators, each of which can have an initializer.

```
init-declarator-list:
  init-declarator
  init-declarator-list , init-declarator

init-declarator:
  declarator initializer_opt
```

The three components of a *simple-declaration* are the attributes (7.6), the specifiers (*decl-specifier-seq*; 7.1) and the declarators (*init-declarator-list*). The specifiers indicate the type, storage class or other properties of the entities being declared. The declarators specify the names of these entities and (optionally) modify the type of the specifiers with operators such as `*` (pointer to) and `()` (function returning). Initial values can also be specified in a declarator; initializers are discussed in 8.5 and 12.6.

Each *init-declarator* in a declaration is analyzed separately as if it was in a declaration by itself.\(^{99}\)

Declarators have the syntax

```
declarator:
  ptr-declarator
  noptr-declarator parameters-and-qualifiers trailing-return-type

ptr-declarator:
  noptr-declarator
  ptr-operator ptr-declarator

nopr-declarator:
  declarator-id attribute-specifier-seq_opt
  nptr-declarator parameters-and-qualifiers
  nptr-declarator [ constant-expression_opt ] attribute-specifier-seq_opt
  ( ptr-declarator )

parameters-and-qualifiers:
  ( parameter-declaration-clause ) cv-qualifier-seq_opt
  ref-qualifier_opt exception-specification_opt attribute-specifier-seq_opt
```

\(^{99}\) A declaration with several declarators is usually equivalent to the corresponding sequence of declarations each with a single declarator. That is

```
T D1, D2, ... Dn;
```

is usually equivalent to

```
T D1; T D2; ... T Dn;
```

where T is a *decl-specifier-seq* and each Di is an *init-declarator*. An exception occurs when a name introduced by one of the declarators hides a type name used by the *decl-specifiers*, so that when the same *decl-specifiers* are used in a subsequent declaration, they do not have the same meaning, as in

```
struct S ... ;
S S, T; // declare two instances of struct S
```

which is not equivalent to

```
struct S ... ;
S S;
S T; // error
```

Another exception occurs when T is *auto* (7.1.6.4), for example:

```
auto i = 1, j = 2.0; // error: deduced types for i and j do not match
```

as opposed to

```
auto i = 1; // OK: i deduced to have type int
auto j = 2.0; // OK: j deduced to have type double
```
trailing-return-type:
   → trailing-type-specifier-seq abstract-declarator$_{\text{opt}}$
ptr-operator:
   * attribute-specifier-seq$_{\text{opt}}$ cv-qualifier-seq$_{\text{opt}}$
   & attribute-specifier-seq$_{\text{opt}}$
   && attribute-specifier-seq$_{\text{opt}}$
   nested-name-specifier * attribute-specifier-seq$_{\text{opt}}$ cv-qualifier-seq$_{\text{opt}}$
cv-qualifier-seq:
   cv-qualifier cv-qualifier-seq$_{\text{opt}}$
cv-qualifier:
   const
   volatile
ref-qualifier:
   &
   &&
declarator-id:
   ...$_{\text{opt}}$ id-expression

The optional attribute-specifier-seq in a trailing-return-type appertains to the indicated return type. The type-id in a trailing-return-type includes the longest possible sequence of abstract-declarators. [Note: This resolves the ambiguous binding of array and function declarators. [Example:

    auto f()->int(*)[4];  // function returning a pointer to array[4] of int
    // not function returning array[4] of pointer to int

   —end example] —end note]

8.1 Type names [dcl.name]

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$_{\text{opt}}$
abstract-declarator:
   ptr-abstract-declarator
   noptr-abstract-declarator$_{\text{opt}}$ parameters-and-qualifiers trailing-return-type
   abstract-pack-declarator
ptr-abstract-declarator:
   noptr-abstract-declarator
   ptr-operator ptr-abstract-declarator$_{\text{opt}}$
noptr-abstract-declarator:
   noptr-abstract-declarator$_{\text{opt}}$ parameters-and-qualifiers
   noptr-abstract-declarator$_{\text{opt}}$( constant-expression$_{\text{opt}}$ ) attribute-specifier-seq$_{\text{opt}}$
   ( 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$( constant-expression$_{\text{opt}}$ ) attribute-specifier-seq$_{\text{opt}}$
   ...

It is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. [Example:
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

2 A type can also be named (often more easily) by using a typedef (7.1.3).

8.2 Ambiguity resolution [dcl.ambig.res]

1 The ambiguity arising from the similarity between a function-style cast and a declaration mentioned in 6.8 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 6.8, the resolution is to consider any construct that could possibly be a declaration a declaration. [Note: A declaration can be explicitly disambiguated by a nonfunction-style cast, by an = to indicate initialization or by removing the redundant parentheses around the parameter name. —end note] [Example:

```c
struct S {
  S(int);
};

void foo(double a) {
  S w(int(a)); // function declaration
  S x(int()); // function declaration
  S y((int)a); // object declaration
  S z = int(a); // object declaration
}
—end example]
```

2 The ambiguity arising from the similarity between a function-style cast and a type-id can occur in different contexts. The ambiguity appears as a choice between a function-style cast expression and a declaration of a type. The resolution is that any construct that could possibly be a type-id in its syntactic context shall be considered a type-id.

3 [Example:

```c
#include <cstddef>
char* p;
void* operator new(std::size_t, int);
void foo() {
  const int x = 63;
  new (int(*p)) int; // new-placement
  new (int(*[x]));  // parenthesized type-id
}
```

4 For another example,

```c
template <class T>
struct S {
  T* p;
};
```

§ 8.2
S<int()> x; // type-id
S<int(1)> y;  // expression (ill-formed)

For another example,

```c
void foo() {
    sizeof(int(1)); // expression
    sizeof(int());  // type-id (ill-formed)
}
```

For another example,

```c
void foo() {
    (int(1));     // expression
    (int())1;     // type-id (ill-formed)
}
```

—end example—

Another ambiguity arises in a parameter-declaration-clause of a function declaration, or in a type-id that is the operand of a sizeof or typeid operator, when a type-name is nested in parentheses. In this case, the choice is between the declaration of a parameter of type pointer to function and the declaration of a parameter with redundant parentheses around the declarator-id. The resolution is to consider the type-name as a simple-type-specifier rather than a declarator-id. [Example:

```c
class C { }
void f(int(C)) { }  // void f(int(*fp)(C c)) { }
                    // not: void f(int C);
int g(C);

void foo() {
    f(1);           // error: cannot convert 1 to function pointer
    f(g);           // OK
}
```

For another example,

```c
class C { }
void h(int *(C[10])); // void h(int *(*_fp)(C _parm[10]));
                    // not: void h(int *C[10]);
```

—end example—

8.3 Meaning of declarators

A list of declarators appears after an optional (Clause 7) decl-specifier-seq (7.1). Each 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 (12.1, 12.3, 12.4, 13.5) and for the declaration of template specializations or partial specializations (14.7). 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 (7.3.1)) 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

\[ § 8.3 \]
in the global namespace scope. — end note] The optional attribute-specifier-seq following a declarator-id appertains to the entity that is declared.

2 A static, thread_local, extern, register, mutable, friend, inline, virtual, or typedef specifier applies directly to each declarator-id in an init-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\textsubscript{opt} decl-specifier-seq and D is a declarator. Following is a recursive procedure for determining the type specified for the contained declarator-id by such a declaration.

4 First, the decl-specifier-seq determines a type. In a declaration

T D

the decl-specifier-seq T determines the type T. [Example: in the declaration

\begin{verbatim}
int unsigned i;
\end{verbatim}

the type specifiers \texttt{int unsigned} determine the type “\texttt{unsigned int}” (7.1.6.2). — end example]

5 In a declaration attribute-specifier-seq\textsubscript{opt} T D where D is an unadorned identifier the type of this identifier is “T”.

6 In a declaration T D where D has the form

( D1 )

the type of the contained declarator-id is the same as that of the contained declarator-id in the declaration T D1

Parentheses do not alter the type of the embedded declarator-id, but they can alter the binding of complex declarators.

**8.3.1 Pointers** [dcl.ptr]

1 In a declaration T D where D has the form

\begin{verbatim}
* attribute-specifier-seq\textsubscript{opt} cv-qualifier-seq\textsubscript{opt} D1
\end{verbatim}

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 (7.6.1) appertains to the pointer and not to the object pointed to.

[Example: the declarations

\begin{verbatim}
const int ci = 10, *pc = &ci, *const cpc = pc, **ppc;
int i, *p, *const cp = &i;
\end{verbatim}

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{verbatim}
i = ci;
*cp = ci;
pc++;
pc = cpc;
\end{verbatim}

§ 8.3.1 195
pc = p;
ppc = &pc;

Examples of ill-formed operations are

```
ci = 1;    // error
ci++;     // error
*pc = 2;  // error
cp = &ci; // error
cpc++;    // error
p = pc;   // error
ppc = &p; // error
```

Each is unacceptable because it would either change the value of an object declared `const` or allow it to be changed through a `cv-unqualified` pointer later, for example:

```
*ppc = &ci;  // OK, but would make p point to ci ...
            // ... because of previous error
*p = 5;     // clobber ci
```

— end example]

See also 5.18 and 8.5.

4 [Note: Forming a pointer to reference type is ill-formed; see 8.3.2. Forming a pointer to function type is ill-formed if the function type has `cv-qualifiers` or a `ref-qualifier`; see 8.3.5. Since the address of a bit-field (9.6) cannot be taken, a pointer can never point to a bit-field. — end note]

8.3.2 References [dcl.ref]

1 In a declaration `T D` where `D` has either of the forms

   ```
   & attribute-specifier-seqopt D1
   && attribute-specifier-seqopt D1
   ```

and the type of the identifier in the declaration `T D1` is "derived-declarator-type-list `T`,` then the type of the identifier of `D` is "derived-declarator-type-list reference to `T`." The optional `attribute-specifier-seq` appertains to the reference type. `cv-qualified` references are ill-formed except when the `cv-qualifiers` are introduced through the use of a `typedef-name` (7.1.3, 14.1) or `decltype-specifier` (7.1.6.2), in which case the `cv-qualifiers` are ignored. [Example:

```
typedef int& A;
const A aref = 3; // ill-formed; lvalue reference to non-const initialized with rvalue
```

The type of `aref` is "lvalue reference to `int`," not "lvalue reference to `const int`." — end example] [Note: A reference can be thought of as a name of an object. — end note] A declarator that specifies the type "reference to `cv void`" is ill-formed.

2 A reference type that is declared using `&` is called an `lvalue reference`, and a reference type that is declared using `&&` is called an `rvalue reference`. Lvalue references and rvalue references are distinct types. Except where explicitly noted, they are semantically equivalent and commonly referred to as references.

3 [Example:

```java
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`.

§ 8.3.2
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 8.5.3.

---end example---

4 It is unspecified whether or not a reference requires storage (3.7).

5 There shall be no references to references, no arrays of references, and no pointers to references. The declaration of a reference shall contain an initializer (8.5.3) except when the declaration contains an explicit extern specifier (7.1.1), is a class member (9.2) declaration within a class definition, or is the declaration of a parameter or a return type (8.3.5); see 3.1. 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 9.6, a reference cannot be bound directly to a bit-field. —end note]

6 If a typedef-name (7.1.3, 14.1) or a decltype-specifier (7.1.6.2) 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. [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 = i;        // r5 has the type int&&
decltype(r2)& r6 = i;  // r6 has the type int&
decltype(r2)&& r7 = i;  // r7 has the type int&&
```

§ 8.3.2
8.3.3 Pointers to members

In a declaration \( T \ D \) where \( D \) has the form
\[
\text{nested-name-specifier} \ast \text{attribute-specifier-seq}\text{opt} \text{cv-qualifier-seq}\text{opt} \ D1
\]
and the \text{nested-name-specifier} denotes a class, and the type of the identifier in the declaration \( T \ D1 \) is “\text{derived-declarator-type-list} \ T \” , then the type of the identifier of \( D \) is “\text{derived-declarator-type-list cv-qualifier-seq} pointer to member of class \text{nested-name-specifier} of type \( T \)” . The optional \text{attribute-specifier-seq} (7.6.1) appertains to the pointer-to-member.

Example:
```c
struct X {
    void f(int);
    int a;
};
struct Y;
int X::* pmi = &X::a;
void (X::* pmf)(int) = &X::f;
double X::* pmd;
char Y::* pmc;
```
declares \( \text{pmi, pmf, pmd and pmc} \) to be a pointer to a member of \( X \) of type \text{int}, a pointer to a member of \( X \) of type \text{void(int)}, a pointer to a member of \( X \) of type \text{double} and a pointer to a member of \( Y \) of type \text{char} respectively. The declaration of \( \text{pmd} \) is well-formed even though \( X \) has no members of type \text{double}. Similarly, the declaration of \( \text{pmc} \) is well-formed even though \( Y \) is an incomplete type. \( \text{pmi} \) and \( \text{pmf} \) can be used like this:
```c
X obj;
// ...
obj.*pmi = 7;  // assign 7 to an integer
(obj.*pmf)(7);  // call a function member of obj
```
—end example

A pointer to member shall not point to a static member of a class (9.4), a member with reference type, or “\text{cv void}.”

[\text{Note: See also 5.3 and 5.5. 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]

8.3.4 Arrays

In a declaration \( T \ D \) where \( D \) has the form
\[
D1 [ \text{constant-expression}\text{opt} ] \text{attribute-specifier-seq}\text{opt}
\]
and the type of the identifier in the declaration \( T \ D1 \) is “\text{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 \text{auto type-specifier}, the program is ill-formed. \( T \) is called the array \text{element type}; this type shall not be a reference type, the (possibly cv-qualified) type \text{void}, a function type or an abstract class type. If the \text{constant-expression} (5.20) 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 3.9. Any type of the form “`cv-qualifier-seq` array of \( N \) \( T \)” is adjusted to “array of \( N \) \( cv-qualifier-seq \) \( T \)”, and similarly for “array of unknown bound of \( T \)”. The optional attribute-specifier-seq appertains to the array.

Example:

```c
typedef int A[5], AA[2][3];  // type is “array of 5 int”
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 \( N \) \( cv-qualifier-seq \) \( T \)” has \( cv \)-qualified type; see 3.9.3. — end note]

An array can be constructed from one of the fundamental types (except `void`), from a pointer, from a pointer to member, from a class, from an enumeration type, or from another array.

When several “array of” specifications are adjacent, a multidimensional array 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 (8.3.5). An array bound may also be omitted when the declarator is followed by an `initializer` (8.5). In this case the bound is calculated from the number of initial elements (say, \( N \)) supplied (8.5.1), and the type of the identifier of \( D \) is “array of \( N \) \( T \)” Further more, if there is a preceding declaration of the entity in the same scope in which the bound was specified, an omitted array bound is taken to be the same as in that earlier declaration, and similarly for the definition of a static data member of a class.

Example:

```c
float fa[17], *afp[17];
```
declar es an array of `float` numbers and an array of pointers to `float` numbers. For another example,

```c
static int x3d[3][5][7];
```
de clares a static three-dimensional array of integers, with rank \( 3 \times 5 \times 7 \). In complete detail, \( x3d \) is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions \( x3d, x3d[i], x3d[i][j], x3d[i][j][k] \) can reasonably appear in an expression. Finally,

```c
extern int x[10];
struct S {
    static int y[10];
};

int x[];                // OK: bound is 10
int S::*y[];             // OK: bound is 10

void f() {
    extern int x[];
    int i = sizeof(x);   // error: incomplete object type
}
```

—end example]

§ 8.3.4 199
5 [Note: conversions affecting expressions of array type are described in 4.2. Objects of array types cannot be modified, see 3.10. —end note]

6 [Note: Except where it has been declared for a class (13.5.5), the subscript operator [] is interpreted in such a way that E1[E2] is identical to *((E1)+(E2)). 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.

7 A consistent rule is followed for multidimensional arrays. If E is an n-dimensional array of rank i × j × ... × k, then E appearing in an expression that is subject to the array-to-pointer conversion (4.2) is converted to a pointer to an (n − 1)-dimensional array with rank j × ... × k. If the * operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to (n − 1)-dimensional array, which itself is immediately converted into a pointer.

8 [Example: consider

    int x[3][5];

Here x is a 3 × 5 array of integers. When x appears in an expression, it is converted to a pointer to (the first of three) five-membered arrays of integers. In the expression x[1] which is equivalent to *((x+i), x is first converted to a pointer as described; then x+i is converted to the type of x, which involves multiplying i by the length of the object to which the pointer points, namely five integer objects. The results are added and indirection applied to yield an array (of five integers), which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer. —end example] —end note]

9 [Note: It follows from all this that arrays in C++ are stored row-wise (last subscript varies fastest) and that 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]

8.3.5 Functions

In a declaration T D where D has the form

D1 ( parameter-declaration-clause ) cv-qualifier-seq_opt
    ref-qualifier_opt exception-specification_opt attribute-specifier-seq_opt

and the type of the contained declarator-id in the declaration T D1 is “derived-declarator-type-list T”, the type of the declarator-id in D is “derived-declarator-type-list function of (parameter-declaration-clause ) cv-qualifier-seq_opt ref-qualifier_opt returning T”. The optional attribute-specifier-seq appertains to the function type.

In a declaration T D where D has the form

D1 ( parameter-declaration-clause ) cv-qualifier-seq_opt
    ref-qualifier_opt exception-specification_opt attribute-specifier-seq_opt trailing-return-type

and the type of the contained declarator-id in the declaration T D1 is “derived-declarator-type-list T”, T shall be the single type-specifier auto. The type of the declarator-id in D is “derived-declarator-type-list function of (parameter-declaration-clause ) cv-qualifier-seq_opt ref-qualifier_opt returning trailing-return-type”. The optional attribute-specifier-seq appertains to the function type.

A type of either form is a function type."^100

parameter-declaration-clause:
    parameter-declaration-list_opt...
    parameter-declaration

parameter-declaration-list:
    parameter-declaration
    parameter-declaration-list , parameter-declaration

---

100) As indicated by syntax, cv-qualifiers are a significant component in function return types.

§ 8.3.5
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 5.2.2. — 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 (14.5.3), 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 <cstdarg> contains a mechanism for accessing arguments passed using the ellipsis (see 5.2.2 and 18.10). — end note]

5 A single name can be used for several different functions in a single scope; this is function overloading (Clause 13). All declarations for a function shall agree exactly in both the return type and the parameter-type-list. 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 “function returning T” is adjusted to be “pointer to T” or “pointer to function returning T,” respectively. After producing the list of parameter types, any top-level cv-qualifiers modifying a parameter type are deleted when forming the function type. The resulting list of transformed parameter types and the presence or absence of the ellipsis or a function parameter pack is the function’s parameter-type-list. [Note: This transformation does not affect the types of the parameters. For example, int(*)(const int p, decltype(p)*) and int(*)(int, const int*) are identical types. — end note]

6 A function type with a cv-qualifier-seq or a ref-qualifier (including a type named by typedef-name (7.1.3, 14.1)) shall appear only as:

(6.1) — the function type for a non-static member function,
(6.2) — the function type to which a pointer to member refers,
(6.3) — the top-level function type of a function typedef declaration or alias-declaration,
(6.4) — the type-id in the default argument of a type-parameter (14.1), or
(6.5) — the type-id of a template-argument for a type-parameter (14.3.1).

[Example:

typedef int FIC(int) const;
FIC f; // ill-formed: does not declare a member function

§ 8.3.5
struct S {
    FIC f;          // OK
};
FIC S::*pm = &S::f; // OK

— end example ]

7 The effect of a \textit{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. \textit{[Note:} a function type that has a \textit{cv-qualifier-seq} is not a cv-qualified type; there are no cv-qualified function types. \textit{— end note]} \textit{[Example:}

typedef void F();
struct S {
    const F f;       // OK: equivalent to: void f();
};

— end example]

8 The return type, the parameter-type-list, the \textit{ref-qualifier}, and the \textit{cv-qualifier-seq}, but not the default arguments (8.3.6) or the exception specification (15.4), are part of the function type. \textit{[Note:} Function types are checked during the assignments and initializations of pointers to functions, references to functions, and pointers to member functions. \textit{— end note]}

[Example: the declaration

\begin{verbatim}
int fseek(FILE*, long, int);
\end{verbatim}

declares a function taking three arguments of the specified types, and returning \texttt{int} (7.1.6). — end example]

9 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.

10 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 (8.4.3).

11 A typedef of function type may be used to declare a function but shall not be used to define a function (8.4).

[Example:]

typedef void F();
F fv;          // OK: equivalent to void fv();
F fv {}        // ill-formed
void fv() {}   // OK: definition of fv

— end example]

12 An identifier can optionally be provided as a parameter name; if present in a function definition (8.4), it names a parameter. \textit{[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 (3.3.4). \textit{— end note]}

[Example: the declaration

\begin{verbatim}
int i,
    *pi,
    f(),
    *fpi(int),
    (*fpi)(const char*, const char*),
    (*fpi(int))(int);
\end{verbatim}

\S 8.3.5
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

15 A non-template function is a function that is not a function template specialization. [Note: A function template is not a function. — end note]

16 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 (14.5.3). When it is part of a parameter-declaration-clause, the parameter pack is a function parameter pack (14.5.3). [Note: Otherwise, the parameter-declaration is part of a template-parameter-list and the parameter pack is a template parameter pack; see 14.1. — end note] A function parameter pack is a pack expansion (14.5.3). [Example:

```c
template<typename... T> void f(T (*...t)(int, int));
```

```c
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. 101

101 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).

§ 8.3.5 203
8.3.6 Default arguments

1 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.

2 [Example: the declaration
   
   ```
   void point(int = 3, int = 4);
   ```

decides a function that can be called with zero, one, or two arguments of type int. It can be called in any of these ways:

   ```
   point(1,2); point(1); point();
   ```

   The last two calls are equivalent to `point(1,4)` and `point(3,4)`, respectively. — end example]

3 A default argument shall be specified only in the parameter-declaration-clause of a function declaration or lambda-declarator or in a template-parameter (14.1); 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.\(^\text{102}\)

4 For non-template functions, default arguments can be added in later declarations of a function in the same scope. Declarations in different scopes have completely distinct sets of default arguments. That is, declarations in inner scopes do not acquire default arguments from declarations in outer scopes, and vice versa. In a given function declaration, each parameter subsequent to a parameter with a default argument shall have a default argument supplied in this or a previous declaration or shall be a function parameter pack. A default argument shall not be redefined by a later declaration (not even to the same value). [Example:

   ```
   void g(int = 0, ...);
   void f(int, int);
   void f(int, int = 7);
   void h() {
      f(3);
      void f(int = 1, int);
   }  // OK, calls f(3, 7)
   void f(int, int = 7);  // 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 3.2. 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.

\(^{102}\) This means that default arguments cannot appear, for example, in declarations of pointers to functions, references to functions, or typedef declarations.
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 (8.5). 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 14.7.1. [Example: in the following code, \( g \) will be called with the value \( f(2) \):

```c
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 3.4.1. Access checking applies to names in default arguments as described in Clause 11. — end note]

Except for member functions of class templates, the default arguments in a member function definition that appears outside of the class definition are added to the set of default arguments provided by the member function declaration in the class definition; the program is ill-formed if a default constructor (12.1), copy or move constructor, or copy or move assignment operator (12.8) 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:

```c
class C {
    void f(int i = 3);
    void g(int i, int j = 99);
};

void C::f(int i = 3) {  // error: default argument already specified in class scope
}  // in this translation unit,
void C::g(int i = 88, int j) {  // C::g can be called with no argument
}  // C::g can be called with no argument
```

—end example]

Local variables shall not be used in a default argument. [Example:

```c
void f() {
    int i;
    extern void g(int x = i);  //error
    // ...
}
```

—end example]

The keyword \texttt{this} shall not be used in a default argument of a member function. [Example:

```c
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. The order of evaluation of function arguments is unspecified. Consequently, parameters of a function shall not be used in a default argument, even if they are not evaluated. 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));  // error, parameter a used
// in default argument
```

— end example] Similarly, a non-static member shall not be used in a default argument, even if it is not evaluated, unless it appears as the *id-expression* of a class member access expression (5.2.5) or unless it is used to form a pointer to member (5.3.1). [Example: the declaration of `X::mem1()` in the following example is ill-formed because no object is supplied for the non-static member `X::a` used as an initializer.

```cpp
int b;
class X {
    int a;
    int mem1(int i = a);  // error: non-static member a
    // used as default argument
    int mem2(int i = b);  // OK; use X::b
    static int b;
};
```

The declaration of `X::mem2()` is meaningful, however, since no object is needed to access the static member `X::b`. Classes, objects, and members are described in Clause 9. — end example] A default argument is not part of the type of a function. [Example:

```cpp
int f(int = 0);
void h() {
    int j = f(1);
    int k = f();  // OK, means f(0)
}
int (*p1)(int) = &f;
int (*p2)() = &f;  // error: type mismatch
```

— end example] When a declaration of a function is introduced by way of a *using-declaration* (7.3.3), 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 (10.3) 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:

```cpp
struct A {
    virtual void f(int a = 7);
};
struct B : public A {
    void f(int a);
}
```
};

void m() {
    B* pb = new B;
    A* pa = pb;
    pa->f();       // OK, calls pa->B::f(7)
    pb->f();       // error: wrong number of arguments for B::f()
}

— end example —

8.4 Function definitions

8.4.1 In general

1 Function definitions have the form

```plaintext
function-definition:
    attribute-specifier-seq_opt decl-specifier-seq_opt declarator virt-specifier-seq_opt function-body

function-body:
    ctor-initializer_opt compound-statement
    function-try-block
    = default ;
    = delete ;
```

Any informal reference to the body of a function should be interpreted as a reference to the non-terminal function-body. The optional attribute-specifier-seq in a function-definition appertains to the function. A virt-specifier-seq can be part of a function-definition only if it is a member-declaration (9.2).

2 The declarator in a function-definition shall have the form

```plaintext
D1 ( parameter-declaration-clause ) cv-qualifier-seq_opt
    ref-qualifier_opt exception-specification_opt attribute-specifier-seq_opt trailing-return-type_opt
```

as described in 8.3.5. A function shall be defined only in namespace or class scope.

3 [ Example: a simple example of a complete function definition is

```plaintext
int max(int a, int b, int c) {
    int m = (a > b) ? a : b;
    return (m > c) ? m : c;
}
```

Here int is the decl-specifier-seq; max(int a, int b, int c) is the declarator; { /* ... */ } is the function-body. — end example ]

4 A ctor-initializer is used only in a constructor; see 12.1 and 12.6.

5 [ Note: A cv-qualifier-seq affects the type of this in the body of a member function; see 8.3.2. — end note ]

6 [ Note: Unused parameters need not be named. For example,

```plaintext
void print(int a, int) {
    std::printf("a = %d\n",a);
}
```

— end note ]

7 In the function-body, a function-local predefined variable denotes a block-scope object of static storage duration that is implicitly defined (see 3.3.3).

8 The function-local predefined variable __func__ is defined as if a definition of the form

```plaintext
static const char __func__[] = "function-name";
```

§ 8.4.1
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.\footnote{Implementations are permitted to provide additional predefined variables with names that are reserved to the implementation (2.10). If a predefined variable is not odr-used (3.2), its string value need not be present in the program image.}

\[\text{Example:}\]

\begin{verbatim}
struct S {
    S() : s(__func__) { } // OK
    const char* s;
};
void f(const char* s = __func__); // error: __func is undeclared

— end example —
\end{verbatim}

8.4.2 Explicitly-defaulted functions \[dcl.fct.def.default\]

A function definition of the form:

\begin{verbatim}
attribute-specifier-seq_{opt} decl-specifier-seq_{opt} declarator virt-specifier-seq_{opt} = default ;
\end{verbatim}

is called an explicitly-defaulted definition. A function that is explicitly defaulted shall

\[(1.1)\] — be a special member function,

\[(1.2)\] — have the same declared function type (except for possibly differing ref-qualifiers and except that in the case of a copy constructor or copy assignment operator, the parameter type may be “reference to non-\(\text{const}\) \(T\)”, where \(T\) is the name of the member function’s class) as if it had been implicitly declared, and

\[(1.3)\] — not have default arguments.

\[2\] 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,

\[(2.1)\] — it is implicitly considered to be constexpr if the implicit declaration would be, and,

\[(2.2)\] — it has the same exception specification as if it had been implicitly declared (15.4).\n
\[3\] If a function that is explicitly defaulted is declared with an exception-specification that is not compatible (15.4) with the exception specification on the implicit declaration, then

\[(3.1)\] — if the function is explicitly defaulted on its first declaration, it is defined as deleted;

\[(3.2)\] — otherwise, the program is ill-formed.

\[4\] \[Example:\]

\begin{verbatim}
struct S {
    constexpr S() = default; // ill-formed: implicit S() is not constexpr
    S(int a = 0) = default; // ill-formed: default argument
    void operator=(const S&) = default; // ill-formed: non-matching return type
    ~S() throw(int) = default; // deleted: exception specification does not match
private:
    int i;
    S(S&); // OK: private copy constructor
};
S::S(S&); // OK: defines copy constructor
\end{verbatim}
Explicitly-defaulted functions and implicitly-declared functions are collectively called *defaulted* functions, and the implementation shall provide implicit definitions for them (12.1 12.4, 12.8), 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
struct trivial {
    trivial() = default;
    trivial(const trivial&) = default;
    trivial(trivial&&) = default;
    trivial& operator=(const trivial&) = default;
    trivial& operator=(trivial&&) = default;
    ~trivial() = default;
};
struct nontrivial1 {
    nontrivial1();
};
nontrivial1::nontrivial1() = default;  // not first declaration
```
—end example]

8.4.3 Deleted definitions [dcl.fct.def.delete]

1 A function definition of the form:

```
attribute-specifier-seq_opt decl-specifier-seq_opt declarator virt-specifier-seq_opt = delete ;
```

is called a *deleted definition*. A function with a deleted definition is also called a *deleted function*.

2 A program that refers to a deleted function implicitly or explicitly, other than to declare it, is ill-formed. [Note: This includes calling the function implicitly or explicitly and forming a pointer or pointer-to-member to the function. It applies even for references in expressions that are not potentially-evaluated. If a function is overloaded, it is referenced only if the function is selected by overload resolution. — end note]

3 [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);
};
```
—end example]

[Example: One can prevent use of a class in certain *new-expressions* by using deleted definitions of a user-declared *operator new* for that class.

```cpp
struct sometype {
    void* operator new(std::size_t) = delete;
    void* operator new[](std::size_t) = delete;
};
```

```
sometype* p = new sometype;  // error, deleted class operator new
```
sometype* q = new sometype[3]; // error, deleted class operator new[]

— end example]

[Example: One can make a class uncopyable, i.e. move-only, by using deleted definitions of the copy constructor and copy assignment operator, and then providing defaulted definitions of the move constructor and move assignment operator.

```cpp
struct moveonly {
    moveonly() = default;
    moveonly(const moveonly&) = delete;
    moveonly(moveonly&&) = default;
    moveonly& operator=(const moveonly&) = delete;
    moveonly& operator=(moveonly&&) = default;
    ~moveonly() = default;
};
moveonly* p;
moveonly q(*p); // error, deleted copy constructor
```

— end example]

A deleted function is implicitly inline. [Note: The one-definition rule (3.2) 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. [Example:

```cpp
struct sometype {
    sometype();
};
sometype::sometype() = delete; // ill-formed; not first declaration
```

— end example]

8.5 Initializers [dcl.init]

1 A declarator can specify an initial value for the identifier being declared. The identifier designates a variable being initialized. The process of initialization described in the remainder of 8.5 applies also to initializations specified by other syntactic contexts, such as the initialization of function parameters with argument expressions (5.2.2) or the initialization of return values (6.6.3).

```cpp
initializer:
    brace-or-equal-initializer
    ( expression-list )
brace-or-equal-initializer:
    = initializer-clause
braced-init-list
initializer-clause:
    assignment-expression
braced-init-list
initializer-list:
    initializer-clause ...opt
initializer-list , initializer-clause ...opt
braced-init-list:
    { initializer-list , opt }

```

2 Except for objects declared with the constexpr specifier, for which see 7.1.5, 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:
int f(int);
int a = 2;
int b = f(a);
int c(b);

— end example ]

3 [ Note: Default arguments are more restricted; see 8.3.6.

4 The order of initialization of variables with static storage duration is described in 3.6 and 6.7. — end note ]

5 A declaration of a block-scope variable with external or internal linkage that has an initializer is ill-formed.

6 To zero-initialize an object or reference of type \( T \) means:

\begin{enumerate}
\item If \( T \) is a scalar type (3.9), the object is initialized to the value obtained by converting the integer literal 0 (zero) to \( T \);\(^{104}\)
\item If \( T \) is a (possibly cv-qualified) non-union class type, each non-static data member and each base-class subobject is zero-initialized and padding is initialized to zero bits;
\item If \( T \) is a (possibly cv-qualified) union type, the object’s first non-static named data member is zero-initialized and padding is initialized to zero bits;
\item If \( T \) is an array type, each element is zero-initialized;
\item If \( T \) is a reference type, no initialization is performed.
\end{enumerate}

7 To default-initialize an object of type \( T \) means:

\begin{enumerate}
\item If \( T \) is a (possibly cv-qualified) class type (Clause 9), constructors are considered. The applicable constructors are enumerated (13.3.1.3), and the best one for the initializer \( () \) is chosen through overload resolution (13.3). The constructor thus selected is called, with an empty argument list, to initialize the object.
\item If \( T \) is an array type, each element is default-initialized.
\item Otherwise, no initialization is performed.
\end{enumerate}

If a program calls for the default initialization of an object of a const-qualified type \( T \), \( T \) shall be a class type with a user-provided default constructor.

8 To value-initialize an object of type \( T \) means:

\begin{enumerate}
\item If \( T \) is a (possibly cv-qualified) class type (Clause 9) with either no default constructor (12.1) or a default constructor that is user-provided or deleted, then the object is default-initialized;
\item 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;
\item If \( T \) is an array type, then each element is value-initialized;
\item Otherwise, the object is zero-initialized.
\end{enumerate}

An object that is value-initialized is deemed to be constructed and thus subject to provisions of this International Standard applying to “constructed” objects, objects “for which the constructor has completed,” etc., even if no constructor is invoked for the object’s initialization.

9 A program that calls for default-initialization or value-initialization of an entity of reference type is ill-formed.

\(^{104}\) As specified in 4.10, converting an integer literal whose value is 0 to a pointer type results in a null pointer value.
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.

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 (5.18). [Note: Objects with static or thread storage duration are zero-initialized, see 3.6.2. —end note]

13 An initializer for a static member is in the scope of the member’s class. [Example:}

```c
int a;

struct X {
    static int a;
    static int b;
};
```

§ 8.5
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 (6.4), as well as in argument passing, function return, throwing an exception (15.1), handling an exception (15.3), and aggregate member initialization (8.5.1), is called copy-initialization. [ Note: Copy-initialization may invoke a move (12.8). — end note ]

16 The initialization that occurs in the forms

T x(a);
T x{a};

as well as in new expressions (5.3.4), static_cast expressions (5.2.9), functional notation type conversions (5.2.3), mem-initializers (12.6.2), 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.

(17.1) — If the initializer is a (non-parenthesized) braced-init-list, the object or reference is list-initialized (8.5.4).

(17.2) — If the destination type is a reference type, see 8.5.3.

(17.3) — 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 8.5.2.

(17.4) — If the initializer is (), the object is value-initialized.

(17.5) — Otherwise, if the destination type is an array, the program is ill-formed.

(17.6) — If the destination type is a (possibly cv-qualified) class type:

(17.6.1) — If the initialization is direct-initialization, or if it is copy-initialization where the cv-unqualified version of the source type is the same class as, or a derived class of, the class of the destination, constructors are considered. The applicable constructors are enumerated (13.3.1.3), and the best one is chosen through overload resolution (13.3). 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.

(17.6.2) — 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 13.3.1.4, and the best one is chosen through overload resolution (13.3). 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 initializes a temporary of the cv-unqualified version of the destination type. The temporary is a prvalue. The result of the call (which is the temporary for the constructor case) is then used to direct-initialize, according to the rules above, the object that is the destination of the copy-initialization. In certain cases, an implementation is permitted to eliminate the copying inherent in this direct-initialization by constructing the intermediate result directly into the object being initialized; see 12.2, 12.8.
Otherwise, if the source type is a (possibly cv-qualified) class type, conversion functions are considered. The applicable conversion functions are enumerated (13.3.1.5), and the best one is chosen through overload resolution (13.3). The user-defined conversion so selected is called to convert the initializer expression into the object being initialized. If the conversion cannot be done or is ambiguous, the initialization is ill-formed.

Otherwise, the initial value of the object being initialized is the (possibly converted) value of the initializer expression. Standard conversions (Clause 4) will be used, if necessary, to convert the initializer expression to the cv-unqualified version of the destination type; no user-defined conversions are considered. If the conversion cannot be done, the initialization is ill-formed. When initializing a bit-field with a value that it cannot represent, the resulting value of the bit-field is implementation-defined.

\[\text{Note: An expression of type "cv1 T" can initialize an object of type "cv2 T" independently of the cv-qualifiers cv1 and cv2.}\]

```
int a;
const int b = a;
int c = b;
```

— end note

18 An initialiser-clause followed by an ellipsis is a pack expansion (14.5.3).

8.5.1 Aggregates

An aggregate is an array or a class (Clause 9) with no user-provided constructors (12.1), no private or protected non-static data members (Clause 11), no base classes (Clause 10), and no virtual functions (10.3).

When an aggregate is initialized by an initialiser list, as specified in 8.5.4, the elements of the initialiser list are taken as initializers for the members of the aggregate, in increasing subscript or member order. Each member is copy-initialized from the corresponding initialiser-clause. If the initialiser-clause is an expression and a narrowing conversion (8.5.4) is required to convert the expression, the program is ill-formed. \[\text{Note: If an initialiser-clause is itself an initialiser list, the member is list-initialized, which will result in a recursive application of the rules in this section if the member is an aggregate. — end note}\]

```
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. — end example

An aggregate that is a class can also be initialized with a single expression not enclosed in braces, as described in 8.5.

An array of unknown size initialized with a brace-enclosed initialiser-list containing n initialiser-clauses, where n shall be greater than zero, is defined as having n elements (8.3.4). \[\text{Example:}\]

```
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 initialiser list {} shall not be used as the initialiser-clause for an array of unknown bound.¹⁰⁵

¹⁰⁵) The syntax provides for empty initialiser-lists, but nonetheless C++ does not have zero length arrays.
Static data members and anonymous bit-fields are not considered members of the class for purposes of aggregate initialization. [Example:

```c
struct A {
    int i;
    static int s;
    int j;
    int :17;
    int k;
} a = { 1, 2, 3 };
```

Here, the second initializer 2 initializes `a.j` and not the static data member `A::s`, and the third initializer 3 initializes `a.k` and not the anonymous bit-field before it. — end example]

An *initializer-list* is ill-formed if the number of *initializer-clauses* exceeds the number of members or elements to initialize. [Example:

```c
char cv[4] = { 'a', 's', 'd', 'f', 0 };  // error
```

is ill-formed. — end example]

If there are fewer *initializer-clauses* in the list than there are members in the aggregate, then each member not explicitly initialized shall be initialized from its *brace-or-equal-initializer* or, if there is no *brace-or-equal-initializer*, from an empty initializer list (8.5.4). [Example:

```c
struct S { int a; const char* b; int c; int d = b[a]; };
S ss = { 1, "asdf" };
```

initializes `ss.a` with 1, `ss.b` with "asdf", `ss.c` with the value of an expression of the form `int{}` (that is, 0), and `ss.d` with the value of `ss.b[ss.a]` (that is, 's'), and in

```c
struct X { int i, j, k = 42; };
X a[] = { 1, 2, 3, 4, 5, 6 };
X b[2] = { { 1, 2, 3 }, { 4, 5, 6 } };
```

`a` and `b` have the same value — end example]

If a reference member is initialized from its *brace-or-equal-initializer* and a potentially-evaluated subexpression thereof is an aggregate initialization that would use that *brace-or-equal-initializer*, the program is ill-formed. [Example:

```c
struct A;
extern A a;
struct A {
    const A& a1 { A{a,a} };  // OK
    const A& a2 { A{} };    // error
};
A a(a,a);        // OK
```

— end example]

If an aggregate class `C` contains a subaggregate member `m` that has no members for purposes of aggregate initialization, the *initializer-clause* for `m` shall not be omitted from an *initializer-list* for an object of type `C` unless the *initializer-clauses* for all members of `C` following `m` are also omitted. [Example:

```c
struct S { } s;
struct A {
    S s1;
    int i1;
```
S s2;
int i2;
S s3;
int i3;
} a = {
    {}, // Required initialization
    0,
    s, // Required initialization
    0
}; // Initialization not required for A::s3 because A::i3 is also not initialized

— end example

10 If an incomplete or empty initializer-list leaves a member of reference type uninitialized, the program is ill-formed.

11 When initializing a multi-dimensional array, the initializer-clauses initialize the elements with the last (right-most) index of the array varying the fastest (8.3.4). [Example:

```c
int x[2][2] = { 3, 1, 4, 2 };
```

initializes \( x[0][0] \) to 3, \( x[0][1] \) to 1, \( x[1][0] \) to 4, and \( x[1][1] \) to 2. On the other hand,

```c
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of \( y \) (regarded as a two-dimensional array) and leaves the rest zero. — end example

12 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 members of a subaggregate; it is erroneous for there to be more initializer-clauses than members. 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 members of the subaggregate; any remaining initializer-clauses are left to initialize the next member of the aggregate of which the current subaggregate is a member. [Example:

```c
float y[4][3] = {
    1, 3, 5,
    2, 4, 6,
    3, 5, 7
};
```

is a completely-braced initialization: 1, 3, and 5 initialize the first row of the array \( y[0] \), namely \( y[0][0] \), \( y[0][1] \), and \( y[0][2] \). Likewise the next two lines initialize \( y[1] \) and \( y[2] \). The initializer ends early and therefore \( y[3] \)'s elements are initialized as if explicitly initialized with an expression of the form \( \text{float()} \), that is, are initialized with 0.0. In the following example, braces in the initializer-list are elided; however the initializer-list has the same effect as the completely-braced initializer-list of the above example,

```c
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for \( y \) begins with a left brace, but the one for \( y[0] \) does not, therefore three elements from the list are used. Likewise the next three are taken successively for \( y[1] \) and \( y[2] \). — end example

13 All implicit type conversions (Clause 4) are considered when initializing the aggregate member with an assignment-expression. If the assignment-expression can initialize a member, the member is initialized.
Otherwise, if the member is itself a subaggregate, brace elision is assumed and the *assignment-expression* is considered for the initialization of the first member of the subaggregate.  [Note: As specified above, brace elision cannot apply to subaggregates with no members for purposes of aggregate initialization; an *initializer-clause* for the entire subobject is required. —end note]

*Example:*

```c
struct A {
    int i;
    operator int();
};
struct B {
    A a1, a2;
    int z;
};
A a;
B b = { 4, a, a };
```

Braces are elided around the *initializer-clause* for `b.a1.i`. `b.a1.i` is initialized with 4, `b.a2` is initialized with `a`, `b.z` is initialized with whatever `a.operator int()` returns. —end example]

[Note: An aggregate array or an aggregate class may contain members of a class type with a user-provided constructor (12.1). Initialization of these aggregate objects is described in 12.6.1. —end note]

[Note: Whether the initialization of aggregates with static storage duration is static or dynamic is specified in 3.6.2 and 6.7. —end note]

16 When a union is initialized with a brace-enclosed initializer, the braces shall only contain an *initializer-clause* for the first non-static data member of the union. [Example:

```c
union u { int a; const char* b; }; 
 u a = { 1 }; 
 u b = a; 
 u c = 1; // error 
 u d = { 0, "asdf" }; // error 
 u e = { "asdf" }; // error 
```

—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]

### 8.5.2 Character arrays [dcl.init.string]

1 An array of narrow character type (3.9.1), `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 (2.13.5). Successive characters of the value of the string literal initialize the elements of the array. [Example:

```c
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a *string-literal*. Note that because `\n` is a single character and because a trailing `\0` is appended, `sizeof(msg)` is 25. —end example]

2 There shall not be more initializers than there are array elements. [Example:

```c
char cv[4] = "asdf"; // error
```

is ill-formed since there is no space for the implied trailing `\0`. —end example]
3. If there are fewer initializers than there are array elements, each element not explicitly initialized shall be zero-initialized (8.5).

8.5.3 References [dcl.init.ref]

1. A variable declared to be a T& or T&&, that is, “reference to type T” (8.3.2), shall be initialized by an object, or function, of type T or by an object that can be converted into a T. [Example:

```c
int g(int);
void f() {
    int i;
    int& r = i; // r refers to i
    r = 1; // the value of i becomes 1
    int* p = &r; // p points to i
    int& rr = r; // rr refers to what r refers to, that is, to i
    int (&rg)(int) = g; // rg refers to the function g
    rg(i); // calls function g
    int a[3];
    int (&ra)[3] = a; // ra refers to the array a
    ra[1] = i; // modifies a[1]
}
```

—end example]

2. A reference cannot be changed to refer to another object after initialization. Note that initialization of a reference is treated very differently from assignment to it. Argument passing (5.2.2) and function value return (6.6.3) are initializations.

3. The initializer can be omitted for a reference only in a parameter declaration (8.3.5), in the declaration of a function return type, in the declaration of a class member within its class definition (9.2), and where the extern specifier is explicitly used. [Example:

```c
int& r1; // error: initializer missing
extern int& r2; // OK
```

—end example]

4. Given types “cv1 T1” and “cv2 T2,” “cv1 T1” is reference-related to “cv2 T2” if T1 is the same type as T2, or T1 is a base class of T2. “cv1 T1” is reference-compatible with “cv2 T2” if T1 is reference-related to T2 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 11) or ambiguous (10.2) 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”\(^{106}\) (this conversion is selected by enumerating the applicable conversion functions (13.3.1.6) and choosing the best one through overload resolution (13.3)), then the reference is bound to the initializer expression lvalue in the first case and to the lvalue result of the conversion in the second case (or, in either case, to the appropriate base class subobject of the object). [Note: The usual lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer

---

\(^{106}\) This requires a conversion function (12.3.2) returning a reference type.
(4.3) standard conversions are not needed, and therefore are suppressed, when such direct bindings to lvalues are done. — end note]

[Example:

```c
double d = 2.0;
double& rd = d; // rd refers to d
const double& rcd = d; // rcd refers to d

struct A { };
struct B : A { operator int&(); } b;
A& ra = b; // ra refers to A subobject in b
const A& rca = b; // rca refers to A subobject in b
int& ir = B(); // ir refers to the result of B::operator int&
```

— end note]

(5.2) — 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:

```c
double& rd2 = 2.0; // error: not an lvalue and reference not const
int i = 2;
double& rd3 = i; // error: type mismatch and reference not const
```

— end example]

(5.2.1) — If the initializer expression

(5.2.1.1) — is an xvalue (but not a bit-field), class prvalue, array prvalue or function lvalue and “`cv1 T1`” is reference-compatible with “`cv2 T2`”, or

(5.2.1.2) — 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 xvalue, class prvalue, or function lvalue of type “`cv3 T3`”, where “`cv1 T1`” is reference-compatible with “`cv3 T3`” (see 13.3.1.6),

then the reference is bound to the value of the initializer expression in the first case and to the result of the conversion in the second case (or, in either case, to an appropriate base class subobject).

[Example:

```c
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]

(5.2.2) — Otherwise:

(5.2.2.1) — If `T1` or `T2` is a class type and `T1` is not reference-related to `T2`, user-defined conversions are considered using the rules for copy-initialization of an object of type “`cv1 T1`” by user-defined conversion (8.5, 13.3.1.4, 13.3.1.5); 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, a temporary of type “cv1 T1” is created and copy-initialized (8.5) from the initializer expression. The reference is then bound to the temporary.

If T1 is reference-related to T2:

— cv1 shall be the same cv-qualification as, or greater cv-qualification than, cv2; and
— if the reference is an rvalue reference, the initializer expression shall not be an lvalue.

Example:

```c
struct Banana { }
struct Enigma { operator const Banana(); }
struct Alaska { operator Banana&(); }
void enigmatic() {
    typedef const Banana ConstBanana;
    Banana &&banana1 = ConstBanana(); // ill-formed
    Banana &&banana2 = Enigma(); // ill-formed
    Banana &&banana3 = Alaska(); // ill-formed
}
```

```c
const double& rcd2 = 2; // rcd2 refers to temporary with value 2.0
double&& rrd = 2; // rrd refers to temporary with value 2.0
const volatile int cvi = 1;
const int& r2 = cvi;
struct A { operator volatile int&(); } a;
const int& r3 = a; // error: type qualifiers dropped
// from result of conversion function
```

```c
double d2 = 1.0;
double&& rrd2 = d2; // error: initializer is lvalue of related type
struct X { operator int&(); };
int&& rri2 = X(); // error: result of conversion function is lvalue of related type
int i3 = 2;
double&& rrd3 = i3; // rrd3 refers to temporary with value 2.0
```

— end example]

In all cases except the last (i.e., creating and initializing a temporary from the initializer expression), the reference is said to bind directly to the initializer expression.

6 [Note: 12.2 describes the lifetime of temporaries bound to references. — end note]

### 8.5.4 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 list are called the elements of the initializer list. An initializer list may be empty. List-initialization can occur in direct-initialization or copy-initialization contexts; list-initialization in a direct-initialization context is called direct-list-initialization and list-initialization in a copy-initialization context is called copy-list-initialization. [Note: List-initialization can be used

(1.1) — as the initializer in a variable definition (8.5)
(1.2) — as the initializer in a new-expression (5.3.4)
(1.3) — in a return statement (6.6.3)
(1.4) — as a `for-range-initializer` (6.5)
(1.5) — as a function argument (5.2.2)
(1.6) — as a subscript (5.2.1)
(1.7) — as an argument to a constructor invocation (8.5, 5.2.3)
(1.8) — as an initializer for a non-static data member (9.2)
(1.9) — in a `mem-initializer` (12.6.2)
(1.10) — on the right-hand side of an assignment (5.18)

Example:

```cpp
class S2 {
    int m1;
    double m2, m3;
};
S2 s21 = { 1, 2, 3.0 };  // OK
S2 s22 { 1.0, 2, 3 };    // error: narrowing
S2 s23 { };  // OK: default to 0,0,0
```

2 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 (8.3.6). [Note: Initializer-list constructors are favored over other constructors in list-initialization (13.3.1.7). 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 (14.8.2.1). — 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 (7.1.6.4) — the program is ill-formed.

3 List-initialization of an object or reference of type T is defined as follows:

(3.1) — If T is a class type 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).

(3.2) — Otherwise, if T is a character array and the initializer list has a single element that is an appropriately-typed string literal (8.5.2), initialization is performed as described in that section.

(3.3) — Otherwise, if T is an aggregate, aggregate initialization is performed (8.5.1).
— end example]

(3.4) — Otherwise, if the initializer list has no elements and T is a class type with a default constructor, the object is value-initialized.

(3.5) — Otherwise, if T is a specialization of std::initializer_list<E>, a prvalue initializer_list object is constructed as described below and used to initialize the object according to the rules for initialization of an object from a class of the same type (8.5).

(3.6) — Otherwise, if T is a class type, constructors are considered. The applicable constructors are enumerated and the best one is chosen through overload resolution (13.3, 13.3.1.7). If a narrowing conversion (see below) is required to convert any of the arguments, the program is ill-formed.

[Example:

```cpp
struct S {
    S(std::initializer_list<double>); // #1
    S(std::initializer_list<int>);    // #2
    S();                            // #3
    // ...
};
S s1 = { 1.0, 2.0, 3.0 };    // invoke #1
S s2 = { 1, 2, 3 };          // invoke #2
S s3 = { };                  // invoke #3
```
— end example]

[Example:

```cpp
struct Map {
    Map(std::initializer_list<std::pair<std::string,int>>);
};
Map ship = {{"Sophie",14}, {"Surprise",28}};
```
— end example]

[Example:

```cpp
struct S {
    // 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]

(3.7) — Otherwise, if the initializer list has a single element of type E and either T is not a reference type or its referenced type is reference-related to E, the object or reference is initialized from that element (by copy-initialization for copy-list-initialization, or by direct-initialization for direct-list-initialization); if a narrowing conversion (see below) is required to convert the element to T, the program is ill-formed.

[Example:

```cpp
int x1 {2};                // OK
int x2 {2.0};              // error: narrowing
```
— end example]

(3.8) Otherwise, if \( T \) is a reference type, a prvalue temporary of the type referenced by \( T \) is copy-list-initialized or direct-list-initialized, depending on the kind of initialization for the reference, and the reference is bound to that temporary. [Note: As usual, the binding will fail and the program is ill-formed if the reference type is an lvalue reference to a non-const type. — end note]

[Example:

```cpp
struct S {
    S(std::initializer_list<double>); // #1
    S(const std::string&); // #2
    // ...
};
const S& r1 = { 1, 2, 3.0 }; // OK: invoke #1
const S& r2 = "Spinach"; // OK: invoke #2
S& r3 = { 1, 2, 3 }; // error: initializer is not an lvalue
const int& i1 = { 1 }; // OK
const int& i2 = { 1.1 }; // error: narrowing
const int (&iar)[2] = { 1, 2 }; // OK: iar is bound to temporary array
— end example]

(3.9) Otherwise, if the initializer list has no elements, the object is value-initialized.

[Example:

```cpp
int** pp {};
// initialized to null pointer
— end example]

(3.10) 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 (14.5.3), 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. 

§ 8.5.4
An object of type `std::initializer_list<E>` is constructed from an initializer list as if the implementation allocated a temporary array of \(N\) elements of type `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 11) 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
class 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}};  
X x(std::initializer_list<double>(__a, __a+3));  
```

assuming that the implementation can construct an `initializer_list` object with a pair of pointers. — end example

The array has the same lifetime as any other temporary object (12.2), 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:

```cpp
typedef std::complex<double> cmplx;  
std::vector<cmplx> v1 = { 1, 2, 3 };  
void f() {  
  std::vector<cmplx> v2{ 1, 2, 3 };  
  std::initializer_list<int> i3 = { 1, 2, 3 };  
}  
struct A {  
  A() : i4{ 1, 2, 3 } {}  // creates an A with 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 a constructor's `ctor-initializer`, so the array persists only until the constructor exits, and so any use of the elements of `i4` after the constructor exits produces undefined behavior. — end example

Note: The implementation is free to allocate the array in read-only memory if an explicit array with the same initializer could be so allocated. — end note

A narrowing conversion is an implicit conversion

1. from a floating-point type to an integer type, or
2. from `long double` to `double` or `float`, or from `double` to `float`, except where the source is a constant expression and the actual value after conversion is within the range of values that can be represented (even if it cannot be represented exactly), or
(7.3) — from an integer type or unscoped enumeration type to a floating-point type, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted back to the original type, or

(7.4) — from an integer type or unscoped enumeration type to an integer type that cannot represent all the values of the original type, except where the source is a constant expression whose value after integral promotions will fit into the target type.

[Note: As indicated above, such conversions are not allowed at the top level in list-initializations. — end note] [Example:

```c
int x = 999;  // x is not a constant expression
const int y = 999;
const int z = 99;
char c1 = x;   // OK, though it might narrow (in this case, it does narrow)
char c2(x);   // error: might narrow
char c3(y);   // error: narrows (assuming char is 8 bits)
char c4(z);   // OK: no narrowing needed
unsigned char uc1 = {5}; // OK: no narrowing needed
unsigned char uc2 = {-1}; // error: narrows
unsigned int ui1 = {-1}; // error: narrows
signed int si1 =
    { (unsigned int)-1 }; // error: narrows
int ii = {2.0}; // error: narrows
float f1 { x }; // error: might narrow
float f2 { 7 }; // OK: 7 can be exactly represented as a float
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]
A class is a type. Its name becomes a class-name (9.1) within its scope.

\[
\text{class-name:} \\
\quad \text{identifier} \\
\quad \text{simple-template-id}
\]

Class-specifiers and elaborated-type-specifiers (7.1.6.3) are used to make class-names. An object of a class consists of a (possibly empty) sequence of members and base class objects.

\[
\text{class-specifier:} \\
\quad \text{class-head} \{ \text{member-specification}_{\text{opt}} \}
\]

The 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.

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]

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.

If a class is marked with the class-virt-specifier final and it appears as a base-type-specifier in a base-clause (Clause 10), 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:

\[
\text{struct A;} \\
\text{struct A final \{}; \quad // OK: definition of struct A,} \\
\quad \text{not value-initialization of variable final}
\]

\[
\text{struct X \{} \\
\text{struct C \{ constexpr operator int() \{ return 5; \} \}}; \\
\text{struct B final : C\{}; \quad // OK: definition of nested class B,} \\
\quad \text{not declaration of a bit-field member final}
\}
\]

— end example]

Complete objects and member subobjects of class type shall have nonzero size. [Note: Class objects can

\[107\] Base class subobjects are not so constrained.
be assigned, passed as arguments to functions, and returned by functions (except objects of classes for which copying or moving has been restricted; see 12.8). Other plausible operators, such as equality comparison, can be defined by the user; see 13.5. — end note]

5 A union is a class defined with the class-key union; it holds at most one data member at a time (9.5). [Note: Aggregates of class type are described in 8.5.1. — end note]

6 A trivially copyable class is a class that:

(6.1) has no non-trivial copy constructors (12.8),

(6.2) has no non-trivial move constructors (12.8),

(6.3) has no non-trivial copy assignment operators (13.5.3, 12.8),

(6.4) has no non-trivial move assignment operators (13.5.3, 12.8), and

(6.5) has a trivial destructor (12.4).

A trivial class is a class that has a default constructor (12.1), has no non-trivial default constructors, and is trivially copyable. [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 (10.3) and no virtual base classes (10.1),

(7.3) has the same access control (Clause 11) 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.\(^{108}\)

$M(X)$ is defined as follows:

(7.8) If $X$ is a non-union class type, the set $M(X)$ is empty if $X$ has no (possibly inherited (Clause 10)) non-static data members; otherwise, it consists of the type of the first non-static data member of $X$ (where said member may be an anonymous union), $X_0$, and the elements of $M(X_0)$.

(7.9) 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 $X$.

(7.10) If $X$ is a non-class 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:]

\(^{108}\) 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 (5.10).
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.

9 [ Note: Standard-layout classes are useful for communicating with code written in other programming languages. Their layout is specified in 9.2. — end note ]

10 A POD struct is a non-union class that is both a trivial class and a standard-layout class, and has no non-static data members of type non-POD struct, non-POD union (or array of such types). Similarly, a POD union is a union that is both a trivial class and a standard-layout class, and has no non-static data members of type non-POD struct, non-POD union (or array of such types). A POD class is a class that is either a POD struct or a POD union.

[ Example:

struct N {  // neither trivial nor standard-layout
    int i;
    int j;
    virtual ~N();
};

struct T {  // trivial but not standard-layout
    int i;
    private:
        int j;
};

struct SL {  // standard-layout but not trivial
    int i;
    int j;
    ~SL();
};

struct POD {  // both trivial and standard-layout
    int i;
    int j;
};

— end example ]

11 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 (7.3.1) 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.

109) The acronym POD stands for “plain old data.”
In such cases, the nested-name-specifier of the class-head-name of the definition shall not begin with a decltype-specifier.

### 9.1 Class names

A class definition introduces a new type. [Example:

```c
struct X { int a; };
struct Y { int a; };
X a1;
Y a2;
int a3;
```

declares three variables of three different types. This implies that

- `a1 = a2;`  // error: Y assigned to X
- `a1 = a3;`  // error: int assigned to X

are type mismatches, and that

- `int f(X);`
- `int f(Y);`

declare an overloaded (Clause 13) function `f()` and not simply a single function `f()` twice. For the same reason,

```c
struct S { int a; };
struct S { int a; };  // error, double definition
```

is ill-formed because it defines `S` twice. —end example]

A class declaration introduces the class name into the scope where it is declared and hides any class, variable, function, or other declaration of that name in an enclosing scope (3.3). 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 (3.4.4). [Example:

```c
struct stat {
    // ...
};

stat gstat;  // use plain stat to
            // define variable
int stat(struct stat*);  // redeclare stat as function

void f() {
    struct stat* ps;  // struct prefix needed
    // to name struct stat
    stat(ps);  // call stat()
}
```

—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:

```c
struct s { int a; };

void g() {
    struct s;  // hide global struct s
```

§ 9.1
class Vector;

class Matrix {
    friend Vector operator*(const Matrix&, const Vector&);
};

class Vector {
    friend Vector operator*(const Matrix&, const Vector&);
};

Declaration of friends is described in 11.3, operator functions in 13.5. — end example] — end note]

3 [Note: An elaborated-type-specifier (7.1.6.3) 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 (7.1.3) 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.

9.2 Class members

member-specification:
    member-declaration member-specification_opt
    access-specifier : member-specification_opt

§ 9.2
The member specification in a class definition declares the full set of members of the class; no member can be added elsewhere. Members of a class are data members, member functions (9.3), nested types, and enumerators. Data members and member functions are static or non-static; see 9.4. Nested types are classes (9.1, 9.7) and enumerations (7.2) defined in the class, and arbitrary types declared as members by use of a typedef declaration (7.1.3). The enumerators of an unscoped enumeration (7.2) defined in the class are members of the class. Except when used to declare friends (11.3), to declare an unnamed bit-field (9.6), or to introduce the name of a member of a base class into a derived class (7.3.3), or when the declaration is an empty-declaration, member-declarations declare members of the class, and each such member-declaration shall declare at least one member name of the class. A member shall not be declared twice in the member-specification, except that a nested class or member class template can be declared and then later defined, and except that 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 (3.9) (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, using-declarations introducing inheriting constructors (12.9), exception-specifications, and brace-or-equal-initializers for non-static data members (including such things in nested classes). Otherwise it is regarded as incomplete within its own class member-specification.

[Note: A single name can denote several function members provided their types are sufficiently different (Clause 13). — end note]

A brace-or-equal-initializer shall appear only in the declaration of a data member. (For static data members, see 9.4.2; for non-static data members, see 12.6.2). A brace-or-equal-initializer for a non-static data member 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 or register storage-class-specifier. Within a class definition, a member shall not be declared with the thread_local storage-class-specifier unless also declared static.

The decl-specifier-seq may be omitted in constructor, destructor, and conversion function declarations only; when declaring another kind of member the decl-specifier-seq shall contain a type-specifier that is not a cv-
qualifier. The member-declarator-list can be omitted only after a class-specifier or an enum-specifier or in a
friend declaration (11.3). A pure-specifier shall be used only in the declaration of a virtual function (10.3).

7 The optional attribute-specifier-seq in a member-declaration appertains to each of the entities declared by
the member-declerators; it shall not appear if the optional member-declarator-list is omitted.

8 A virt-specifier-seq shall contain at most one of each virt-specifier. A virt-specifier-seq shall appear only in
the declaration of a virtual member function (10.3).

9 Non-static (9.4) data members shall not have incomplete types. In particular, a class C shall not contain
a non-static member of class C, but it can contain a pointer or reference to an object of class C.

10 [ Note: See 5.1 for restrictions on the use of non-static data members and non-static member functions.
— end note ]

11 [ 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 ]

12 [ Example: A simple example of a class definition is

```c
struct tnode {
    char tword[20];
    int count;
    tnode* left;
    tnode* right;
};
```

which contains an array of twenty characters, an integer, and two pointers to objects of the same type. Once
this definition has been given, the declaration

```c
tnode s, *sp;
```

declares s to be a tnode and sp to be a pointer to a tnode. With these declarations, sp->count refers to
the count member of the object to which sp points; s.left refers to the left subtree pointer of the object
s; and s.right->tword[0] refers to the initial character of the tword member of the right subtree of s.
— end example ]

13 Nonstatic data members of a (non-union) class with the same access control (Clause 11) 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 11). 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 (10.3) and virtual base classes (10.1).

14 If T is the name of a class, then each of the following shall have a name different from T:

(14.1) — every static data member of class T;

(14.2) — every member function of class T [ Note: This restriction does not apply to constructors, which do not
have names (12.1) — end note ];

(14.3) — every member of class T that is itself a type;

(14.4) — every member template of class T;

(14.5) — every enumerator of every member of class T that is an unscoped enumerated type; and

(14.6) — every member of every anonymous union that is a member of class T.

15 In addition, if class T has a user-declared constructor (12.1), every non-static data member of class T shall
have a name different from T.
The common initial sequence of two standard-layout struct (Clause 9) 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:

```c
struct A { int a; char b; };
struct B { const int b1; volatile char b2; };
struct C { int c; unsigned : 0; char b; };
struct D { int d; char b : 4; };
struct E { unsigned int e; char b; };
```

The common initial sequence of A and B comprises all members of either class. The common initial sequence of A and C and of A and D comprises the first member in each case. The common initial sequence of A and E is empty. —end example]

Two standard-layout struct (Clause 9) types are layout-compatible if their common initial sequence comprises all members and bit-fields of both classes (3.9).

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 (3.9).

In a standard-layout union with an active member (9.5) of struct type T1, it is permitted to read a non-static data member m of another union member of struct type T2 provided m is part of the common initial sequence of T1 and T2. [Note: Reading a volatile object through a non-volatile glvalue has undefined behavior (7.1.6.1). —end note]

If a standard-layout class object has any non-static data members, its address is the same as the address of its first non-static data member. 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]

### 9.3 Member functions

1. Functions declared in the definition of a class, excluding those declared with a friend specifier (11.3), are called member functions of that class. A member function may be declared static in which case it is a static member function of its class (9.4); otherwise it is a non-static member function of its class (9.3.1, 9.3.2).

2. A member function may be defined (8.4) in its class definition, in which case it is an inline member function (7.1.2), 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 (14.7) appearing outside of the class definition, a member function shall not be redeclared.

3. 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. [Note: Member functions of a class in namespace scope have the linkage of that class. Member functions of a local class (9.8) have no linkage. See 3.5. —end note]

4. There shall be at most one definition of a non-inline member function in a program; no diagnostic is required. There may be more than one inline member function definition in a program. See 3.2 and 7.1.2.

5. 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 (8.3.6) or in the member function body) is looked up as described in 3.4. —end note] [Example:

```c
struct X {
```
typedef int T;
static T count;
void f(T);
};
void X::f(T t = count) { }

The member function \( f \) of class \( X \) is defined in global scope; the notation \( X::f \) specifies that the function \( f \) is a member of class \( X \) and in the scope of class \( X \). In the function definition, the parameter type \( T \) refers to the typedef member \( T \) declared in class \( X \) and the default argument \( \text{count} \) refers to the static data member \( \text{count} \) declared in class \( X \). —end example

6 A static local variable in a member function always refers to the same object, whether or not the member function is inline.

7 Previously declared member functions may be mentioned in friend declarations.

8 Member functions of a local class shall be defined inline in their class definition, if they are defined at all.

9 [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 8.3.5. For example,

t typedef void fv(void);
  typedef void fvc(void) const;
  struct S {
    fv memfunc1; // equivalent to: void memfunc1(void);
    void memfunc2();
    fvc memfunc3; // equivalent to: void memfunc3(void) const;
  };
  fv S::* pmfv1 = &S::memfunc1;
  fv S::* pmfv2 = &S::memfunc2;
  fvc S::* pmfv3 = &S::memfunc3;

Also see 14.3. —end note]

9.3.1 Nonstatic member functions [class.mfct.non-static]

1 A non-static member function may be called for an object of its class type, or for an object of a class derived (Clause 10) from its class type, using the class member access syntax (5.2.5, 13.3.1.1). A non-static member function may also be called directly using the function call syntax (5.2.2, 13.3.1.1) from within the body of a member function of its class or of a class derived from its class.

2 If a non-static member function of a class \( X \) is called for an object that is not of type \( X \), or of a type derived from \( X \), the behavior is undefined.

3 When an id-expression (5.1) that is not part of a class member access syntax (5.2.5) and not used to form a pointer to member (5.3.1) is used in a member of class \( X \) in a context where this can be used (5.1.1), if name lookup (3.4) 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 (5.2.5) using (**this) (9.3.2) 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 (5.1) used in the definition of a member function for class \( X \) resolves to a static member, an enumerator or a nested type of class \( X \) or of a base class of \( X \), the unqualified-id is transformed into a qualified-id (5.1) in which the nested-name-specifier names the class of the member function. [Example:

struct tnode {
    char tword[20];

§ 9.3.1
int count;
tnode* left;
tnode* right;
void set(const char*, tnode* l, tnode* r);
};

void tnode::set(const char* w, tnode* l, tnode* r) {
count = strlen(w)+1;
if (sizeof(tword)<=count)
    perror("tnode string too long");
strcpy(tword,w);
left = l;
right = r;
}

void f(tnode n1, tnode n2) {
n1.set("abc",&n2,0);
n2.set("def",0,0);
}

In the body of the member function tnode::set, the member names tword, count, left, and right refer to members of the object for which the function is called. Thus, in the call n1.set("abc",&n2,0), tword refers to n1.tword, and in the call n2.set("def",0,0), it refers to n2.tword. The functions strlen, perror, and strcpy are not members of the class tnode and should be declared elsewhere. — 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 (9.3.2). They also affect the function type (8.3.5) 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:

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 (8.3.5); see 13.3.1.

A non-static member function may be declared virtual (10.3) or pure virtual (10.4).

9.3.2 The this pointer

In the body of a non-static (9.3) 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:

struct s {
    int a;
    int f() const;
    int g() { return a++; }
}  

110) See, for example, <cstring> (21.8).
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

Similarly, volatile semantics (7.1.6.1) apply in volatile member functions when accessing the object and its non-static data members.

A cv-qualified member function can be called on an object-expression (5.2.5) only if the object-expression is as cv-qualified or less-cv-qualified than the member function. [Example:

```cpp
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]

Constructors (12.1) and destructors (12.4) 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 (12.1) and (12.4). —end note]

9.4 Static members [class.static]

A data or function member of a class may be declared static in a class definition, in which case it is a static member of the class.

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 (5.2.5) 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:

```cpp
struct process {
    static void reschedule();
};

process& g();

void f() {
    process::reschedule();  // OK: no object necessary
    g().reschedule();  // g() is called
}
```

—end example]

A static member may be referred to directly in the scope of its class or in the scope of a class derived (Clause 10) from its class; in this case, the static member is referred to as if a qualified-id expression was used, with the nested-name-specifier of the qualified-id naming the class scope from which the static member is referenced. [Example:

```cpp
int g();
struct X {
    static int g();
};
```
struct Y : X {
    static int i;
};
int Y::i = g();  // equivalent to Y::g();

--- end example ---

4 If an unqualified-id (5.1) is used in the definition of a static member following the member’s declarator-id, and name lookup (3.4.1) 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 5.1 for restrictions on the use of non-static data members and non-static member functions. —end note]

5 Static members obey the usual class member access rules (Clause 11). 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]

§ 9.4.1 Static member functions

[Note: The rules described in 9.3 apply to static member functions. —end note]

2 [Note: A static member function does not have a this pointer (9.3.2). —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 (13.1). A static member function shall not be declared const, volatile, or const volatile.

§ 9.4.2 Static data members

[Note: 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 static data member in its class definition is not a definition and may be of an incomplete type other than cv-qualified void. The definition for a static data member 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 (3.3.7). [Example:

```cpp
class process {
    static process* run_chain;
    static process* running;
};

process* process::running = get_main();
process* process::run_chain = running;
```

The static data member run_chain of class process is defined in global scope; the notation process::run_chain specifies that the member run_chain is a member of class process and in the scope of class process. In the static data member definition, the initializer expression refers to the static data member running of class process. —end example]

[Note: Once the static data member has been defined, it exists even if no objects of its class have been created. [Example: in the example above, run_chain and running exist even if no objects of class process are created by the program. —end example] —end note]
If a non-volatile `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 (5.20). A `static` data member of literal type can be declared in the class definition with the `constexpr` specifier; if so, its declaration shall specify a brace-or-equal-initializer in which every initializer-clause that is an assignment-expression is a constant expression. [Note: In both these cases, the member may appear in constant expressions. —end note] The member shall still be defined in a namespace scope if it is odr-used (3.2) in the program and the namespace scope definition shall not contain an initializer.

[Note: There shall be exactly one definition of a `static` data member that is odr-used (3.2) 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.

Static data members of a class in namespace scope have the linkage of that class (3.5). A local class shall not have `static` data members.

Static data members are initialized and destroyed exactly like non-local variables (3.6.2, 3.6.3).

A `static` data member shall not be `mutable` (7.1.1).

### 9.5 Unions

In a union, at most one of the non-static data members 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 (9.2), and if an object of this standard-layout union type contains one of the standard-layout structs, it is permitted to inspect the common initial sequence of any of the standard-layout struct members; see 9.2. —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. All non-static data members of a union object have the same address.

A union can have member functions (including constructors and destructors), but not virtual (10.3) 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: If any non-static data member of a union has a non-trivial default constructor (12.1), copy constructor (12.8), move constructor (12.8), copy assignment operator (12.8), move assignment operator (12.8), or destructor (12.4), the corresponding member function of the union must be user-provided or it will be implicitly deleted (8.4.3) for the union. —end note]

[Example: Consider the following union:

```cpp
union U {
    int i;
    float f;
    std::string s;
};
```

Since `std::string` (21.3) declares non-trivial versions of all of the special member functions, `U` will have an implicitly deleted default constructor, copy/move constructor, copy/move assignment operator, and destructor. To use `U`, some or all of these member functions must be user-provided. —end example]

[Note: In general, one must use explicit destructor calls and placement new operators to change the active member of a union. —end note] [Example: Consider an object `u` of a `union` type `U` having non-static data members `m` of type `M` and `n` of type `N`. If `M` has a non-trivial destructor and `N` has a non-trivial constructor (for instance, if they declare or inherit virtual functions), the active member of `u` can be safely switched from `m` to `n` using the destructor and placement new operator as follows:

```cpp
u.m.~M();
```
A union of the form

```c
union { member-specification } ;
```

is called an anonymous union; it defines an unnamed object of unnamed type. 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:

```c
void f() {
    union { int a; const char* p; };  
    a = 1;
    p = "Jennifer";
}
```

Here `a` and `p` are used like ordinary (nonmember) 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 11). An anonymous union shall not have function members.

A union for which objects, pointers, or references are declared is not an anonymous union. [Example:

```c
void f() {
    union { int aa; char* p; } obj, *ptr = &obj;
    aa = 1; // error
    ptr->aa = 1; // OK
}
```

The assignment to plain `aa` is ill-formed since the member name is not visible outside the union, and even if it were visible, it is not associated with any particular object. —end example] [Note: Initialization of unions with no user-declared constructors is described in (8.5.1). —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 brace-or-equal-initializer. [Example:

```c
union U {
    int x = 0;
    union {  }
    union {
        int x;
        int y = 1; // error: initialization for second variant member of U
    }
};
```

—end example]
9.6 Bit-fields

A member-declarator of the form

\[ \text{identifier}_{\text{opt}} \text{ attribute-specifier-seq}_{\text{opt}} : \text{constant-expression} \]

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 (3.9) of the bit-field’s type; in such cases the extra bits are used as padding bits and do not participate in the value representation (3.9) of the bit-field. Allocation of bit-fields within a class object is implementation-defined. Alignment of bit-fields is implementation-defined. Bit-fields are packed into some addressable allocation unit. \[ \text{Note: Bit-fields straddle allocation units on some machines, left-to-right on others. — end note} \]

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. \[ \text{Note: An unnamed bit-field is useful for padding to conform to externally-imposed layouts. — end note} \] 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 (3.9.1). 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 (8.5.3). \[ \text{Note: If the initializer for a reference of type const T& is an lvalue that refers to a bit-field, the reference is bound to a temporary initialized to hold the value of the bit-field; the reference is not bound to the bit-field directly. See 8.5.3. — end note} \]

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 (7.2), the original enumerator value and the value of the bit-field shall compare equal. \[ \text{Example:} \\

```cpp
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]

9.7 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. \[ \text{Note: See 5.1 for restrictions on the use of non-static data members and non-static member functions. — end note} \]

\[ \text{Example:} \\

```cpp
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 { };
  class E::I2 { };
};

—end example
```

Like a member function, a friend function (11.3) 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 (9.4), but it has no
special access rights to members of an enclosing class.
9.8 Local class declarations

1 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. Declarations in a local class shall not odr-use (3.2) a variable with automatic storage duration from an enclosing scope. [Example:

```c
int x;
void f() {
    static int s ;
    int x;
    const int N = 5;
    extern int q();

    struct local {
        int g() { return x; }  // error: odr-use of automatic 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 automatic variable N
    };

    local* p = 0;        // error: local not in scope
}
```
—end example]

2 An enclosing function has no special access to members of the local class; it obeys the usual access rules (Clause 11). Member functions of a local class shall be defined within their class definition, if they are defined at all.

3 If class X is a local class a nested class Y may be declared in class X and later defined in the definition of class X or be later defined in the same scope as the definition of class X. A class nested within a local class is a local class.

4 A local class shall not have static data members.

9.9 Nested type names

1 Type names obey exactly the same scope rules as other names. In particular, type names defined within a class definition cannot be used outside their class without qualification. [Example:

```c
struct X {
    typedef int I;
    class Y { /* ... */ };
    I a;
};

I b;          // error
Y c;          // error
X::Y d;       // OK
X::I e;       // OK
```

—end example]
10 Derived classes

1 A list of base classes can be specified in a class definition using the notation:

```
base-clause:
  : base-specifier-list
base-specifier-list:
  base-specifier ... opt
  base-specifier-list , base-specifier ... opt
base-specifier:
  attribute-specifier-seq opt base-type-specifier
  attribute-specifier-seq opt virtual access-specifier opt base-type-specifier
  attribute-specifier-seq opt access-specifier virtual opt base-type-specifier
class-or-decltype:
  nested-name-specifier opt class-name
decltype-specifier
base-type-specifier:
  class-or-decltype
access-specifier:
  private
  protected
  public
```

The optional attribute-specifier-seq appertains to the base-specifier.

2 The type denoted by a base-type-specifier shall be a class type that is not an incompletely defined class (Clause 9); this class is called a direct base class for the class being defined. During the lookup for a base class name, non-type names are ignored (3.3.10). 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 11 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. The base class members are said to be inherited by the derived class. 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 (10.2). [Note: The scope resolution operator :: (5.1) can be used to refer to a direct or indirect base member explicitly. This allows access to a name that has been redeclared in the derived class. A derived class can itself serve as a base class subject to access control; see 11.2. A pointer to a derived class can be implicitly converted to a pointer to an accessible unambiguous base class (4.10). An lvalue of a derived class type can be bound to a reference to an accessible unambiguous base class (8.5.3). — end note]

3 The base-specifier-list specifies the type of the base class subobjects contained in an object of the derived class type. [Example:

```
struct Base {
  int a, b, c;
};

struct Derived : Base {
  int b;
};
```

Derived classes 243
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 (14.5.3).

The order in which the base class subobjects are allocated in the most derived object (1.8) 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.” A DAG of subobjects is often referred to as a “subobject lattice.”]

![Directed acyclic graph]

Figure 2 — Directed acyclic graph

The arrows need not have a physical representation in memory. —end note]

[Note: Initialization of objects representing base classes can be specified in constructors; see 12.6.2. — end note]

[Note: A base class subobject might have a layout (3.7) different from the layout of a most derived object of the same type. A base class subobject might have a polymorphic behavior (12.7) different from the polymorphic behavior of a most derived object of the same type. A base class subobject may be of zero size (Clause 9); 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 (5.10). — end note]

10.1 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 (12.6.2), cleanup (12.4), and storage layout (9.2, 11.1). — 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 { /* ... */};

§ 10.1 244
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 (1.8) 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. [Example: 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 below.

```
L L
A -- B
C
```

Figure 3 — Non-virtual base

5 In such lattices, explicit qualification can be used to specify which subobject is meant. The body of function C::f could refer to the member `next` of each L subobject:

```cpp
void C::f() { A::next = B::next; } // well-formed
```

Without the A:: or B:: qualifiers, the definition of C::f above would be ill-formed because of ambiguity (10.2).

6 For another example,

```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 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 below.

7 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 below.

--- end example

10.2 Member name lookup

Member name lookup determines the meaning of a name (`id-expression`) in a class scope (3.3.7). 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 (3.4, Clause 11).

The following steps define the result of name lookup for a member name `f` in a class scope `C`.

1. 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 (7.3.3), and type declarations (including injected-class-names) are replaced by the types they designate. `S(f,C)` is calculated as follows:

2. 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 (3.4.4) or base-specifier (Clause 10), for instance, ignores all non-type declarations, while looking up a name in a nested-name-specifier (3.4.3) ignores function, variable, and enumerator declarations. As another example, looking up a name in a using-declaration (7.3.3) includes the declaration of a class or enumeration that would ordinarily be hidden by another declaration of that name. — end note]
in the same scope. — end note] If the resulting declaration set is not empty, the subobject set contains C itself, and calculation is complete.

5 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).

6 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.

7 The result of name lookup for f in C is the declaration set of S(f,C). If it is an invalid set, the program is ill-formed. [Example:

```c
struct A { int x; }; // S(x,A) = { { A::x }, { A } }
struct B { float x; }; // S(x,B) = { { B::x }, { B } }
struct C: public A, public B { }; // S(x,C) = { invalid, { A in C, B in C } }
struct D: public virtual C { }; // S(x,D) = S(x,C)
struct E: public virtual C { char x; }; // S(x,E) = { { E::x }, { E } }
struct F: public D, public E { }; // S(x,F) = S(x,E)
int main() {
    F f;
    f.x = 0; // OK, lookup finds E::x
}
```

S(x,F) is unambiguous because the A and B base subobjects of D are also base subobjects of E, so S(x,D) is discarded in the first merge step. — end example]

8 If the name of an overloaded function is unambiguously found, overloading resolution (13.3) also takes place before access control. Ambiguities can often be resolved by qualifying a name with its class name. [Example:

```c
struct A {
    int f();
};

struct B {
    int f();
};

struct C : A, B {
    int f() { return A::f() + B::f(); }
};

— end example]

9 [Note: A static member, a nested type or an enumerator defined in a base class T can unambiguously be found even if an object has more than one base class subobject of type T. Two base class subobjects share the non-static member subobjects of their common virtual base classes. — end note] [Example:
struct V {
    int v;
};
struct A {
    int a;
    static int s;
    enum { e };
};
struct B : A, virtual V { };
struct C : A, virtual V { };
struct D : B, C { };

void f(D* pd) {
    pd->v++;
    // OK: only one v (virtual)
    pd->s++;
    // OK: only one s (static)
    int i = pd->e;
    // OK: only one e (enumerator)
    pd->a++;
    // error, ambiguous: two a in D
}

— end example

[ 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)

Note: 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. — end note]

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()
}

§ 10.2
An explicit or implicit conversion from a pointer to or an expression designating an object of a derived class to a pointer or reference to one of its base classes shall unambiguously refer to a unique object representing the base class. [Example:

```c
struct V { };  
struct A { };  
struct B : A, virtual V { };  
struct C : A, virtual V { };  
struct D : B, C { };  

void g() {
  D d;  
  B* pb = &d;  
  A* pa = &d;  // error, ambiguous: C's A or B's A? 
  V* pv = &d;  // OK: only one V subobject
}
```
—end example]

[Note: Even if the result of name lookup is unambiguous, use of a name found in multiple subobjects might still be ambiguous (4.11, 5.2.5, 11.2). — end note] [Example:

```c
struct B1 {  
  void f();  
  static void f(int);  
  int i;
};  
struct B2 {  
  void f(double);  
};  
struct I1: B1 { };  
struct I2: B1 { };

struct D: I1, I2, B2 {  
  using B1::f;  
  using B2::f;  
  void g() {
    f();  // Ambiguous conversion of this
    f(0);  // Unambiguous (static)
    f(0.0);  // Unambiguous (only one B2)
    int B1::* mpB1 = &D::i;  // Unambiguous
    int D::* mpD = &D::i;  // Ambiguous conversion
  }
};
```
—end example]

10.3 Virtual functions  [class.virtual]

Virtual functions support dynamic binding and object-oriented programming. A class that declares or inherits a virtual function is called a **polymorphic class**.

1. If a virtual member function \texttt{vf} is declared in a class \texttt{Base} and in a class \texttt{Derived}, derived directly or indirectly from \texttt{Base}, a member function \texttt{vf} with the same name, parameter-type-list (8.3.5), cv-qualification, and ref-
qualifier (or absence of same) as \texttt{Base::vf} is declared, then \texttt{Derived::vf} is also virtual (whether or not it is so declared) and it \textit{overrides} \texttt{Base::vf}. For convenience we say that any virtual function overrides itself. A virtual member function \texttt{C::vf} of a class object \texttt{S} is a \textit{final overrider} unless the most derived class (1.8) of which \texttt{S} is a base class subobject (if any) declares or inherits another member function that overrides \texttt{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. \[ Example:\]

\begin{verbatim}
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{verbatim}

\end{quote}

\[ Example: \]

\begin{verbatim}
struct A { virtual void f(); }; 
struct B : A { }; 
struct C : A { void f(); }; 
struct D : B, C { }; // OK: A::f and C::f are the final overriders 

// for the B and C subobjects, respectively
\end{verbatim}

\end{quote}

\[ Note: A virtual member function does not have to be visible to be overridden, for example, \]

\begin{verbatim}
struct B { 
  virtual void f(); 
};
struct D : B { 
  void f(int);
};
struct D2 : D { 
  void f();
}
\end{verbatim}

the function \texttt{f(int)} in class \texttt{D} hides the virtual function \texttt{f()} in its base class \texttt{B}; \texttt{D::f(int)} is not a virtual function. However, \texttt{f()} declared in class \texttt{D2} has the same name and the same parameter list as \texttt{B::f()}, and therefore is a virtual function that overrides the function \texttt{B::f()} even though \texttt{B::f()} is not visible in class \texttt{D2}. \[ end note]\n
\[ If a virtual function \texttt{f} in some class \texttt{B} is marked with the \textit{virt-specifier final} and in a class \texttt{D} derived from \texttt{B} a function \texttt{D::f} overrides \texttt{B::f}, the program is ill-formed. \[ Example: \]

\begin{verbatim}
struct A { virtual void f(); }; 
struct B : A { }; 
struct C : A { void f(); }; 
struct D : B, C { }; // OK: A::f and C::f are the final overriders 

// for the B and C subobjects, respectively
\end{verbatim}

\end{quote}

\[ end example]\n
\[ Note: A function with the same name but a different parameter list (Clause 13) as a virtual function is not necessarily virtual and does not override. The use of the \textit{virtual} specifier in the declaration of an overriding function is legal but redundant (has empty semantics). Access control (Clause 11) is not considered in determining overriding. \]
struct B {
    virtual void f() const final;
};

struct D : B {
    void f() const; // error: D::f attempts to override final B::f
};

— end example]

5 If a virtual function is marked with the virt-specifier override and does not override a member function of a base class, the program is ill-formed. [Example:

    struct B {
        virtual void f(int);
    };

    struct D : B {
        virtual void f(long) override; // error: wrong signature overriding B::f
        virtual void f(int) override; // OK
    };

— end example]

6 Even though destructors are not inherited, a destructor in a derived class overrides a base class destructor declared virtual; see 12.4 and 12.5.

7 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:

(7.1) — both are pointers to classes, both are lvalue references to classes, or both are rvalue references to classes

(7.2) — 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

(7.3) — 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.

8 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 (5.2.2). [Example:

    class B {
    class D : private B { friend class Derived; };
    struct Base {
        virtual void vf1();
        virtual void vf2();
        virtual void vf3();
        virtual B* vf4();
        virtual B* vf5();
        void f();
    };

112) Multi-level pointers to classes or references to multi-level pointers to classes are not allowed.
struct No_good : public Base {
    D* vf4(); // error: B (base class of D) inaccessible
};

class A;
struct Derived : public Base {
    void vf1(); // virtual and overrides Base::vf1()
    void vf2(int); // not virtual, hides Base::vf2()
    char vf3(); // error: invalid difference in return type only
    D* vf4(); // OK: returns pointer to derived class
    A* vf5(); // error: returns pointer to incomplete class
    void f();
};

void g() {
    Derived d;
    Base* bp = &d;
    // standard conversion:
    Derived* to Base*
    bp->vf1(); // calls Derived::vf1()
    bp->vf2(); // calls Base::vf2()
    bp->f(); // calls Base::f() (not virtual)
    B* p = bp->vf4(); // calls Derived::pf() and converts the
    // result to B*
    Derived* dp = &d;
    D* q = dp->vf4(); // calls Derived::pf() and does not
    // convert the result to B*
    dp->vf2(); // ill-formed: argument mismatch
}

—end example—

9 [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) (5.2.2). —end note]

10 [Note: The virtual specifier implies membership, so a virtual function cannot be a nonmember (7.1.2) 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 in another class. —end note]

11 A virtual function declared in a class shall be defined, or declared pure (10.4) in that class, or both; but no diagnostic is required (3.2).

12 [Example: here are some uses of virtual functions with multiple base classes:

struct A {
    virtual void f();
};

struct B1 : A { // note non-virtual derivation
    void f();
};

struct B2 : A {
    void f();
};
struct D : B1, B2 {    // D has two separate A subobjects
};

void foo() {
    D d;
    // A* ap = &d; // would be ill-formed: ambiguous
    B1* blp = &d;
    A* ap = blp;
    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.

The following example shows a function that does not have a unique final overrider:

struct A {
    virtual void f();
};

struct VB1 : virtual A {    // note virtual derivation
    void f();
};

struct VB2 : virtual A {    void f();
};

struct Error : VB1, VB2 {    // 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.

The following example uses the well-formed classes from above.

struct VB1a : virtual A {    // does not declare f
};

struct Da : VB1a, VB2 {
};

void foe() {
    VB1a* vblap = new Da;
    vblap->f();    // calls VB2::f
}

— end example]

Explicit qualification with the scope operator (5.1) 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 (8.4) 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.

10.4 Abstract classes [class.abstract]

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.

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 (9.2) in the function declaration in the class definition. A pure virtual function need be defined only if called with, or as if with (12.4), the qualified-id syntax (5.1).

Example:

class point { /* ... */ };
class shape {
    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:

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:

class ab_circle : public shape {
    int radius;

§ 10.4
public:
    void rotate(int) { }
    // ab_circle::draw() is a pure virtual
};

Since shape::draw() is a pure virtual function ab_circle::draw() is a pure virtual by default. The alternative declaration,

class circle : public shape {
    int radius;
public:
    void rotate(int) { }
    void draw(); // a definition is required somewhere
};

would make class circle nonabstract 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 (10.3) to a pure virtual function directly or indirectly for the object being created (or destroyed) from such a constructor (or destructor) is undefined.
11 Member access control [class.access]

1 A member of a class can be

(1.1) — **private**; that is, its name can be used only by members and friends of the class in which it is declared.

(1.2) — **protected**; that is, its name can be used only by members and friends of the class in which it is declared, by classes derived from that class, and by their friends (see 11.4).

(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.\(^\underline{113}\)

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 (11.3) and **using-declarations** (7.3.3). — 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 { };  // OK, typedef name A::BB is public
  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

\(^{113}\) Access permissions are thus transitive and cumulative to nested and local classes.
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 11 affect the ability to access a class member name from the declaration of a
particular entity, including parts of the declaration preceding the name of the entity being declared and, if the
entity is a class, the definitions of members of the class appearing outside the class’s member-specification.
[Note: this access also applies to implicit references to constructors, conversion functions, and destructors.
—end note] [Example:

```c
class A {
    typedef int I;   // private member
    I f();
    friend I g(I);
    static I x;
    template<int> struct Q;
    template<int> friend struct R;
    protected:
        struct B { };
    };

    A::I A::f() { return 0; }
    A::I g(A::I p = A::x);
    A::I g(A::I p) { return 0; }
    A::I A::x = 0;
    template<A::I> struct A::Q { };
    template<A::I> struct R { };

    struct D: A::B, A { };
```

Here, all the uses of A::I are well-formed because A::f, A::x, and A::Q are members of class A and g and
R are friends of class A. This implies, for example, that access checking on the first use of A::I must be
defered 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]

7 The names in a default argument (8.3.6) 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 14.7.1.

8 The names in a default template-argument (14.1) have their access checked in the context in which they
appear rather than at any points of use of the default template-argument. [Example:

```c
class B { }
template <class T> class C {
    protected:
        typedef T TT;
    };

    template <class U, class V = typename U::TT>
    class D : public U { };

    D <C<B>> *d;    // access error, C::TT is protected
```

—end example]

11.1 Access specifiers [class.access.spec]

1 Member declarations can be labeled by an access-specifier (Clause 10):

§ 11.1
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:

```cpp
class X {
    int a;       // X::a is private by default: class used
public:
    int b;       // X::b is public
    int c;       // X::c is public
};
```

— end example]

Any number of access specifiers is allowed and no particular order is required. [Example:

```cpp
struct S {
    int a;       // S::a is public by default: struct used
protected:
    int b;       // S::b is protected
private:
    int c;       // S::c is private
public:
    int d;       // S::d is public
};
```

— end example]

[Note: The effect of access control on the order of allocation of data members is described in 9.2. — end note]

When a member is redeclared within its class definition, the access specified at its redeclaration shall be the same as at its initial declaration. [Example:

```cpp
struct S {
    class A;
    enum E : int;
private:
    class A { };    // error: cannot change access
    enum E: int { e0 }; // error: cannot change access
};
```

— end example]

[Note: In a derived class, the lookup of a base class name will find the injected-class-name instead of the name of the base class in the scope in which it was declared. The injected-class-name might be less accessible than the name of the base class in the scope in which it was declared. — end note]

[Example:

```cpp
class A { };  
class B : private A { };  
class C : public B {
    A* p;          // error: injected-class-name A is inaccessible
    ::A* q;        // OK
};
```

— end example]
11.2 Accessibility of base classes and base class members

If a class is declared to be a base class (Clause 10) 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.

In the absence of an access-specifier for a base class, public is assumed when the derived class is defined with the class-key struct and private is assumed when the class is defined with the class-key class. [Example:]

```cpp
class B { /* ... */};
class D1 : private B { /* ... */};
class D2 : public B { /* ... */};
class D3 : B { /* ... */}; // B private by default
struct D4 : public B { /* ... */};
struct D5 : private B { /* ... */};
struct D6 : B { /* ... */}; // B public by default
class D7 : protected B { /* ... */};
struct D8 : protected B { /* ... */};
```

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]\n
[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 (4.10) and explicit casts (5.4), a conversion from a pointer to a derived class to a pointer to an inaccessible base class might be ill-formed if an implicit conversion is used, but well-formed if an explicit cast is used. For example,]

```cpp
class B {
    public:
        int mi; // non-static member
        static int si; // static member
};
class D : private B {
};
class DD : public D {
    void f();
};

void DD::f() {
    mi = 3; // error: mi is private in D
    si = 3; // error: si is private in D
    ::B b;
b.mi = 3; // OK (b.mi is different from this->mi)
b.si = 3; // OK (b.si is different from this->si)
::B::si = 3; // OK
::B* bp1 = this; // error: B is a private base class
::B* bp2 = (::B*)this; // OK with cast
bp2->mi = 3; // OK: access through a pointer to B.
}
```

\[114\] As specified previously in Clause 11, private members of a base class remain inaccessible even to derived classes unless friend declarations within the base class definition are used to grant access explicitly.

§ 11.2
A base class $B$ of $N$ is accessible at $R$, if

(4.1) an invented public member of $B$ would be a public member of $N$, or

(4.2) $R$ occurs in a member or friend of class $N$, and an invented public member of $B$ would be a private or protected member of $N$, or

(4.3) $R$ occurs in a member or friend of a class $P$ derived from $N$, and an invented public member of $B$ would be a private or protected member of $P$, or

(4.4) there exists a class $S$ such that $B$ is a base class of $S$ accessible at $R$ and $S$ is a base class of $N$ accessible at $R$.

[Example:

```cpp
class B {
    public:
        int m;
};
class S: private B {
    friend class N;
};
class N: private S {
    void f() {
        B* p = this; // OK because class S satisfies the fourth condition  
        // above: B is a base class of N accessible in f() because 
        // B is an accessible base class of S and S is an accessible 
        // base class of N.
    }
};
```

— end example]

If a base class is accessible, one can implicitly convert a pointer to a derived class to a pointer to that base class (4.10, 4.11). [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 (5.2.5) 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

(5.1) $m$ as a member of $N$ is public, or

(5.2) $m$ as a member of $N$ is private, and $R$ occurs in a member or friend of class $N$, or

(5.3) $m$ as a member of $N$ is protected, and $R$ occurs in a member or friend of class $N$, or in a member or friend of a class $P$ derived from $N$, where $m$ as a member of $P$ is public, private, or protected, or

(5.4) there exists a base class $B$ of $N$ that is accessible at $R$, and $m$ is accessible at $R$ when named in class $B$.

[Example:
6 If a class member access operator, including an implicit “this->,” is used to access a non-static data member or non-static member function, the reference is ill-formed if the left operand (considered as a pointer in the “.” operator case) cannot be implicitly converted to a pointer to the naming class of the right operand. [Note: This requirement is in addition to the requirement that the member be accessible as named. —end note]

11.3 Friends

A friend of a class is a function or class that is given permission to use the private and protected member names from the class. A class specifies its friends, if any, by way of friend declarations. Such declarations give special access rights to the friends, but they do not make the nominated friends members of the befriending class. [Example: the following example illustrates the differences between members and friends:

```cpp
class X {
   int a;
   friend void friend_set(X*, int);
public:
   void member_set(int);
};

void friend_set(X* p, int i) { p->a = i; }
void X::member_set(int i) { a = i; }
void f() {
   X obj;
   friend_set(&obj, 10);
   obj.member_set(10);
}

—end example]

Declaring a class to be a friend implies that the names of private and protected members from the class granting friendship can be accessed in the base-specifiers and member declarations of the befriended class. [Example:

```cpp
class A {
   class B {}
   friend class X;
};

struct X : A::B { // OK: A::B accessible to friend
   A::B mx; // OK: A::B accessible to member of friend
```

§ 11.3
class Y {
   A::B my;  // OK: A::B accessible to nested member of friend
};

— end example] [Example:

class X {
   enum { a=100 };  
   friend class Y;
};

class Y {
   int v[X::a];  // OK, Y is a friend of X
};

class Z {
   int v[X::a];  // error: X::a is private
};

— end example]

A class shall not be defined in a friend declaration. [Example:

class A {
   friend class B { };  // error: cannot define class in friend declaration
};

— end example]

3 A friend declaration that does not declare a function shall have one of the following forms:

   friend elaborated-type-specifier;
   friend simple-type-specifier;
   friend typename-specifier;

[Note: A friend declaration may be the declaration in a template-declaration (Clause 14, 14.5.4).—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

§ 11.3
A function first declared in a friend declaration has the linkage of the namespace of which it is a member (3.5). Otherwise, the function retains its previous linkage (7.1.1).

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:

```c
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 (9.8), the function name is unqualified, and the function has namespace scope. [Example:

```c
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 inline. 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 (3.4.1).

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 (9.2) portion of the class member-specification.

Friendship is neither inherited nor transitive. [Example:

```c
class A {
    friend class B;
    int a;
};

class B {
    friend class C;
};

class C {
    void f(A* p) { // error: C is not a friend of A
        p->a++;
        // despite being a friend of a friend
    }
};

class D : public B { // error: D is not a friend of A
    void f(A* p) {
        p->a++;
        // despite being derived from a friend
    }
};
```

§ 11.3
If a friend declaration appears in a local class (9.8) and the name specified is an unqualified name, a prior declaration is looked up without considering scopes that are outside the innermost enclosing non-class scope. For a friend function declaration, if there is no prior declaration, the program is ill-formed. For a friend class declaration, if there is no prior declaration, the class that is specified belongs to the innermost enclosing non-class scope, but if it is subsequently referenced, its name is not found by name lookup until a matching declaration is provided in the innermost enclosing non-class scope. [Example:

```
class X;
void a();
void f() {
    class Y;
    extern void b();
class A {
    friend class X; // OK, but X is a local class, not ::X
    friend class Y; // OK
    friend class Z; // OK, introduces local class Z
    friend void a(); // error, ::a is not considered
    friend void b(); // OK
    friend void c(); // error
    };
X* px; // OK, but ::X is found
Z* pz; // error, no Z is found
}
```
—end example]

11.4 Protected member access [class.protected]

An additional access check beyond those described earlier in Clause 11 is applied when a non-static data member or non-static member function is a protected member of its naming class (11.2)\(^\text{115}\). 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 (5.3.1), the nested-name-specifier shall denote C or a class derived from C. All other accesses involve a (possibly implicit) object expression (5.2.5). In this case, the class of the object expression shall be C or a class derived from C. [Example:

```
class B {
protected:
    int i;
    static int j;
};
class D1 : public B {
};
class D2 : public B {
    friend void fr(B*,D1*,D2*);
    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)
```

\(^{115}\) This additional check does not apply to other members, e.g., static data members or enumerator member constants.
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::* pmi_B2 = &D2::i; // OK (type of &D2::i is int B::*
B::j = 5; // OK (because refers to static member)
D2::j = 6; // OK (because refers to static member)
}

void D2::mem(B* pb, D1* p1) {
pb->i = 1; // ill-formed
p1->i = 2; // ill-formed
i = 3; // OK (access through this)
B::i = 4; // OK (access through this, qualification ignored)
int B::* pmi_B = &B::i; // ill-formed
int B::* pmi_B2 = &D2::i; // OK
j = 5; // OK (because j refers to static member)
B::j = 6; // OK (because B::j refers to static member)
}

void g(B* pb, D1* p1, D2* p2) {
pb->i = 1; // ill-formed
p1->i = 2; // ill-formed
p2->i = 3; // ill-formed
}

—end example

11.5 Access to virtual functions

The access rules (Clause 11) for a virtual function are determined by its declaration and are not affected by the rules for a function that later overrides it. [Example:

class B {
public:
  virtual int f();
};

class D : public B {
private:
  int f();
};

void f() {
  D d;
  B* pb = &d;
  D* pd = &d;

  pb->f(); // OK: B::f() is public,
  // D::f() is invoked
  pd->f(); // error: D::f() is private
}

—end example

2 Access is checked at the call point using the type of the expression used to denote the object for which the member function is called (B* in the example above). The access of the member function in the class in which it was defined (D in the example above) is in general not known.
11.6 Multiple access

If a name can be reached by several paths through a multiple inheritance graph, the access is that of the path that gives most access. [Example:

```cpp
class W { public: void f(); }; 
class A : private virtual W { }; 
class B : public virtual W { }; 
class C : public A, public B {
   void f() { W::f(); } // OK
};
```

2 Since \(W::f()\) is available to \(C::f()\) along the public path through \(B\), access is allowed. —end example]

11.7 Nested classes

A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules (Clause 11) 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]
12 Special member functions [special]

1 The default constructor (12.1), copy constructor and copy assignment operator (12.8), move constructor and move assignment operator (12.8), and destructor (12.4) 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 (3.2). See 12.1, 12.4 and 12.8. — 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.

2 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.

```c
struct A { }; // implicitly declared A::operator=
struct B : A {
    B& operator=(const B &);
};
B& B::operator=(const B& s) {
    this->A::operator=(s);  // well formed
    return *this;
}
— end example]
```

3 [Note: The special member functions affect the way objects of class type are created, copied, moved, and destroyed, and how values can be converted to values of other types. Often such special member functions are called implicitly. — end note]

4 Special member functions obey the usual access rules (Clause 11). [Example: declaring a constructor protected ensures that only derived classes and friends can create objects using it. — end example]

5 For a class, its non-static data members, its non-virtual direct base classes, and, if the class is not abstract (10.4), its virtual base classes are called its potentially constructed subobjects.

12.1 Constructors [class.ctor]

1 Constructors do not have names. In a declaration of a constructor, the declarator is a function declarator (8.3.5) of the form

```
ptr-declarator ( parameter-declaration-clause ) exception-specification_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 (11.3), the id-expression is the injected-class-name (Clause 9) 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 a class-name that names the current instantiation (14.6.2.1) of the immediately-enclosing class template; or

(1.3) — in a declaration at namespace scope or in a friend declaration, the id-expression is a qualified-id that names a constructor (3.4.3.1).

§ 12.1
The class-name shall not be a typedef-name. In a constructor declaration, each decl-specifier in the optional decl-specifier-seq shall be friend, inline, explicit, or constexpr. [Example:

```c
struct S {
    S(); // declares the constructor
};
S::S() { } // defines the constructor
```
— end example]

2 A constructor is used to initialize objects of its class type. Because constructors do not have names, they are never found during name lookup; however an explicit type conversion using the functional notation (5.2.3) will cause a constructor to be called to initialize an object. [Note: For initialization of objects of class type see 12.6. — end note]

3 A constructor can be invoked for a const, volatile or const volatile object. const and volatile semantics (7.1.6.1) are not applied on an object under construction. They come into effect when the constructor for the most derived object (1.8) ends.

4 A default constructor for a class X is a constructor of class X that either has no parameters or else each parameter that is not a function parameter pack has a default argument. If there is no user-declared constructor for class X, a constructor having no parameters is implicitly declared as defaulted (8.4). An implicitly-declared default constructor is an inline public member of its class. A defaulted default constructor for class X is defined as deleted if:

(4.1) — X is a union-like class that has a variant member with a non-trivial default constructor,

(4.2) — any non-static data member with no brace-or-equal-initializer is of reference type,

(4.3) — any non-variant non-static data member of const-qualified type (or array thereof) with no brace-or-equal-initializer does not have a user-provided default constructor,

(4.4) — X is a union and all of its variant members are of const-qualified type (or array thereof),

(4.5) — X is a non-union class and all members of any anonymous union member are of const-qualified type (or array thereof),

(4.6) — any potentially constructed subobject, except for a non-static data member with a brace-or-equal-initializer, has class type M (or array thereof) and either M has no default constructor or overload resolution (13.3) as applied to M’s default constructor results in an ambiguity or in a function that is deleted or inaccessible from the defaulted default constructor, or

(4.7) — any potentially constructed subobject has a type with a destructor that is deleted or inaccessible from the defaulted default constructor.

A default constructor is trivial if it is not user-provided and if:

(4.8) — its class has no virtual functions (10.3) and no virtual base classes (10.1), and

(4.9) — no non-static data member of its class has a brace-or-equal-initializer, and

(4.10) — all the direct base classes of its class have trivial default constructors, and

(4.11) — 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.

§ 12.1 268
Otherwise, the default constructor is non-trivial.

A default constructor that is defaulted and not defined as deleted is implicitly defined when it is odr-used (3.2) to create an object of its class type (1.8) 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} (12.6.2) 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 (7.1.5), 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. \textit{Note:} An implicitly-declared default constructor has an exception specification (15.4). An explicitly-defaulted definition might have an implicit exception specification, see 8.4. \textit{— end note}

Default constructors are called implicitly to create class objects of static, thread, or automatic storage duration (3.7.1, 3.7.2, 3.7.3) defined without an initializer (8.5), are called to create class objects of dynamic storage duration (3.7.4) created by a \textit{new-expression} in which the \textit{new-initilizer} is omitted (5.3.4), or are called when the explicit type conversion syntax (5.2.3) is used. A program is ill-formed if the default constructor for an object is implicitly used and the constructor is not accessible (Clause 11).

\textit{Note:} 12.6.2 describes the order in which constructors for base classes and non-static data members are called and describes how arguments can be specified for the calls to those constructors. \textit{— end note}

A \texttt{return} statement in the body of a constructor shall not specify a return value. The address of a constructor shall not be taken.

A functional notation type conversion (5.2.3) can be used to create new objects of its type. \textit{Note:} The syntax looks like an explicit call of the constructor. \textit{— end note} \textit{[Example:]}

\begin{verbatim}
complex zz = complex(1,2.3);
cprint( complex(7.8,1.2) );
\end{verbatim}

\textit{— end example}

An object created in this way is unnamed. \textit{Note:} 12.2 describes the lifetime of temporary objects. \textit{— end note} \textit{[Note:} Explicit constructor calls do not yield lvalues, see 3.10. \textit{— end note]}

\textit{Note:} some language constructs have special semantics when used during construction; see 12.6.2 and 12.7. \textit{— end note]

During the construction of a \texttt{const} 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 \texttt{this} pointer, the value of the object or subobject thus obtained is unspecified. \textit{[Example:}

\begin{verbatim}
struct C;
void no_opt(C*);

struct C {
  int c;
  C() : c(0) { no_opt(this); }
};

const C cobj;

void no_opt(C* cpotr) {
  int i = cobj.c * 100;  // value of cobj.c is unspecified
  cpotr->c = 1;
  cout << cobj.c * 100  // value of cobj.c is unspecified
    << '\n';
\end{verbatim}

\textit{§ 12.1 269}
12.2 Temporary objects

Temporaries of class type are created in various contexts: binding a reference to a prvalue (8.5.3), returning a prvalue (6.6.3), a conversion that creates a prvalue (4.1, 5.2.9, 5.2.11, 5.4), throwing an exception (15.1), and in some initializations (8.5). [Note: The lifetime of exception objects is described in 15.1. — end note]

Even when the creation of the temporary object is unevaluated (Clause 5) or otherwise avoided (12.8), all the semantic restrictions shall be respected as if the temporary object had been created and later destroyed. [Note: This includes accessibility (11) and whether it is deleted, for the constructor selected and for the destructor. However, in the special case of a function call used as the operand of a `decltype-specifier` (5.2.2), no temporary is introduced, so the foregoing does not apply to the prvalue of any such function call. — end note]

Example: Consider the following code:

```cpp
class X {
published:
    X(int);
    X(const X&);
    X& operator=(const X&);
    ~X();
};
class Y {
published:
    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);
}
```

An implementation might use a temporary in which to construct `X(2)` before passing it to `f()` using `X`'s copy constructor; alternatively, `X(2)` might be constructed in the space used to hold the argument. Likewise, an implementation might use a temporary in which to construct `Y(3)` before passing it to `g()` using `Y`'s move constructor; alternatively, `Y(3)` might be constructed in the space used to hold the argument. Also, a temporary might be used to hold the result of `f(X(2))` before copying it to `b` using `X`'s copy constructor; alternatively, `f()`'s result might be constructed in `b`. Likewise, a temporary might be used to hold the result of `g(Y(3))` before moving it to `c` using `Y`'s move constructor; alternatively, `g()`'s result might be constructed in `c`. On the other hand, the expression `a = f(a)` requires a temporary for the result of `f(a)`, which is then assigned to `a`. — end example]

When an implementation introduces a temporary object of a class that has a non-trivial constructor (12.1, 12.8), 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 (12.4). Temporary objects are destroyed as the last step

§ 12.2
in evaluating the full-expression (1.9) 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.

4 There are two 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. 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.

5 The second context is when a reference is bound to a temporary. The temporary to which the reference is bound or the temporary that is the complete object of a subobject to which the reference is bound persists for the lifetime of the reference except:

(5.1) — A temporary object bound to a reference parameter in a function call (5.2.2) persists until the completion of the full-expression containing the call.

(5.2) — The lifetime of a temporary bound to the returned value in a function return statement (6.6.3) is not extended; the temporary is destroyed at the end of the full-expression in the return statement.

(5.3) — A temporary bound to a reference in a new-initializer (5.3.4) persists until the completion of the full-expression containing the new-initializer. [Example:

```c
struct S { int mi; const std::pair<int,int>& mp; }; S a { 1, {2,3} }; S* p = new S{ 1, {2,3} }; // Creates dangling reference
```

— end example] [Note: This may introduce a dangling reference, and implementations are encouraged to issue a warning in such a case. — end note]

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 (3.7.1, 3.7.2, 3.7.3); 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:

```c
struct S {
 S();
 S(int);
 friend S operator+(const S&, const S&);
 ~S();
};
S obj1;
const S& cr = S(16)+S(23);
S obj2;
```

the expression `S(16) + S(23)` creates three temporaries: a first temporary `T1` to hold the result of the expression `S(16)`, a second temporary `T2` to hold the result of the expression `S(23)`, and a third temporary `T3` to hold the result of the addition of these two expressions. The temporary `T3` is then bound to the reference `cr`. It is unspecified whether `T1` or `T2` is created first. On an implementation where `T1` is created before `T2`, `T2` shall be destroyed before `T1`. The temporaries `T1` and `T2` are bound to the reference parameters of

---

116) The same rules apply to initialization of an `initializer_list` object (8.5.4) with its underlying temporary array.
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]

12.3 Conversions

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 4), for
initialization (8.5), and for explicit type conversions (5.4, 5.2.9).

2 User-defined conversions are applied only where they are unambiguous (10.2, 12.3.2). Conversions obey the
access control rules (Clause 11). Access control is applied after ambiguity resolution (3.4).

[ Note: See 13.3 for a discussion of the use of conversions in function calls as well as examples below. — end
note ]

4 At most one user-defined conversion (constructor or conversion function) is implicitly applied to a single
value.

[ Example:

```c
struct X {
    operator int();
};

struct Y {
    operator X();
};

Y a;
int b = a;       // error
    // a.operator X().operator int() not tried
int c = X(a);    // OK: a.operator X().operator int()
```
— end example ]

5 User-defined conversions are used implicitly only if they are unambiguous. A conversion function in a
derived class does not hide a conversion function in a base class unless the two functions convert to the same
type. Function overload resolution (13.3.3) selects the best conversion function to perform the conversion.

[ Example:

```c
struct X {
    operator int();
};

struct Y : X {
    operator char();
};

void f(Y& a) {
    if (a) {      // ill-formed:
        // X::operator int() or Y::operator char()
    }
}
```
— end example ]
12.3.1 Conversion by constructor

A constructor declared without the function-specifier `explicit` specifies a conversion from the types of its parameters to the type of its class. Such a constructor is called a converting constructor. [Example:

```c
struct X {
    X(int);
    X(const char*, int = 0);
    X(int, int);
};

void f(X arg) {
    X a = 1;       // a = X(1)
    X b = "Jessie"; // b = X("Jessie", 0)
    a = 2;         // a = X(2)
    f(3);          // f(X(3))
    f({1, 2});     // f(X(1, 2))
}
```

— end example]

An explicit constructor constructs objects just like non-explicit constructors, but does so only where the direct-initialization syntax (8.5) or where casts (5.2.9, 5.4) are explicitly used. A default constructor may be an explicit constructor; such a constructor will be used to perform default-initialization or value-initialization (8.5). [Example:

```c
struct Z {
    explicit Z();
    explicit Z(int);
    explicit Z(int, int);
};

Z a;                  // OK: default-initialization performed
Z a1 = 1;             // error: no implicit conversion
Z a3 = Z(1);          // OK: direct initialization syntax used
Z a2(1);              // OK: direct initialization syntax used
Z* p = new Z(1);      // OK: direct initialization syntax used
Z a4 = (Z)1;          // OK: explicit cast used
Z a5 = static_cast<Z>(1);        // OK: explicit cast used
Z a6 = { 3, 4 };      // error: no implicit conversion
```

— end example]

A non-explicit copy/move constructor (12.8) is a converting constructor. An implicitly-declared copy/move constructor is not an explicit constructor; it may be called for implicit type conversions.

12.3.2 Conversion functions

A member function of a class `X` having no parameters with a name of the form

```
conversion-function-id:
    operator conversion-type-id
conversion-type-id:
    type-specifier-seq conversion-declarator_opt
conversion-declarator:
    ptr-operator conversion-declarator_opt
```

specifies a conversion from `X` to the type specified by the `conversion-type-id`. Such functions are called conversion functions. No return type can be specified. If a conversion function is a member function, the
type of the conversion function (8.3.5) is “function taking no parameter returning conversion-type-id”. A conversion function is never used to convert a (possibly cv-qualified) object to the (possibly cv-qualified) same object type (or a reference to it), to a (possibly cv-qualified) base class of that type (or a reference to it), or to (possibly cv-qualified) void.\footnote{These conversions are considered as standard conversions for the purposes of overload resolution (13.3.3.1, 13.3.3.1.4) and therefore initialization (8.5) and explicit casts (5.2.9). A conversion to void does not invoke any conversion function (5.2.9). 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.}

[Example:

```cpp
struct X {
    operator int();
};

void f(X a) {
    int i = int(a);
    i = (int)a;
    i = a;
}
```

In all three cases the value assigned will be converted by X::operator int(). —end example]

A conversion function may be explicit (7.1.2), in which case it is only considered as a user-defined conversion for direct-initialization (8.5). Otherwise, user-defined conversions are not restricted to use in assignments and initializations. [Example:

```cpp
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
}
```

—end example]

The conversion-type-id shall not represent a function type nor an array type. The conversion-type-id in a conversion-function-id is the longest possible sequence of conversion-declarators. [Note: This prevents ambiguities between the declarator operator * and its expression counterparts. [Example:

```cpp
&ac.operator int*; // syntax error:
// parsed as: &ac.operator int* i
// not as: &ac.operator int*i
```

The * is the pointer declarator and not the multiplication operator. —end example| —end note]

Conversion functions are inherited.
Conversion functions can be virtual.

Conversion functions cannot be declared `static`.

A conversion function template shall not have a deduced return type (7.1.6.4).

### 12.4 Destructors

In a declaration of a destructor, the declarator is a function declarator (8.3.5) of the form

\[ \text{ptr-declarator ( parameter-declaration-clause ) exception-specification\text{opt} attribute-specifier-seq\text{opt} } \]

where the `ptr-declarator` consists solely of an `id-expression`, an optional `attribute-specifier-seq`, and optional surrounding parentheses, and the `id-expression` has one of the following forms:

1. In a `member-declaration` that belongs to the `member-specification` of a class but is not a friend declaration (11.3), the `id-expression` is `~class-name` and the `class-name` is the injected-class-name (Clause 9) 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 `~class-name` and the `class-name` names the current instantiation (14.6.2.1) of the immediately-enclosing class template; or
3. In a declaration at namespace scope or in a friend declaration, the `id-expression` is `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 (8.3.5). In a destructor declaration, each `decl-specifier` of the optional `decl-specifier-seq` shall be `friend`, `inline`, or `virtual`.

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 (7.1.6.1) are not applied on an object under destruction. They stop being in effect when the destructor for the most derived object (1.8) starts.

A declaration of a destructor that does not have an `exception-specification` has the same exception specification as if had been implicitly declared (15.4).

If a class has no user-declared destructor, a destructor is implicitly declared as defaulted (8.4). An implicitly-declared destructor is an `inline public` member of its class.

A defaulted destructor for a class `X` is defined as deleted if:

1. `X` is a union-like class that has a variant member with a non-trivial destructor,
2. any potentially constructed subobject has class type `M` (or array thereof) and `M` has a deleted destructor or a destructor that is inaccessible from the defaulted destructor,
3. or, for a virtual destructor, lookup of the non-array deallocation function results in an ambiguity or in a function that is deleted or inaccessible from the defaulted destructor.

A destructor is trivial if it is not user-provided and if:

1. the destructor is not `virtual`,
2. all of the direct base classes of its class have trivial destructors, and
3. for all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor.

§ 12.4
Otherwise, the destructor is non-trivial.

6 A destructor that is defaulted and not defined as deleted is implicitly defined when it is odr-used (3.2) to destroy an object of its class type (3.7) or when it is explicitly defaulted after its first declaration.

7 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.

8 After executing the body of the destructor and destroying any automatic objects allocated within the body, a destructor for class \textit{X} calls the destructors for \textit{X}'s direct non-variant non-static data members, the destructors for \textit{X}'s direct base classes and, if \textit{X} is the type of the most derived class (12.6.2), its destructor calls the destructors for \textit{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 12.6.2). A \textit{return} statement (6.6.3) 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 12.6).

9 A destructor can be declared \texttt{virtual} (10.3) or pure virtual (10.4); 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.

[\textit{Note:} some language constructs have special semantics when used during destruction; see 12.7. — end note]

10 A destructor is invoked implicitly

\begin{itemize}
\item[(11.1)] — for a constructed object with static storage duration (3.7.1) at program termination (3.6.3),
\item[(11.2)] — for a constructed object with thread storage duration (3.7.2) at thread exit,
\item[(11.3)] — for a constructed object with automatic storage duration (3.7.3) when the block in which an object is created exits (6.7),
\item[(11.4)] — for a constructed temporary object when its lifetime ends (12.2).
\end{itemize}

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 \texttt{delete-expression} (5.3.5) for a constructed object allocated by a \texttt{new-expression} (5.3.4); the context of the invocation is the \texttt{delete-expression}. [\textit{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 5.3.4 and 12.6.2. A program is ill-formed if a destructor that is potentially invoked is deleted or not accessible from the context of the invocation.

12 At the point of definition of a virtual destructor (including an implicit definition (12.8)), the non-array deallocation function is determined as if for the expression \texttt{delete this} appearing in a non-virtual destructor of the destructor’s class (see 5.3.5). If the lookup fails or if the deallocation function has a deleted definition (8.4), the program is ill-formed. [\textit{Note:} This assures that a deallocation function corresponding to the dynamic type of an object is available for the \texttt{delete-expression} (12.5). — end note]

13 In an explicit destructor call, the destructor name appears as a \texttt{-} followed by a \texttt{type-name} or \texttt{decltype-specifier} that denotes the destructor’s class type. The invocation of a destructor is subject to the usual rules for member functions (9.3); 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. [\textit{Note:} invoking \texttt{delete} on a null pointer does not call the destructor; see 5.3.5. — end note] [Example:]

\begin{verbatim}
struct B {
  virtual ~B() { }
}
\end{verbatim}
struct D : B {
-~D() {} 
};
D D_object;
typedef B B_alias;
B* B_ptr = &D_object;

void f() {
  D_object.B::~B(); // calls B's destructor
  B_ptr->~B(); // calls D's destructor
  B_ptr->B_alias::~B(); // calls B's destructor
  B_ptr->B_alias::~B_alias(); // calls B's destructor
  B_ptr->B::~B(); // calls B's destructor
}

—end example] [ Note: An explicit destructor call must always be written using a member access operator (5.2.5) or a qualified-id (5.1); in particular, the unary-expression ~X() in a member function is not an explicit destructor call (5.3.1). — end note]

[ Note: explicit calls of destructors are rarely needed. One use of such calls is for objects placed at specific addresses using a placement new-expression. Such use of explicit placement and destruction of objects can be necessary to cope with dedicated hardware resources and for writing memory management facilities. For example,

```c
void* operator new(std::size_t, void* p) { return p; }
struct X {
    X(int);
    ~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]

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 (3.8). [ 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 ]

[ Note: the notation for explicit call of a destructor can be used for any scalar type name (5.2.4). 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]

§ 12.4
12.5 Free store

1 Any allocation function for a class T is a static member (even if not explicitly declared static).

2 [Example:

```cpp
class Arena;
struct B {
    void* operator new(std::size_t, Arena*);
};
struct D1 : B {
};

Arena* ap;
void foo(int i) {
    new (ap) D1; // calls B::operator new(std::size_t, Arena*)
    new D1[i]; // calls ::operator new[](std::size_t)
    new D1; // ill-formed: ::operator new(std::size_t) hidden
}
```
—end example]

3 When an object is deleted with a delete-expression (5.3.5), 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 (3.7.4.2).

4 Class-specific deallocation function lookup is a part of general deallocation function lookup (5.3.5) 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 (12.4).\footnote{A similar provision is not needed for the array version of operator delete because 5.3.5 requires that in this situation, the static type of the object to be deleted be the same as its dynamic type.} Otherwise, if the delete-expression is used to deallocate an object of class T or array thereof, the 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 (5.3.5) 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.

5 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]

6 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();
```

\footnote{118}
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,

    struct B {
        virtual ~B();
        void operator delete[](void*, std::size_t);
    };

    struct D : B {
        void operator delete[](void*, std::size_t);
    };

    void f(int i) {
        D* dp = new D[i];
        delete[] dp;  // uses D::operator delete[](void*, std::size_t)
        B* bp = new D[i];
        delete[] bp;  // undefined behavior
    }

—end note]

7 Access to the deallocation function is checked statically. 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 exception-specification, it has a non-throwing exception specification (15.4). —end note]

12.6 Initialization [class.init]

1 When no initializer is specified for an object of (possibly cv-qualified) class type (or array thereof), or the initializer has the form (), the object is initialized as specified in 8.5.

2 An object of class type (or array thereof) can be explicitly initialized; see 12.6.1 and 12.6.2.

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 8.3.4. [Note: Destructors for the array elements are called in reverse order of their construction. —end note]
12.6.1 Explicit initialization

An object of class type can be initialized with a parenthesized expression-list, where the expression-list is construed as an argument list for a constructor that is called to initialize the object. Alternatively, a single assignment-expression can be specified as an initializer using the = form of initialization. Either direct-initialization semantics or copy-initialization semantics apply; see 8.5. [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 (13.5.3) has no effect on initialization. — end note]

An object of class type can also be initialized by a braced-init-list. List-initialization semantics apply; see 8.5 and 8.5.4. [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 8.5.1. — end note]

[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 12.6 and 8.5). — end note]

[Note: the order in which objects with static or thread storage duration are initialized is described in 3.6.2 and 6.7. — end note]
12.6.2 Initializing bases and members

In the definition of a constructor for a class, initializers for direct and virtual base subobjects and non-static data members can be specified by a *ctor-initializer*, which has the form

```
celor-initializer:
    : mem-initializer-list
mem-initializer-list:
    mem-initializer ... opt
    mem-initializer-list , mem-initializer ... opt
mem-initializer:
    mem-initializer-id ( expression-list opt )
    mem-initializer-id braced-init-list
mem-initializer-id:
    class-or-decltype
    identifier
```

In a *mem-initializer-id* an initial unqualified *identifier* is looked up in the scope of the constructor’s class and, if not found in that scope, it is looked up in the scope containing the constructor’s definition. [Note: If the constructor’s class contains a member with the same name as a direct or virtual base class of the class, a *mem-initializer-id* naming the member or base class and composed of a single identifier refers to the class member. A *mem-initializer-id* for the hidden base class may be specified using a qualified name. —end note] Unless the *mem-initializer-id* names the constructor’s class, a non-static data member of the constructor’s class, or a direct or virtual base of that class, the *mem-initializer* is ill-formed.

A *mem-initializer-list* can initialize a base class using any *class-or-decltype* that denotes that base class type. [Example:

```c
struct A { A(); };  
typedef A global_A;  
struct B { };  
struct C: public A, public B { C(); };  
C::C(): global_A() {}  // mem-initializer for base A
```

—end example]

If a *mem-initializer-id* is ambiguous because it designates both a direct non-virtual base class and an inherited virtual base class, the *mem-initializer* is ill-formed. [Example:

```c
struct A { A(); };  
struct B: public virtual A { };  
struct C: public A, public B { C(); };  
C::C(): A() {}  // ill-formed: which A?
```

—end example]

A *ctor-initializer* may initialize a variant member of the constructor’s class. If a *ctor-initializer* specifies more than one *mem-initializer* for the same member or for the same base class, the *ctor-initializer* is ill-formed.

A *mem-initializer-list* can delegate to another constructor of the constructor’s class using any *class-or-decltype* that denotes the constructor’s class itself. If a *mem-initializer-id* designates the constructor’s class, it shall be the only *mem-initializer*; the constructor is a delegating constructor, and the constructor selected by the *mem-initializer* is the target constructor. The principal constructor is the first constructor invoked in the construction of an object (that is, not a target constructor for that object’s construction). 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 is required. [Example:

```c
struct C {
```

§ 12.6.2
C( int ) { } // #1: non-delegating constructor
C(): C(42) { } // #2: delegates to #1
C( char c ): C(42.0) { } // #3: ill-formed due to recursion with #4
C( double d ): C('a') { } // #4: ill-formed due to recursion with #3
;
— end example |

7 The *expression-list* or *braced-init-list* in a *mem-initializer* is used to initialize the designated subobject (or, in the case of a delegating constructor, the complete class object) according to the initialization rules of 8.5 for direct-initialization.

[Example:

```cpp
struct B1 { B1(int); /* ... */ };  
struct B2 { B2(int); /* ... */ };  
struct D : B1, B2 {
    D(int);
    B1 b;
    const int c;
};
```

```cpp
D::D(int a) : B2(a+1), B1(a+2), c(a+3), b(a+4) {
    /* ... */
}
D d(10);
— end example |

The initialization performed by each *mem-initializer* constitutes a full-expression. Any expression in a *mem-initializer* is evaluated as part of the full-expression that performs the initialization. A *mem-initializer* where the *mem-initializer-id* denotes a virtual base class is ignored during execution of a constructor of any class that is not the most derived class.

8 A temporary expression bound to a reference member in a *mem-initializer* is ill-formed. [Example:

```cpp
struct A {
    A() : v(42) { } // error
    const int& v;
};
— end example |

9 In a non-delegating constructor, if a given potentially constructed subobject is not designated by a *mem-initializer-id* (including the case where there is no *mem-initializer-list* because the constructor has no *ctor-initializer*), then

(9.1) — if the entity is a non-static data member that has a *brace-or-equal-initializer* and either
(9.1.1) — the constructor’s class is a union (9.5), and no other variant member of that union is designated by a *mem-initializer-id* or
(9.1.2) — the constructor’s class is not a union, and, if the entity is a member of an anonymous union, no other member of that union is designated by a *mem-initializer-id*,
the entity is initialized as specified in 8.5;
(9.2) — otherwise, if the entity is an anonymous union or a variant member (9.5), no initialization is performed;
(9.3) — otherwise, the entity is default-initialized (8.5).

[Note: An abstract class (10.4) 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

§ 12.6.2
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 \textit{compound-statement} of the body of the constructor, the member has
an indeterminate value. \textit{— end note} [Example:

```cpp
struct A {
    A();
};

struct B {
    B(int);
};

struct C {
    C() { } // initializes members as follows:
    A a;    // OK: calls A::A()
    const B b; // error: B has no default constructor
    int i;    // OK: i has indeterminate value
    int j = 5; // OK: j has the value 5
};
```

\textit{— end example}]

10 If a given non-static data member has both a \textit{brace-or-equal-initializer} and a \textit{mem-initializer}, the initialization
specified by the \textit{mem-initializer} is performed, and the non-static data member’s \textit{brace-or-equal-initializer}
is ignored. [Example: Given

```cpp
struct A {
    int i = /* some integer expression with side effects */ ;
    A(int arg) : i(arg) { }
    // ...
};
```

the \( A(\text{int}) \) constructor will simply initialize \( i \) to the value of \( \text{arg} \), and the side effects in \( i \)’s \textit{brace-or-equal-initializer}
will not take place. \textit{— end example}]

11 A temporary expression bound to a reference member from a \textit{brace-or-equal-initializer} is ill-formed. [Example:
Given

```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
```

\textit{— end example}]

12 In a non-delegating constructor, the destructor for each potentially constructed subobject of class type is
potentially invoked (12.4). \textit{[Note: This provision ensures that destructors can be called for fully-constructed
sub-objects in case an exception is thrown (15.2). \textit{— end note}]

13 In a non-delegating constructor, initialization proceeds in the following order:

\textit{[13.1]} \textit{— First, and only for the constructor of the most derived class (1.8), 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 \textit{base-specifier-list}.}
(13.2) — Then, direct base classes are initialized in declaration order as they appear in the base-specifier-list (regardless of the order of the mem-initializers).

(13.3) — Then, non-static data members are initialized in the order they were declared in the class definition (again regardless of the order of the mem-initializers).

(13.4) — Finally, the compound-statement of the constructor body is executed.

[Note: The declaration order is mandated to ensure that base and member subobjects are destroyed in the reverse order of initialization. — end note]

14 [Example:

```cpp
struct V {
    V();
    V(int);
};

struct A : virtual V {
    A();
    A(int);
};

struct B : virtual V {
    B();
    B(int);
};

struct C : A, B, virtual V {
    C();
    C(int);
};

A::A(int i) : V(i) { /* ... */ }
B::B(int i) { /* ... */ }
C::C(int i) { /* ... */ }

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, 10.3) can be called for an object under construction. Similarly, an object under construction can be the operand of the typeid operator (5.2.8) or of a dynamic_cast (5.2.7). 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 result of the operation is undefined. [Example:

```cpp
class A {
public:
    A(int);
};

class B : public A {
    int j;
public:
    int f();
    // undefined: calls member function
    B() : A(f()), // undefined: calls member function
         j(f()) { } // well-defined: bases are all initialized
};

class C {
public:
    C(int);
};

class D : public B, C {
    int i;
public:
    D() : C(f()), // undefined: calls member function
          i(f()) { } // well-defined: bases are all initialized
};

—end example]

17 [Note: 12.7 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 (14.5.3) 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]
### 12.7 Construction and destruction

1. For an object with a non-trivial constructor, referring to any non-static member or base class of the object before the constructor begins execution results in undefined behavior. For an object with a non-trivial destructor, referring to any non-static member or base class of the object after the destructor finishes execution results in undefined behavior. 

   ```
   struct X { int i; };  // non-trivial
   struct Y : X { Y(); };  // non-trivial
   struct A { int a; };  // non-trivial
   struct B : public A { int j; Y y; };  // non-trivial
   
   extern B bobj;
   B* pb = &bobj;  // OK
   int* p1 = &bobj.a;  // undefined, refers to base class member
   int* p2 = &bobj.y.i;  // undefined, refers to member’s member
   
   A* pa = &bobj;  // undefined, upcast to a base class type
   B bobj;  // definition of bobj
   extern X xobj;
   int* p3 = &xobj.i;  //OK, X is a trivial class
   X xobj;
   ```

2. For another example,

   ```
   struct W { int j; };  // OK
   struct X : public virtual W { };  // undefined, refers to base class member
   struct Y { int* p;
   X x;
   Y() : p(&x.j) {  // undefined, x is not yet constructed
   }
   
   extern X xobj;
   int* p3 = &xobj.i;  //OK, X is a trivial class
   X xobj;
   ```

3. To explicitly or implicitly convert a pointer (a glvalue) referring to an object of class `X` to a pointer (reference) to a direct or indirect base class `B` of `X`, the construction of `X` and the construction of all of its direct or indirect bases that directly or indirectly derive from `B` shall have started and the destruction of these classes shall not have completed, otherwise the conversion results in undefined behavior. To form a pointer to (or access the value of) a direct non-static member of an object `obj`, the construction of `obj` shall have started and its destruction shall not have completed, otherwise the computation of the pointer value (or accessing the member value) results in undefined behavior. 

   ```
   struct A { };  // undefined: upcast from E* to A*
   struct B : virtual A { };  // might use path E* -> D* -> A*
   struct C : B { };  // but D is not constructed
   struct D : virtual A { D(A*); };  // D(C* this), // defined:
   struct X { X(A*); };  // E* -> C* defined because E() has started
   ```
Member functions, including virtual functions (10.3), can be called during construction or destruction (12.6.2). 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 (5.2.5) 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:

```cpp
struct V {
    virtual void f();
    virtual void g();
};

struct A : virtual V {
    virtual void f();
};

struct B : virtual V {
    virtual void g();
    B(V*, A*);  // calls V::f, not A::f
};

struct D : A, B {
    virtual void f();
    virtual void g();
    D() : B((A*)this, this) { }
};

B::B(V* v, A* a) {
    f();  // calls V::f, not A::f
    g();  // calls B::g, not D::g
    v->g();  // v is base of B, the call is well-defined, calls B::g
    a->f();  // undefined behavior, a’s type not a base of B
}
```

—end example]
6 **dynamic_casts** (5.2.7) can be used during construction or destruction (12.6.2). When a **dynamic_cast** is used in a constructor (including the **mem-initializer** or **brace-or-equal-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
    typeid(*a);  // undefined behavior: type A not a base of B
    dynamic_cast<B*>(v); // well-defined: v of type V*, V base of B
    // results in B*
    dynamic_cast<B*>(a); // undefined behavior,
    // a has type A*, A not a base of B
}
```

—end example]

12.8 Copying and moving class objects

A class object can be copied or moved in two ways: by initialization (12.1, 8.5), including for function argument passing (5.2.2) and for function value return (6.6.3); and by assignment (5.18). Conceptually, these two operations are implemented by a copy/move constructor (12.1) and copy/move assignment operator (13.5.3).

2 A non-template constructor for class `X` is a copy constructor if its first parameter is of type `X&`, `const X&`, `volatile X&` or `const volatile X&`, and either there are no other parameters or else all other parameters have default arguments (8.3.6). [Example: `X::X(const X&)` and `X::X(X&, int=1)` are copy constructors.

```cpp
struct X {
    X(int);
    X(const X&, int = 1);
};
X a(1);       // calls X(int);
X b(a, 0);    // calls X(const X&, int);
X c = b;      // calls X(const X&, int);
```

—end example]
A non-template constructor for class X is a move constructor if its first parameter is of type X&&, const X&&, volatile X&&, or const volatile X&&, and either there are no other parameters or else all other parameters have default arguments (8.3.6). [Example: Y::Y(Y&&) is a move constructor.

```cpp
struct Y {
    Y(const Y&);  // copy constructor
    Y(Y&&);      // move constructor
};
```

Example: Y d(f(i)); // calls Y(Y&&)  
Y e = d;       // calls Y(const Y&)

— end example]

4 [Note: All forms of copy/move constructor may be declared for a class. [Example:

```cpp
struct X {
    X(const X&);  // default constructor
    X(X&);       // copy constructor with a nonconst parameter
    X(X&&);     // move constructor
    X(const X&&); // move constructor
};
```

— end example] — end note]

5 [Note: If a class X only has a copy constructor with a parameter of type X&, an initializer of type const X or volatile X cannot initialize an object of type (possibly cv-qualified) X. [Example:

```cpp
struct X {
    X();             // default constructor
    X(X&);          // copy constructor with a nonconst parameter
    const X cx;
    X x = cx;       // error: X::X(X&) cannot copy cx into x
};
```

— end example] — end note]

6 A declaration of a constructor for a class X is ill-formed if its first parameter is of type (optionally cv-qualified) X and either there are no other parameters or else all other parameters have default arguments. A member function template is never instantiated to produce such a constructor signature. [Example:

```cpp
struct S {
    template<typename T> S(T);
    S();
};
```

S g;

void h() {  // does not instantiate the member template to produce S::S<S>(S);
    S a(g);  // uses the implicitly declared copy constructor
}

— end example]

7 If the class definition does not explicitly declare a copy constructor, one is declared implicitly. If the class definition declares a move constructor or move assignment operator, the implicitly declared copy constructor is defined as deleted; otherwise, it is defined as defaulted (8.4). 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 $const\ M\&$ or $const\ volatile\ M\&$.\(^\text{119}\)  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, one will be implicitly declared as defaulted if and only if

1. $X$ does not have a user-declared copy constructor,
2. $X$ does not have a user-declared copy assignment operator,
3. $X$ does not have a user-declared move assignment operator, and
4. $X$ does not have a user-declared destructor.

[Note: When the move constructor is not implicitly declared or explicitly supplied, expressions that otherwise would have invoked the move constructor may instead invoke a copy constructor. — end note]

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 (8.4.3) if $X$ has:

1. a variant member with a non-trivial corresponding constructor and $X$ is a union-like class,
2. a potentially constructed subobject type $M$ (or array thereof) that cannot be copied/moved because overload resolution (13.3), as applied to $M$'s corresponding constructor, results in an ambiguity or a function that is deleted or inaccessible from the defaulted constructor,
3. any potentially constructed subobject of a type with a destructor that is deleted or inaccessible from the defaulted constructor, or,
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 (13.3, 13.4). [Note: A deleted move constructor would otherwise interfere with initialization from an rvalue which can use the copy constructor instead. — end note]

A copy/move constructor for class $X$ is trivial if it is not user-provided, its parameter-type-list is equivalent to the parameter-type-list of an implicit declaration, and if

1. $X$ has no virtual functions (10.3) and no virtual base classes (10.1), and
2. $X$ has no non-static data members of volatile-qualified type, and
3. the constructor selected to copy/move each direct base class subobject is trivial, and
4. 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;

\(^{119}\) This implies that the reference parameter of the implicitly-declared copy constructor cannot bind to a volatile lvalue; see C.1.9.
otherwise the copy/move constructor is non-trivial.

13 A copy/move constructor that is defaulted and not defined as deleted is implicitly defined if it is odr-used (3.2) or when it is explicitly defaulted after its first declaration. [Note: The copy/move constructor is implicitly defined even if the implementation elided its odr-use (3.2, 12.2). — end note] If the implicitly-defined constructor would satisfy the requirements of a constexpr constructor (7.1.5), the implicitly-defined constructor is constexpr.

14 Before the defaulted copy/move constructor for a class is implicitly defined, all non-user-provided copy/move constructors for its potentially constructed subobjects shall have been implicitly defined. [Note: An implicitly-declared copy/move constructor has an implied exception specification (15.4). — end note]

15 The implicitly-defined copy/move constructor for a non-union class X performs a memberwise copy/move of its bases and members. [Note: brace-or-equal-initializers of non-static data members are ignored. See also the example in 12.6.2. — end note] The order of initialization is the same as the order of initialization of bases and members in a user-defined constructor (see 12.6.2). Let x be either the parameter of the constructor or, for the move constructor, an xvalue referring to the parameter. Each base or non-static data member is copied/moved in the manner appropriate to its type:

(15.1) — if the member is an array, each element is direct-initialized with the corresponding subobject of x;

(15.2) — if a member m has rvalue reference type T&&, it is direct-initialized with static_cast<T&&>(x.m);

(15.3) — 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 12.6.2).

16 The implicitly-defined copy/move constructor for a union X copies the object representation (3.9) of X.

17 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&.120 [Note: An overloaded assignment operator must be declared to have only one parameter; see 13.5.3. — 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:

```cpp
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]

18 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 (8.4). The latter case is deprecated if the class has a user-declared copy constructor or a user-declared destructor. The implicitlydeclared copy assignment operator for a class X will have the form

120) 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.
X& X::operator=(const X&)

if

(18.1) 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

(18.2) 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.121

Otherwise, the implicitly-declared copy assignment operator will have the form

X& X::operator=(X&)

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 13.5.3. — 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

(20.1) X does not have a user-declared copy constructor,

(20.2) X does not have a user-declared move constructor,

(20.3) X does not have a user-declared copy assignment operator, and

(20.4) X does not have a user-declared destructor.

[Example: The class definition

```cpp
struct S {
    int a;
    S& operator=(const S&) = default;
};
```

will not have a default move assignment operator implicitly declared because the copy assignment operator has been user-declared. The move assignment operator may be explicitly defaulted.

```cpp
struct S {
    int a;
    S& operator=(const S&) = default;
    S& operator=(S&&) = default;
};
```

— end example]

The implicitly-declared move assignment operator for a class X will have the form

X& X::operator=(X&&);

The implicitly-declared copy/move assignment operator for class X has the return type X&; it returns the object for which the assignment operator is invoked, that is, the object assigned to. An implicitly-declared copy/move assignment operator is an inline public member of its class.

A defaulted copy/move assignment operator for class X is defined as deleted if X has:

121) This implies that the reference parameter of the implicitly-declared copy assignment operator cannot bind to a volatile lvalue; see C.1.9.
A defaulted move assignment operator that is defined as deleted is ignored by overload resolution (13.3, 13.4).

Because a copy/move assignment operator is implicitly declared for a class if not declared by the user, a base class copy/move assignment operator is always hidden by the corresponding assignment operator of a derived class (13.5.3). A using-declaration (7.3.3) 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.

A copy/move assignment operator for class X is trivial if it is not user-provided, its parameter-type-list is equivalent to the parameter-type-list of an implicit declaration, and if

25.1 X has no virtual functions (10.3) and no virtual base classes (10.1), and
25.2 X has no non-static data members of volatile-qualified type, and
25.3 the assignment operator selected to copy/move each direct base class subobject is trivial, and
25.4 for each non-static data member of X that is of class type (or array thereof), the assignment operator selected to copy/move that member is trivial;
otherwise the copy/move assignment operator is non-trivial.

A copy/move assignment operator for a class X that is defaulted and not defined as deleted is implicitly defined when it is odr-used (3.2) (e.g., when it is selected by overload resolution to assign to an object of its class type) or when it is explicitly defaulted after its first declaration. The implicitly-defined copy/move assignment operator is constexpr if

26.1 X is a literal type, and
26.2 the assignment operator selected to copy/move each direct base class subobject is a constexpr function, and
26.3 for each non-static data member of X that is of class type (or array thereof), the assignment operator selected to copy/move that member is a constexpr function.

Before the defaulted copy/move assignment operator for a class is implicitly defined, all non-user-provided copy/move assignment operators for its direct base classes and its non-static data members shall have been implicitly defined. [Note: An implicitly-declared copy/move assignment operator has an implied exception specification (15.4). —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 xvalue referring to the parameter. Each subobject is assigned in the manner appropriate to its type:

§ 12.8
— if the subobject is of class type, as if by a call to `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.  

```cpp
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`.  

29  The implicitly-defined copy assignment operator for a union `X` copies the object representation (3.9) of `X`.

30  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 11).  [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]

31  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, and the destruction of that object 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):

(31.1) — in a `return` statement in a function with a class return type, when the expression is the name of a non-volatile automatic object (other than a function or catch-clause parameter) with the same cv-unqualified type as the function return type, the copy/move operation can be omitted by constructing the automatic object directly into the function’s return value

(31.2) — in a `throw-expression` (5.17), when the operand is the name of a non-volatile automatic object (other than a function or catch-clause parameter) whose scope does not extend beyond the end of the innermost enclosing `try-block` (if there is one), the copy/move operation from the operand to the exception object (15.1) can be omitted by constructing the automatic object directly into the exception object

(31.3) — when a temporary class object that has not been bound to a reference (12.2) would be copied/moved to a class object with the same cv-unqualified type, the copy/move operation can be omitted by constructing the temporary object directly into the target of the omitted copy/move

(31.4) — when the `exception-declaration` of an exception handler (Clause 15) declares an object of the same type (except for cv-qualification) as the exception object (15.1), the copy operation can be omitted by treating the `exception-declaration` as an alias for the exception object if the meaning of the program will be unchanged except for the execution of constructors and destructors for the object declared by the `exception-declaration`.  [Note: There cannot be a move from the exception object because it is always an lvalue. —end note]

122) 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.
Example:

```cpp
class Thing {
public:
    Thing();
    ~Thing();
    Thing(const Thing&);
};

Thing f() {
    Thing t;
    return t;
}

Thing t2 = f();
```

Here the criteria for elision can be combined to eliminate two calls to the copy constructor of class `Thing`: the copying of the local automatic object `t` into the temporary object for the return value of function `f()` and the copying of that temporary object into 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 temporary object to `t2` that is elided. —end example]}

When the criteria for elision of a copy/move operation are met, but not for an exception-declaration, and the object to be copied is designated by an lvalue, or when the expression in a `return` statement 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, 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]}

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: Thing(Thing&&) used (or elided) to construct t2
```

—end example]}

§ 12.8 295
12.9 Inheriting constructors

A *using-declaration* (7.3.3) that names a constructor implicitly declares a set of *inheriting constructors*. The candidate set of inherited constructors from the class \( X \) named in the *using-declaration* consists of actual constructors and notional constructors that result from the transformation of defaulted parameters and ellipsis parameter specifications as follows:

1. for each non-template constructor of \( X \), the constructor that results from omitting any ellipsis parameter specification, and
2. for each non-template constructor of \( X \) that has at least one parameter with a default argument, the set of constructors that results from omitting any ellipsis parameter specification and successively omitting parameters with a default argument from the end of the parameter-type-list, and
3. for each constructor template of \( X \), the constructor template that results from omitting any ellipsis parameter specification, and
4. for each constructor template of \( X \) that has at least one parameter with a default argument, the set of constructor templates that results from omitting any ellipsis parameter specification and successively omitting parameters with a default argument from the end of the parameter-type-list.

The *constructor characteristics* of a constructor or constructor template are

1. the template parameter list (14.1), if any,
2. the parameter-type-list (8.3.5), and
3. absence or presence of *explicit* (12.3.1).

For each non-template constructor in the candidate set of inherited constructors other than a constructor having no parameters or a copy/move constructor having a single parameter, a constructor is implicitly declared with the same constructor characteristics unless there is a user-declared constructor with the same signature in the complete class where the *using-declaration* appears or the constructor would be a default, copy, or move constructor for that class. Similarly, for each constructor template in the candidate set of inherited constructors, a constructor template is implicitly declared with the same constructor characteristics unless there is an equivalent user-declared constructor template (14.5.6.1) in the complete class where the *using-declaration* appears. [Note: Default arguments are not inherited. An exception specification is implied as specified in 15.4. — end note]

A constructor so declared has the same access as the corresponding constructor in \( X \). It is *constexpr* if the user-written constructor (see below) would satisfy the requirements of a *constexpr* constructor (7.1.5). It is deleted if the corresponding constructor in \( X \) is deleted (8.4.3) or if a defaulted default constructor (12.1) would be deleted, except that the construction of the direct base class \( X \) is not considered in the determination. An inheriting constructor shall not be explicitly instantiated (14.7.2) or explicitly specialized (14.7.3).

[Note: Default and copy/move constructors may be implicitly declared as specified in 12.1 and 12.8. — end note]

[Example:

```c
struct B1 {
    B1(int);
};

struct B2 {
    B2(int = 13, int = 42);
};
```]

§ 12.9
struct D1 : B1 {
    using B1::B1;
};

struct D2 : B2 {
    using B2::B2;
};

The candidate set of inherited constructors in D1 for B1 is

(6.1) — B1(const B1&)
(6.2) — B1(B1&&)
(6.3) — B1(int)

The set of constructors present in D1 is

(6.4) — D1(), implicitly-declared default constructor, ill-formed if odr-used
(6.5) — D1(const D1&), implicitly-declared copy constructor, not inherited
(6.6) — D1(D1&&), implicitly-declared move constructor, not inherited
(6.7) — D1(int), implicitly-declared inheriting constructor

The candidate set of inherited constructors in D2 for B2 is

(6.8) — B2(const B2&)
(6.9) — B2(B2&&)
(6.10) — B2(int = 13, int = 42)
(6.11) — B2(int = 13)
(6.12) — B2()

The set of constructors present in D2 is

(6.13) — D2(), implicitly-declared default constructor, not inherited
(6.14) — D2(const D2&), implicitly-declared copy constructor, not inherited
(6.15) — D2(D2&&), implicitly-declared move constructor, not inherited
(6.16) — D2(int, int), implicitly-declared inheriting constructor
(6.17) — D2(int), implicitly-declared inheriting constructor

— end example

[Note: If two using-declarations declare inheriting constructors with the same signatures, the program is ill-formed (9.2, 13.1), because an implicitly-declared constructor introduced by the first using-declaration is not a user-declared constructor and thus does not preclude another declaration of a constructor with the same signature by a subsequent using-declaration. [Example:
struct B1 {
    B1(int);
};

struct B2 {
    B2(int);
};

struct D1 : B1, B2 {
    using B1::B1;
    using B2::B2;
};      // ill-formed: attempts to declare D1(int) twice

struct D2 : B1, B2 {
    using B1::B1;
    using B2::B2;
    D2(int);    // OK: user declaration supersedes both implicit declarations
};

—end example —end note

An inheriting constructor for a class is implicitly defined when it is odr-used (3.2) to create an object of its class type (1.8). An implicitly-defined inheriting constructor performs the set of initializations of the class that would be performed by a user-written inline constructor for that class with a mem-initializer-list whose only mem-initializer has a mem-initializer-id that names the base class denoted in the nested-name-specifier of the using-declaration and an expression-list as specified below, and where the compound-statement in its function body is empty (12.6.2). If that user-written constructor would be ill-formed, the program is ill-formed. Each expression in the expression-list is of the form static_cast<T&&>(p), where p is the name of the corresponding constructor parameter and T is the declared type of p.

[Example:

struct B1 {
    B1(int) { }
};

struct B2 {
    B2(double) { }
};

struct D1 : B1 {
    using B1::B1; // implicitly declares D1(int)
    int x;
};

void test() {
    D1 d(6);     // OK: d.x is not initialized
    D1 e;        // error: D1 has no default constructor
}

struct D2 : B2 {
    using B2::B2; // OK: implicitly declares D2(double)
    B1 b;
};

D2 f(1.0); // error: B1 has no default constructor

§ 12.9
template< class T >
struct D : T {
    using T::T;  // declares all constructors from class T
    ~D() { std::clog << "Destroying wrapper" << std::endl; }
};

Class template D wraps any class and forwards all of its constructors, while writing a message to the standard log whenever an object of class D is destroyed. — end example]
13 Overloading

When two or more different declarations are specified for a single name in the same scope, that name is said to be overloaded. By extension, two declarations in the same scope that declare the same name but with different types are called overloaded declarations. Only function and function template declarations can be overloaded; variable and type declarations cannot be overloaded.

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 13.3. [Example:

```cpp
double abs(double);
int abs(int);

abs(1); // calls abs(int);
abs(1.0); // calls abs(double);
```

— end example]

13.1 Overloadable declarations

Not all function declarations can be overloaded. Those that cannot be overloaded are specified here. A program is ill-formed if it contains two such non-overloadable declarations in the same scope. [Note: This restriction applies to explicit declarations in a scope, and between such declarations and declarations made through a using-declaration (7.3.3). 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]

Certain function declarations cannot be overloaded:

1. Function declarations that differ only in the return type cannot be overloaded.
2. Member function declarations with the same name and the same parameter-type-list cannot be overloaded if any of them is a static member function declaration (9.4). 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 (13.3.1) 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:

```cpp
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
};
```

§ 13.1
Member function declarations with the same name and the same parameter-type-list 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 (8.3.5). [Example:

```cpp
class Y {
  void h() &;  // OK
  void h() const &;  // OK, all declarations have a ref-qualifier
  void h() &&;  // OK, all declarations have a ref-qualifier
  void i() &;
  void i() const;  // ill-formed, prior declaration of i
                   // has a ref-qualifier
};
```

— end example]

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 (7.1.3). [Example:

```cpp
typedef int Int;

void f(int i);
void f(Int i);  // OK: redeclaration of f(int)
void f(int i) { /* ... */ }
void f(Int i) { /* ... */ }  // error: redefinition of f(int)
```

— end example]

Enumerations, on the other hand, are distinct types and can be used to distinguish overloaded function declarations. [Example:

```cpp
enum E { a };

void f(int i) { /* ... */ }
void f(E i) { /* ... */ }
```

— end example]

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 (8.3.5). Only the second and subsequent array dimensions are significant in parameter types (8.3.4). [Example:

```cpp
int f(char*);
int f(char[]);  // same as f(char*)
int f(char[7]);  // same as f(char*)
int f(char[9]);  // same as f(char*)

int g(char*)[10]);
int g(char[5][10]);  // same as g(char(*)[10])
int g(char[7][10]);  // same as g(char(*)[10])
int g(char[20]);  // different from g(char(*)[10])
```

— end example]
— 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 (8.3.5). [Example:

```c
void h(int());
void h(int (*)(")); // redeclaration of h(int())
void h(int x()) { } // definition of h(int())
void h(int (*)(")) { } // ill-formed: redefinition of h(int())
```
— end example]

— Parameter declarations that differ only in the presence or absence of `const` and/or `volatile` are equivalent. That is, the `const` and `volatile` type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called. [Example:

```c
typedef const int cInt;
int f (int);
int f (const int); // redeclaration of f(int)
int f (int) { /* ... */ } // definition of f(int)
int f (cInt) { /* ... */ } // error: redefinition of f(int)
```
— 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.\(^\text{123}\) 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]}

### 13.2 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 (13.1). 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:

1. When a parameter type includes a function type, such as in the case of a parameter type that is a pointer to function, the `const` and `volatile` type-specifiers at the outermost level of the parameter type specifications for the inner function type are also ignored.
struct 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:]

void f(const char*);
void g() {
    extern void f(int);
    f("asdf"); // error: f(int) hides f(const char*)
    // so there is no f(const char*) in this scope
}

void caller () {
    extern void callee(int, int);
    {
        extern void callee(int); // hides callee(int, int)
        callee(88, 99); // error: only callee(int) in scope
    }
}

— end example —

Different versions of an overloaded member function can be given different access rules. [Example:]

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 —

13.3 Overload resolution [over.match]

Overload resolution is a mechanism for selecting the best function to call given a list of expressions that are to be the arguments of the call and a set of candidate functions that can be called based on the context of the call. The selection criteria for the best function are the number of arguments, how well the arguments match the parameter-type-list of the candidate function, how well (for non-static member functions) the
object matches the implicit object parameter, and certain other properties of the candidate function. [Note: The function selected by overload resolution is not guaranteed to be appropriate for the context. Other restrictions, such as the accessibility of the function, can make its use in the calling context ill-formed. —end note]

Overload resolution selects the function to call in seven distinct contexts within the language:

- (2.1) invocation of a function named in the function call syntax (13.3.1.1);
- (2.2) invocation of a function call operator, a pointer-to-function conversion function, a reference-to-pointerto-function conversion function, or a reference-to-function conversion function on a class object named in the function call syntax (13.3.1.2);
- (2.3) invocation of the operator referenced in an expression (13.3.1.2);
- (2.4) invocation of a constructor for default- or direct-initialization (8.5) of a class object (13.3.1.3);
- (2.5) invocation of a user-defined conversion for copy-initialization (8.5) of a class object (13.3.1.4);
- (2.6) invocation of a conversion function for initialization of an object of a nonclass type from an expression of class type (13.3.1.5); and
- (2.7) invocation of a conversion function for conversion to a glvalue or class prvalue to which a reference (8.5.3) will be directly bound (13.3.1.6).

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 (13.3.2).
- (2.9) Then the best viable function is selected based on the implicit conversion sequences (13.3.3.1) 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 11) in the context in which it is used, the program is ill-formed.

13.3.1 Candidate functions and argument lists [over.match.funcs]

The subclauses of 13.3.1 describe the set of candidate functions and the argument list submitted to overload resolution in each of the seven contexts in which overload resolution is used. The source transformations and constructions defined in these subclauses are only for the purpose of describing the overload resolution process. An implementation is not required to use such transformations and constructions.

2 The set of candidate functions can contain both member and non-member functions to be resolved against the same argument list. So that argument and parameter lists are comparable within this heterogeneous set, a member function is considered to have an extra parameter, called the implicit object parameter, which represents the object for which the member function has been called. For the purposes of overload resolution, both static and non-static member functions have an implicit object parameter, but constructors do not.

3 Similarly, when appropriate, the context can construct an argument list that contains an implied object argument 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.

4 For non-static member functions, the type of the implicit object parameter is
“lvalue reference to cv X” for functions declared without a ref-qualifier or with the & ref-qualifier

“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 (13.3.3). — end note]

During overload resolution, the implied object argument is indistinguishable from other arguments. The implicit object parameter, however, retains its identity since conversions on the corresponding argument shall obey these additional rules:

— no temporary object can be introduced to hold the argument for the implicit object parameter; and
— 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:

— 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 (13.3.3.2). — 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 (13.3.3, 13.3.3.1). [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 (14.8.3, 14.8.2). Those candidates are then handled as candidate functions in the usual way.124 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 constructor or assignment operator (12.8) that is defined as deleted is excluded from the set of candidate functions in all contexts.

124) 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 (8.3.5) are treated equivalently for the remainder of overload resolution.
13.3.1.1 Function call syntax

1 In a function call (5.2.2)

\[
\text{postfix-expression ( expression-list_{opt} \text{opt})}
\]

if the postfix-expression denotes a set of overloaded functions and/or function templates, overload resolution is applied as specified in 13.3.1.1.1. If the postfix-expression denotes an object of class type, overload resolution is applied as specified in 13.3.1.1.2.

2 If the 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. [ Note: The resolution of the address of an overload set in other contexts is described in 13.4. — end note ]

13.3.1.1.1 Call to named function

1 Of interest in 13.3.1.1.1 are only those function calls in which the postfix-expression ultimately contains a name that denotes one or more functions that might be called. Such a postfix-expression, perhaps nested arbitrarily deep in parentheses, has one of the following forms:

    postfix-expression:
    postfix-expression . id-expression
    postfix-expression -> id-expression
    primary-expression

These represent two syntactic subcategories of function calls: qualified function calls and unqualified function calls.

2 In qualified function calls, the name to be resolved is an id-expression and is preceded by an -> or . operator. Since the construct A->B is generally equivalent to (*A).B, the rest of Clause 13 assumes, without loss of generality, that all member function calls have been normalized to the form that uses an object and the . operator. Furthermore, Clause 13 assumes that the postfix-expression that is the left operand of the . operator has type “cv T” where T denotes a class\(^{125}\). Under this assumption, the id-expression in the call is looked up as a member function of T following the rules for looking up names in classes (10.2). The function declarations found by that lookup constitute the set of candidate functions. The argument list is the expression-list in the call augmented by the addition of the left operand of the . operator in the normalized member function call as the implied object argument (13.3.1).

3 In unqualified function calls, the name is not qualified by an -> or . operator and has the more general form of a primary-expression. The name is looked up in the context of the function call following the normal rules for name lookup in function calls (3.4). 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 T. In case (1), the argument list is the same as the expression-list in the call. In case (2), the argument list is the expression-list in the call augmented by the addition of an implied object argument as in a qualified function call. If the keyword this (9.3.2) is in scope and refers to class T, or a derived class of T, then the implied object argument is (*this). If the keyword this is not in scope or refers to another class, then a contrived object of type T becomes the implied object argument\(^{126}\). If the argument list is augmented by a contrived object and overload resolution selects one of the non-static member functions of T, the call is ill-formed.

\(^{125}\) Note that cv-qualifiers on the type of objects are significant in overload resolution for both glvalue and class prvalue objects.

\(^{126}\) 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.
13.3.1.1.2 Call to object of class type

If the primary-expression \( E \) in the function call syntax evaluates to a class object of type "\( cv \ T \)"\(^{1}\), then the set of candidate functions includes at least the function call operators of \( T \). The function call operators of \( T \) are obtained by ordinary lookup of the name \( \text{operator()} \) in the context of \((E).\text{operator}()\). \(^2\)

In addition, for each non-explicit conversion function declared in \( T \) of the form

\[
\text{operator } \text{conversion-type-id} () \text{ cv-qualifier ref-qualifier opt exception-specification opt attribute-specifier-seq opt;}
\]

where \( \text{cv-qualifier} \) is the same cv-qualification as, or a greater cv-qualification than, \( \text{cv} \), \(^1\) and where \( \text{conversion-type-id} \) denotes the type “pointer to function of \((P_1,...,P_n)\) returning \( R \)”, or the type “reference to pointer to function of \((P_1,...,P_n)\) returning \( R \)”, or the type “reference to function of \((P_1,...,P_n)\) returning \( R \)”, a surrogate call function with the unique name \( \text{call-function} \) and having the form

\[
R \text{ call-function ( conversion-type-id F, P1 a1, \ldots ,Pn an) ; return F (a1,\ldots ,an); }
\]

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. \(^127\)

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. \(^3\)

The argument list submitted to overload resolution consists of the argument expressions present in the function call syntax preceded by the implied object argument \((E)\). \(^4\)[ 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] \(^{127}\)[ Example:

```c
int f1(int); int f2(float); typedef int (*fp1)(int); typedef int (*fp2)(float); struct A {
    operator fp1() { return f1; }
    operator fp2() { return f2; }
} a;
int i = a(1); // calls f1 via pointer returned from conversion function
```

— end example]

13.3.1.2 Operators in expressions

If no operand of an operator in an expression has a type that is a class or an enumeration, the operator is assumed to be a built-in operator and interpreted according to Clause 5. \(^1\)[ Note: Because . , .*, and :: cannot be overloaded, these operators are always built-in operators interpreted according to Clause 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 (5.16). — end note] \(^{127}\)[ Example:

```c
int f1(int); int f2(float); typedef int (*fp1)(int); typedef int (*fp2)(float); struct A {
    operator fp1() { return f1; }
    operator fp2() { return f2; }
} a;
int i = a(1); // calls f1 via pointer returned from conversion function
```

— end example]

\(^{127}\) 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.
struct String {
    String (const String&);
    String (const char*);
    operator const char* ();
};
String operator + (const String&, const String&);

void f(void) {
    const char* p= "one" + "two";  // ill-formed because neither
    // operand has class or enumeration type
    int I = 1 + 1;  // Always evaluates to 2 even if
    // class or enumeration types exist that
    // would perform the operation.
}

— end example —

2 If either operand has a type that is a class or an enumeration, a user-defined operator function might be
declared that implements this operator or a user-defined conversion can be necessary to convert the operand
to a type that is appropriate for a built-in operator. In this case, overload resolution is used to determine
which operator function or built-in operator is to be invoked to implement the operator. Therefore, the
operator notation is first transformed to the equivalent function-call notation as summarized in Table 10
(where @ denotes one of the operators covered in the specified subclause).

Table 10 — 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>13.5.1</td>
<td>@a</td>
<td>(a).operator@ ()</td>
<td>operator@ (a)</td>
</tr>
<tr>
<td>13.5.2</td>
<td>a@b</td>
<td>(a).operator@ (b)</td>
<td>operator@ (a, b)</td>
</tr>
<tr>
<td>13.5.3</td>
<td>a=b</td>
<td>(a).operator= (b)</td>
<td></td>
</tr>
<tr>
<td>13.5.5</td>
<td>a[b]</td>
<td>(a).operator[(b)</td>
<td></td>
</tr>
<tr>
<td>13.5.6</td>
<td>a-&gt;</td>
<td>(a).operator-&gt; ()</td>
<td></td>
</tr>
<tr>
<td>13.5.7</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@ (13.3.1.1.1); 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 (3.4.2)
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 (possibly
cv-qualified) T1”, when T1 is an enumeration type, or (if there is a right operand) a second parameter
of type T2 or “reference to (possibly cv-qualified) 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 13.6 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 13.3.3.1, 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 is the union of the member candidates, the non-member candidates, and the built-in candidates. The argument list contains all of the operands of the operator. The best function from the set of candidate functions is selected according to 13.3.2 and 13.3.3.128 [Example:

```c
struct A {
    operator int();
};
A operator+(const A&, const A&);
void m() {
    A a, b;
    a + b;  // operator+(a,b) chosen over int(a) + int(b)
}
— end example]
```

7 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 (13.3.3.1.2) is not applied. Then the operator is treated as the corresponding built-in operator and interpreted according to Clause 5. [Example:

```c
struct X {
    operator double();
};
struct Y {
    operator int*();
};
int *a = Y() + 100.0;  // error: pointer arithmetic requires integral operand
int *b = Y() + X();   // error: pointer arithmetic requires integral operand
— end example]
```

8 The second operand of operator `->` is ignored in selecting an `operator->` function, and is not an argument when the `operator->` function is called. When `operator->` returns, the operator `->` is applied to the value returned, with the original second operand.129

128) If the set of candidate functions is empty, overload resolution is unsuccessful.
129) If the value returned by the `operator->` function has class type, this may result in selecting and calling another `operator->` function. The process repeats until an `operator->` function returns a value of non-class type.
If the operator is the operator \(\,\), the unary \&, or the operator \(\rightarrow\), and there are no viable functions, then the operator is assumed to be the built-in operator and interpreted according to Clause 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 }

13.3.1.3 Initialization by constructor

When objects of class type are direct-initialized (8.5), copy-initialized from an expression of the same or a derived class type (8.5), or default-initialized (8.5), overload resolution selects the constructor. For direct-initialization or default-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 (12.3.1) of that class. The argument list is the expression-list or assignment-expression of the initializer.

13.3.1.4 Copy-initialization of class by user-defined conversion

Under the conditions specified in 8.5, 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. The converting constructors (12.3.1) of T are candidate functions.
2. When the type of the initializer expression is a class type “cv S”, the non-explicit conversion functions of S and its base classes are considered. When initializing a temporary 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 lvales 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.

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]
13.3.1.5 Initialization by conversion function

Under the conditions specified in 8.5, as part of an initialization of an object of nonclass 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 (13.3.3.1.1) 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 (4.4) 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]

13.3.1.6 Initialization by conversion function for direct reference binding

Under the conditions specified in 8.5.3, a reference can be bound directly to a glvalue 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 “cv1 T” is the underlying 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 “lvalue reference to cv2 T2” (when initializing an lvalue reference or an rvalue reference to function) or “cv2 T2” or “rvalue reference to cv2 T2” (when initializing an rvalue reference or an lvalue reference to function), where “cv1 T” is reference-compatible (8.5.3) 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 (4.4), are also candidate functions.

2 The argument list has one argument, which is the initializer expression. [Note: This argument will be compared against the implicit object parameter of the conversion functions. — end note]

13.3.1.7 Initialization by list-initialization

When objects of non-aggregate class type T are list-initialized such that 8.5.4 specifies that overload resolution is performed according to the rules in this section, overload resolution selects the constructor in two phases:

(1.1) Initially, the candidate functions are the initializer-list constructors (8.5.4) of the class T and the argument list consists of the initializer list as a single argument.

(1.2) If no viable initializer-list constructor is found, overload resolution is performed again, where the candidate functions are all the constructors of the class T and the argument list consists of the elements of the initializer list.

If the initializer list has no elements and T has a default constructor, the first phase is omitted. In copy-list-initialization, if an explicit constructor is chosen, the initialization is ill-formed. [Note: This differs from other situations (13.3.1.3, 13.3.1.4), 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]
13.3.2 Viable functions

1 From the set of candidate functions constructed for a given context (13.3.1), a set of viable functions is chosen, from which the best function will be selected by comparing argument conversion sequences for the best fit (13.3.3). The selection of viable functions considers relationships between arguments and function parameters other than the ranking of conversion sequences.

2 First, to be a viable function, a candidate function shall have enough parameters to agree in number with the arguments in the list.

(2.1) — If there are \( m \) arguments in the list, all candidate functions having exactly \( m \) parameters are viable.

(2.2) — A candidate function having fewer than \( m \) parameters is viable only if it has an ellipsis in its parameter list (8.3.5). For the purposes of overload resolution, any argument for which there is no corresponding parameter is considered to “match the ellipsis” (13.3.3.1.3).

(2.3) — A candidate function having more than \( m \) parameters is viable only if the \((m+1)\)-st parameter has a default argument (8.3.6). For the purposes of overload resolution, the parameter list is truncated on the right, so that there are exactly \( m \) parameters.

3 Second, for \( F \) to be a viable function, there shall exist for each argument an implicit conversion sequence (13.3.3.1) 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-\( \text{const} \) 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 13.3.3.1.4).

13.3.3 Best viable function

1 Define \( \text{ICS}(F) \) as follows:

(1.1) — if \( F \) is a static member function, \( \text{ICS}(F) \) is defined such that \( \text{ICS}(F) \) is neither better nor worse than \( \text{ICS}(G) \) for any function \( G \), and, symmetrically, \( \text{ICS}(G) \) is neither better nor worse than \( \text{ICS}(F) \); otherwise,

(1.2) — let \( \text{ICS}_i(F) \) denote the implicit conversion sequence that converts the \( i \)-th argument in the list to the type of the \( i \)-th parameter of viable function \( F \). 13.3.3.1 defines the implicit conversion sequences and 13.3.3.2 defines what it means for one implicit conversion sequence to be a better conversion sequence or worse conversion sequence than another.

Given these definitions, a viable function \( F_1 \) is defined to be a better function than another viable function \( F_2 \) if for all arguments \( i \), \( \text{ICS}_i(F_1) \) is not a worse conversion sequence than \( \text{ICS}_i(F_2) \), and then

(1.3) — for some argument \( j \), \( \text{ICS}_j(F_1) \) is a better conversion sequence than \( \text{ICS}_j(F_2) \), or, if not that,

(1.4) — the context is an initialization by user-defined conversion (see 8.5, 13.3.1.5, and 13.3.1.6) 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. [Example:

```c
struct A {
    A();
    operator int();
    operator double();
} a;
```

130) According to 8.3.6, parameters following the \((m+1)\)-st parameter must also have default arguments.

131) If a function is a static member function, this definition means that the first argument, the implied object argument, has no effect in the determination of whether the function is better or worse than any other function.

§ 13.3.3
int i = a; // a.operator int() followed by no conversion
         // is better than a.operator double() followed by
         // a conversion to int
float x = a; // ambiguous: both possibilities require conversions,
             // and neither is better than the other

— end example] or, if not that,

(1.5) the context is an initialization by conversion function for direct reference binding (13.3.1.6) of a reference
         to function type, the return type of F1 is the same kind of reference (i.e. lvalue or rvalue) as the
         reference being initialized, and the return type of F2 is not [Example:

template <class T> struct A {
    operator T&(); // #1
    operator T&&(); // #2
};
typedef int Fn();
A<Fn> a;
Fn& lf = a; // calls #1
Fn&& rf = a; // calls #2

— end example] or, if not that,

(1.6) F1 is not a function template specialization and F2 is a function template specialization, or, if not that,

(1.7) F1 and F2 are function template specializations, and the function template for F1 is more specialized
         than the template for F2 according to the partial ordering rules described in 14.5.6.2.

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-formed132.

[Example:

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
}

132) 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.

§ 13.3.3
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]

### 13.3.3.1 Implicit conversion sequences

An implicit conversion sequence is a sequence of conversions used to convert an argument in a function call to the type of the corresponding parameter of the function being called. The sequence of conversions is an implicit conversion as defined in Clause 4, which means it is governed by the rules for initialization of an object or reference by a single expression (8.5, 8.5.3).

Implicit conversion sequences are concerned only with the type, cv-qualification, and value category of the argument and how these are converted to match the corresponding properties of the parameter. 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 (8.4.3), 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.

A well-formed implicit conversion sequence is one of the following forms:

1. A standard conversion sequence (13.3.3.1.1),
2. A user-defined conversion sequence (13.3.3.1.2), or
3. An ellipsis conversion sequence (13.3.3.1.3).

However, if the target is

1. the first parameter of a constructor or
2. the implicit object parameter of a user-defined conversion function

and the constructor or user-defined conversion function is a candidate by

1. 13.3.1.3, when the argument is the temporary in the second step of a class copy-initialization, or
2. 13.3.1.4, 13.3.1.5, or 13.3.1.6 (in all cases),
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
```

—end example]

5 For the case where the parameter type is a reference, see 13.3.3.1.4.

6 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 13; the actual initialization is defined in terms of constructors and is not a conversion. —end note] Any difference in top-level cv-qualification is subsumed by the initialization itself and does not constitute a conversion. [Example: a parameter of type A can be initialized from an argument of type const A. The implicit conversion sequence for that case is the identity sequence; it contains no “conversion” from const A to A. —end example] When the parameter has a class type and the argument expression has the same type, the implicit conversion sequence is an identity conversion. When the parameter has a class type and the argument expression has a derived class type, the implicit conversion sequence is a derived-to-base Conversion from the derived class to the base class. [Note: There is no such standard conversion; this derived-to-base Conversion exists only in the description of implicit conversion sequences. —end note] A derived-to-base Conversion has Conversion rank (13.3.3.1.1).

7 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 that create no temporary object for the result are allowed.

8 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 (13.3.3.1.1).

9 If no sequence of conversions can be found to convert an argument to a parameter type, an implicit conversion sequence cannot be formed.

10 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 in the ambiguous conversion sequence. For the purpose of ranking implicit conversion sequences as described in 13.3.3.2, the ambiguous conversion sequence is treated as a user-defined sequence that is indistinguishable from any other user-defined conversion sequence. If a function that uses the ambiguous conversion sequence is ranked with user-defined conversion sequences because multiple conversion sequences for an argument can exist only if they involve different user-defined conversions. The ambiguous conversion sequence is indistinguishable from any other user-defined conversion sequence because it represents at least two user-defined conversion sequences, each with a different user-defined conversion, and any other user-defined conversion sequence must be indistinguishable from at least one of them.

This rule prevents a function from becoming non-viable because of an ambiguous conversion sequence for one of its parameters. Consider this example:

```cpp
class B;
class A ( B&); // A is a reference
class B { operator A () ; }
class C ( C ( B&); );
```

The ambiguous conversion sequence is ranked with user-defined conversion sequences because multiple conversion sequences for an argument can exist only if they involve different user-defined conversions. The ambiguous conversion sequence is indistinguishable from any other user-defined conversion sequence because it represents at least two user-defined conversion sequences, each with a different user-defined conversion, and any other user-defined conversion sequence must be indistinguishable from at least one of them.

This rule prevents a function from becoming non-viable because of an ambiguous conversion sequence for one of its parameters. Consider this example,
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.

### 13.3.3.1.1 Standard conversion sequences

Table 11 summarizes the conversions defined in Clause 4 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 4, 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. At most one conversion from each category is allowed in a single standard conversion sequence. If there are two or more conversions in the sequence, the conversions are applied in the canonical order: Lvalue Transformation, Promotion or Conversion, Qualification Adjustment. — end note]

Each conversion in Table 11 also has an associated rank (Exact Match, Promotion, or Conversion). These are used to rank standard conversion sequences (13.3.3.2). 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 (13.3.3.1.4). 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>Exact Match</td>
<td>4.1</td>
</tr>
<tr>
<td>Lvalue-to-rvalue conversion</td>
<td>Lvalue Transformation</td>
<td>Exact Match</td>
<td>4.2</td>
</tr>
<tr>
<td>Array-to-pointer conversion</td>
<td></td>
<td></td>
<td>4.3</td>
</tr>
<tr>
<td>Function-to-pointer conversion</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Qualification conversions</td>
<td>Qualification Adjustment</td>
<td></td>
<td>4.4</td>
</tr>
<tr>
<td>Integral promotions</td>
<td>Promotion</td>
<td>Promotion</td>
<td>4.5</td>
</tr>
<tr>
<td>Floating point promotion</td>
<td></td>
<td></td>
<td>4.6</td>
</tr>
<tr>
<td>Integral conversions</td>
<td></td>
<td></td>
<td>4.7</td>
</tr>
<tr>
<td>Floating point conversions</td>
<td></td>
<td></td>
<td>4.8</td>
</tr>
<tr>
<td>Floating-integral conversions</td>
<td></td>
<td></td>
<td>4.9</td>
</tr>
<tr>
<td>Pointer conversions</td>
<td>Conversion</td>
<td>Conversion</td>
<td>4.10</td>
</tr>
<tr>
<td>Pointer to member conversions</td>
<td></td>
<td></td>
<td>4.11</td>
</tr>
<tr>
<td>Boolean conversions</td>
<td></td>
<td></td>
<td>4.12</td>
</tr>
</tbody>
</table>

```c
void f(A) { }
void f(C) { }
B b;
f(b);  // ambiguous because b \rightarrow C via constructor and
      // b \rightarrow A via constructor or conversion function.
```

If it were not for this rule, \( f(A) \) would be eliminated as a viable function for the call \( f(b) \) causing overload resolution to select \( f(C) \) as the function to call even though it is not clearly the best choice. On the other hand, if an \( f(B) \) were to be declared then \( f(b) \) would resolve to that \( f(B) \) because the exact match with \( f(B) \) is better than any of the sequences required to match \( f(A) \).

§ 13.3.3.1.1
13.3.3.1.2 User-defined conversion sequences

A user-defined conversion sequence consists of an initial standard conversion sequence followed by a user-defined conversion (12.3) followed by a second standard conversion sequence. If the user-defined conversion is specified by a constructor (12.3.1), 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 (12.3.2), the initial standard conversion sequence converts the source type to the implicit object parameter of the conversion function.

The second standard conversion sequence converts the result of the user-defined conversion to the target type for the sequence. 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 13.3.3 and 13.3.3.1).

If the user-defined conversion is specified by a specialization of a conversion function template, the second standard conversion sequence shall have exact match rank.

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.

13.3.3.1.3 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 5.2.2).

13.3.3.1.4 Reference binding

When a parameter of reference type binds directly (8.5.3) 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 (13.3.3.1).

Example:

```cpp
struct A {}
struct B : public A {} b;
int f(A&);
int f(B&);
int i = f(b); // calls f(B&), an exact match, rather than
              // f(A&), a conversion
```

— end example] If the parameter binds directly to the result of applying a conversion function to the argument expression, the implicit conversion sequence is a user-defined conversion sequence (13.3.3.1.2), 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 underlying type of the reference according to 13.3.3.1. Conceptually, this conversion sequence corresponds to copy-initializing a temporary of the underlying type with the argument expression. Any difference in top-level cv-qualification is subsumed by the initialization itself and does not constitute a conversion.

Except for an implicit object parameter, for which see 13.3.1, 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 is a temporary or would require one to be created to initialize the lvalue reference (see 8.5.3). — 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-\texttt{\texttt{\texttt{const}}} lvalue reference to a bit-field (8.5.3). — end example]

13.3.3.1.5 List-initialization sequence

1 When an argument is an initializer list (8.5.4), it is not an expression and special rules apply for converting it to a parameter type.

2 If the parameter type is a class X and the initializer list has a single element of type \texttt{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.

3 Otherwise, if the parameter type is a character array and the initializer list has a single element that is an appropriately-typed string literal (8.5.2), the implicit conversion sequence is the identity conversion.

4 Otherwise, if the parameter type is \texttt{\texttt{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
void f(std::initializer\_list<int>);
f ( \{ \} ); \hspace{1em} // OK: f(initializer\_list<int>) identity conversion
f ( \{1,2,3\} ); \hspace{1em} // OK: f(initializer\_list<int>) identity conversion
f ( \{'a','b'\} ); \hspace{1em} // OK: f(initializer\_list<int>) identity conversion
f ( \{1.0\} ); \hspace{1em} // error: narrowing

struct A {
    A(std::initializer\_list<double>); \hspace{1em} // #1
    A(std::initializer\_list<complex<double>>); \hspace{1em} // #2
    A(std::initializer\_list<std::string>); \hspace{1em} // #3
};
A a { 1.0,2.0 }; \hspace{1em} // OK, uses #1

void g(A);
g(\{ "foo", "bar" \}); \hspace{1em} // OK, uses #3

typedef int IA[3];
void h(const IA&);
h(\{ 1, 2, 3 \}); \hspace{1em} // OK: identity conversion
```

— end example]

5 Otherwise, if the parameter type is “array of N X”, if the initializer list has exactly N elements or if it has fewer than N elements and X is default-constructible, and if 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.

6 Otherwise, if the parameter is a non-aggregate class X and overload resolution per 13.3.1.7 chooses a single best constructor of X to perform the initialization of an object of type X from the argument initializer list, 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

134) Since there are no parameters of array type, this will only occur as the underlying type of a reference parameter.
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 13.3.3.1.

[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

void g(B);
g( {'a', 'b'} );  // OK: g(B(int,double)) user-defined conversion
g( {1.0, 1.0} );  // error: narrowing

void f(B);
f( {'a', 'b'} );  // error: ambiguous f(A) or f(B)

struct C {
    C(std::string);
};
void h(C);
h( "foo" );  // OK: h(C(std::string("foo")))

struct D {
    D(A, C);
};
void i(D);
i( {1,2}, {"bar"} );  // OK: i(D(A(std::initializer_list<int>{1,2}),C(std::string("bar"))))
```

—end example]

7 Otherwise, if the parameter has an aggregate type which can be initialized from the initializer list according to the rules for aggregate initialization (8.5.1), the implicit conversion sequence is a user-defined conversion sequence with the second standard conversion sequence an identity conversion. [Example:

```cpp
struct A {
    int m1;
    double m2;
};

void f(A);
f( {'a', 'b'} );  // OK: f(A(int,double)) user-defined conversion
f( {1.0} );  // error: narrowing
```

—end example]

8 Otherwise, if the parameter is a reference, see 13.3.3.1.4. [Note: The rules in this section will apply for initializing the underlying temporary for the reference. —end note] [Example:

```cpp
struct A {
    int m1;
    double m2;
};
```

void f(const A&);
f( {'a', 'b'} );        // OK: f(A(int,double)) user-defined conversion
f( {1.0} );            // error: narrowing

void g(const double &);
g({1});                 // same conversion as int to double

— end example]

9 Otherwise, if the parameter type is not a class:

(9.1) if the initializer list has one element that is not itself an initializer list, the implicit conversion sequence is the one required to convert the element to the parameter type; [Example:

      void f(int);
f( {'a'} );        // OK: same conversion as char to int
f( {1} );         // error: narrowing

      — end example]

(9.2) if the initializer list has no elements, the implicit conversion sequence is the identity conversion. [Example:

      void f(int);
f( {} );         // OK: identity conversion

      — end example]

10 In all cases other than those enumerated above, no conversion is possible.

13.3.3.2 Ranking implicit conversion sequences [over.ics.rank]

1 13.3.3.2 defines a partial ordering of implicit conversion sequences based on the relationships better conversion sequence and better conversion. If an implicit conversion sequence S1 is defined by these rules to be a better conversion sequence than S2, then it is also the case that S2 is a worse conversion sequence than S1. If conversion sequence S1 is neither better than nor worse than conversion sequence S2, S1 and S2 are said to be indistinguishable conversion sequences.

2 When comparing the basic forms of implicit conversion sequences (as defined in 13.3.3.1)

(2.1) a standard conversion sequence (13.3.3.1.1) is a better conversion sequence than a user-defined conversion sequence or an ellipsis conversion sequence, and

(2.2) a user-defined conversion sequence (13.3.3.1.2) is a better conversion sequence than an ellipsis conversion sequence (13.3.3.1.3).

3 Two implicit conversion sequences of the same form are indistinguishable conversion sequences unless one of the following rules applies:

(3.1) List-initialization sequence L1 is a better conversion sequence than list-initialization sequence L2 if

(3.1.1) L1 converts to std::initializer_list<X> for some X and L2 does not, or, if not that,

(3.1.2) L1 converts to type “array of N1 T”, L2 converts to type “array of N2 T”, and N1 is smaller than N2, even if one of the other rules in this paragraph would otherwise apply. [Example:

      void f1(int);       // #1
void f1(std::initializer_list<long>);       // #2
void g1() { f1(42); }            // chooses #2

      void f2(std::pair<const char*, const char*>); // #3
void f2(std::initializer_list<std::string>); // #4
void g2() { f2(“foo”, “bar”); }            // chooses #4

§ 13.3.3.2 320
Standard conversion sequence $S_1$ is a better conversion sequence than standard conversion sequence $S_2$ if

— $S_1$ is a proper subsequence of $S_2$ (comparing the conversion sequences in the canonical form defined by 13.3.3.1.1, 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 $S_1$ is better than the rank of $S_2$, or $S_1$ and $S_2$ have the same rank and are distinguishable by the rules in the paragraph below, or, if not that,

— $S_1$ and $S_2$ are reference bindings (8.5.3) and neither refers to an implicit object parameter of a non-static member function declared without a ref-qualifier, and $S_1$ binds an rvalue reference to an rvalue and $S_2$ binds an lvalue reference.

[Example:

```c
int i;
int f1();
int&& f2();
int g(const int&);
int g(const int&&);
int j = g(i); // calls g(const int&)
int k = g(f1()); // calls g(const int&&)
int l = g(f2()); // calls g(const int&&)
```

```c
struct A {
    A& operator<<(int);
    void p() &;
    void p() &&;
};
A& operator<<(A&&, char);
A() <<= 1; // calls A::operator<<(int)
A() <<= 'c'; // calls A::operator<<(int)
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,

— $S_1$ and $S_2$ are reference bindings (8.5.3) and $S_1$ binds an lvalue reference to a function lvalue and $S_2$ binds an rvalue reference to a function lvalue. [Example:

```c
int f(void(&)()); // #1
int f(void(&&)()); // #2
void g();
int i1 = f(g); // calls #1
```

— end example] or, if not that,

— $S_1$ and $S_2$ differ only in their qualification conversion and yield similar types $T_1$ and $T_2$ (4.4), respectively, and the cv-qualification signature of type $T_1$ is a proper subset of the cv-qualification signature of type $T_2$. [Example:

```c
int f(const volatile int *);
int f(const int *);
int i;
int j = f(&i); // calls f(const int*)
```
— end example] or, if not that,

(3.2.6) — S1 and S2 are reference bindings (8.5.3), 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:

```c
int f(const int &);
int f(int &);
int g(const int &);
int g(int);

int i;
int j = f(i);  // calls f(int &)
int k = g(i);  // ambiguous

struct X {
    void f() const;
    void f();
};
void g(const X& a, X b) {
    a.f();     // calls X::f() const
    b.f();     // calls X::f()
}
— end example]
```

(3.3) — User-defined conversion sequence U1 is a better conversion sequence than another user-defined conversion sequence U2 if they contain the same user-defined conversion function or constructor or they initialize the same class in an aggregate initialization and in either case the second standard conversion sequence of U1 is better than the second standard conversion sequence of U2. [Example:

```c
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]
```

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 13.3.3); in all other contexts, the source types will be the same and the target types will be different. — end note]

13.4 Address of overloaded function [over.over]

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. The function selected is the one whose type is identical to the function type of the target type required in the context. [ 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 (8.5, 8.5.3, 8.5.4),

(1.2) — the left side of an assignment (5.18),

(1.3) — a parameter of a function (5.2.2),

(1.4) — a parameter of a user-defined operator (13.5),

(1.5) — the return value of a function, operator function, or conversion (6.6.3),

(1.6) — an explicit type conversion (5.2.3, 5.2.9, 5.4), or

(1.7) — a non-type template-parameter (14.3.2).

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 (5.1). — end note]

If the name is a function template, template argument deduction is done (14.8.2.2), 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 14.8.1, 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 type “pointer-to-function” or “reference-to-function.” Nonstatic member functions match targets of type “pointer-to-member-function”. If a nonstatic 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 5.3.1.

4 If more than one function is selected, any function template specializations in the set are eliminated if the set also contains a function that is not a function template specialization, and any given function template specialization \( F_1 \) is eliminated if the set contains a second function template specialization whose function template is more specialized than the function template of \( F_1 \) according to the partial ordering rules of 14.5.6.2. After such eliminations, if any, there shall remain exactly one selected function.

5 [ Example:

```c
int f(double);
int f(int);
int (*pf)(double) = &f; // selects f(double)
int (*pf1)(int) = &f; // selects f(int)
int (*pf2)(...) = &f; // error: type mismatch
int &(rf)(int) = f; // selects f(int)
int &(rf2)(double) = f; // selects f(double)
void g() {
  (int (*)(int))*f; // cast expression as selector
}
```

The initialization of \( pf2 \) is ill-formed because no \( f() \) with type \( int(...) \) has been declared, and not because of any ambiguity. For another example,

```c
struct X {
  int f(int);
  static int f(long);
};

int (X::*p1)(int) = &X::f; // OK
int (*p2)(int) = &X::f; // error: mismatch
int (*p3)(long) = &X::f; // OK
int (X::*p4)(long) = &X::f; // error: mismatch
int (X::*p5)(int) = &X::f; // error: wrong syntax for
                         // pointer to member
int (*p6)(long) = &X::f; // OK
```

—end example]

6 [ Note: If \( f() \) and \( g() \) are both overloaded functions, the cross product of possibilities must be considered to resolve \( f(&g) \), or the equivalent expression \( f(g) \).  — end note]

7 [ Note: There are no standard conversions (Clause 4) of one pointer-to-function type into another. In particular, even if \( B \) is a public base of \( D \), we have

```c
D* f();
B* (*p1)() = &f; // error
```

void g(D*);
void (*p2)(B*) = &g; // error

—end note]
13.5 Overloaded operators

A function declaration having one of the following operator-function-ids as its name declares an operator function. A function template declaration having one of the following operator-function-ids as its name declares an operator function template. A specialization of an operator function template is also an operator function. An operator function is said to implement the operator named in its operator-function-id.

operator-function-id:

operator operator

operator: one of

c new delete new[] delete[] + - * / % - k l -
! k = < > += -= *= /= %= k=
>= >> += -= *= /= %= l =
<= || | & | & | = || | += -= *= /= %= l
()

[Note: The last two operators are function call (5.2.2) and subscripting (5.2.1). The operators new[], delete[], (), and [] are formed from more than one token. —end note]

Both the unary and binary forms of

+ - * k

can be overloaded.

The following operators cannot be overloaded:

. .* :: ?:

nor can the preprocessing symbols # and ## (Clause 16).

Operator functions are usually not called directly; instead they are invoked to evaluate the operators they implement (13.5.1 – 13.5.7). They can be explicitly called, however, using the operator-function-id as the name of the function in the function call syntax (5.2.2). [Example:

complex z = a.operator+(b); //complex z = a+b;
void* p = operator new(sizeof(int)*n);

—end example]

The allocation and deallocation functions, operator new, operator new[], operator delete and operator delete[], are described completely in 3.7.4. The attributes and restrictions found in the rest of this subclause do not apply to them unless explicitly stated in 3.7.4.

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.

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.

An operator function cannot have default arguments (8.3.6), except where explicitly stated below. Operator functions cannot have more or fewer parameters than the number required for the corresponding operator, as described in the rest of this subclause.

Operators not mentioned explicitly in subclauses 13.5.3 through 13.5.7 act as ordinary unary and binary operators obeying the rules of 13.5.1 or 13.5.2.
13.5.1 Unary operators

A prefix unary operator shall be implemented by a non-static member function (9.3) with no parameters or a non-member function with one parameter. Thus, for any prefix unary operator @, @x can be interpreted as either x.operator@() or operator@(x). If both forms of the operator function have been declared, the rules in 13.3.1.2 determine which, if any, interpretation is used. See 13.5.7 for an explanation of the postfix unary operators ++ and --.

The unary and binary forms of the same operator are considered to have the same name. [Note: Consequently, a unary operator can hide a binary operator from an enclosing scope, and vice versa. —end note]

13.5.2 Binary operators

A binary operator shall be implemented either by a non-static member function (9.3) with one parameter or by a non-member function with two parameters. Thus, for any binary operator @, x@y can be interpreted as either x.operator@(y) or operator@(x, y). If both forms of the operator function have been declared, the rules in 13.3.1.2 determine which, if any, interpretation is used.

13.5.3 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 (12.8), a base class assignment operator is always hidden by the copy assignment operator of the derived class.

Any assignment operator, even the copy and move assignment operators, can be virtual. [Note: For a derived class D with a base class B for which a virtual copy/move assignment has been declared, the copy/move assignment operator in D does not override B’s virtual copy/move assignment operator. [Example:

```c
struct B {
  virtual int operator= (int);  // calls D::operator=(int)
  virtual B& operator= (const B&);  // ditto
};
struct D : B {
  virtual int operator= (int);  // calls D::operator=(const B&)
  virtual D& operator= (const B&);  // ditto
};
D dobj1;
D dobj2;
B* bptr = &dobj1;
void f() {
  bptr->operator=(99);  // calls D::operator=(int)
  *bptr = 99;  // ditto
  bptr->operator=(dobj2);  // calls D::operator=(const B&)
  *bptr = dobj2;  // ditto
  dobj1 = dobj2;  // calls implicitly-declared
                  // D::operator=(const D&)
}
```
—end example] —end note]

13.5.4 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
postfix-expression ( expression-list,\opt )

where the postfix-expression evaluates to a class object and the possibly empty expression-list matches the parameter list of an operator() member function of the class. Thus, a call \( x(\text{arg}_1, \ldots) \) is interpreted as \( x.\text{operator}()(\text{arg}_1, \ldots) \) for a class object \( x \) of type \( T \) if \( T::\text{operator}()(T_1, T_2, T_3) \) exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3.3).

### 13.5.5 Subscripting

operator[] shall be a non-static member function with exactly one parameter. It implements the subscripting syntax

```
postfix-expression [ expression ]
```

or

```
postfix-expression [ braced-init-list ]
```

Thus, a subscripting expression \( x[y] \) is interpreted as \( x.\text{operator}[](y) \) for a class object \( x \) of type \( T \) if \( T::\text{operator}[](T_1) \) exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3.3). [Example:

```c
struct X {
    Z operator[](std::initializer_list<int>);
};
X x;
x[{1,2,3}] = 7; // OK: meaning x.\text{operator}[](\{1,2,3\})
int a[10];
a[{1,2,3}] = 7; // error: built-in subscript operator
```

— end example]

### 13.5.6 Class member access

operator-> shall be a non-static member function taking no parameters. It implements the class member access syntax that uses ->.

```
postfix-expression -> template\opt id-expression
postfix-expression -> pseudo-destructor-name
```

An expression \( x->m \) is interpreted as \( (x.\text{operator}->())->m \) for a class object \( x \) of type \( T \) if \( T::\text{operator}->() \) exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3).

### 13.5.7 Increment and decrement

The user-defined function called operator++ implements the prefix and postfix ++ operator. If this function is a non-static member function with no parameters, or a non-member function with one parameter, it defines the prefix increment operator ++ for objects of that type. If the function is a non-static member function with one parameter (which shall be of type int) or a non-member function with two parameters (the second of which shall be of type int), it defines the postfix increment operator ++ for objects of that type. When the postfix increment is called as a result of using the ++ operator, the int argument will have value zero.\textsuperscript{135} [Example:

```c
struct X {
    X& operator++(); // prefix ++
    X operator++(int); // postfix a++
};
```

\textsuperscript{135} Calling operator++ explicitly, as in expressions like \( a.\text{operator}++(2) \), has no special properties: The argument to operator++ is 2.
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.

13.5.8 User-defined literals [over.literal]

The string-literal or user-defined-string-literal in a literal-operator-id shall have no encoding-prefix and shall contain no characters other than the implicit terminating ‘\0’. The ud-suffix of the user-defined-string-literal or the identifier in a literal-operator-id is called a literal suffix identifier. Some literal suffix identifiers are reserved for future standardization; see 17.6.4.3.4. A declaration whose literal-operator-id uses such a literal suffix identifier is ill-formed; no diagnostic required.

A declaration whose declarator-id is a literal-operator-id shall be a declaration of a namespace-scope function or function template (it could be a friend function (11.3)), an explicit instantiation or specialization of a function template, or a using-declaration (7.3.3). 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 (8.3.6), the program is ill-formed.

A raw literal operator is a literal operator with a single parameter whose type is const char*.

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 (14.5.3) with element type char.
6 Literal operators and literal operator templates shall not have C language linkage.

7 [Note: Literal operators and literal operator templates are usually invoked implicitly through user-defined literals (2.13.8). 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]

8 [Example:

```c
void operator " _km(long double); // OK
string operator " _i8m(const char*, std::size_t); // OK
template <char...> double operator " _u03C0(); // OK: UCN for lowercase pi
float operator " _e(const char*); // OK
float operator "E(const char*); // error: reserved literal suffix (17.6.4.3.4, 2.13.8)
double operator" _Bq(long double); // OK: does not use the reserved identifier _Bq (2.10)
double operator" _Bq(long double); // OK: uses the reserved identifier _Bq (2.10)
float operator " _Bq(const char*); // error: non-empty string-literal
string operator " _5X(const char*, std::size_t); // error: invalid literal suffix identifier
double operator " _miles(double); // error: invalid parameter-declaration-clause
template <char...> int operator " _j(const char*); // error: invalid parameter-declaration-clause
extern "C" void operator " _m(long double); // error: C language linkage
```

—end example]

13.6 Built-in operators [over.built]

1 The candidate operator functions that represent the built-in operators defined in Clause 5 are specified in this subclause. These candidate functions participate in the operator overload resolution process as described in 13.3.1.2 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 13.3.1.2, after a built-in operator is selected by overload resolution the expression is subject to the requirements for the built-in operator given in Clause 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 (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 For every pair \((T, VQ)\), where \(T\) is an arithmetic type, and \(VQ\) is either volatile or empty, there exist candidate operator functions of the form

   \[ VQ T& \text{ operator}++(VQ T&); \]
   \[ T \text{ operator}++(VQ T&, \text{int}); \]

4 For every pair \((T, VQ)\), where \(T\) is an arithmetic type other than bool, and \(VQ\) is either volatile or empty, there exist candidate operator functions of the form

§ 13.6
VQ T& operator--(VQ T&);
T operator--(VQ T&, int);

5 For every pair (T, VQ), where T is a cv-qualified or cv-unqualified object type, and VQ is either volatile or empty, there exist candidate operator functions of the form

T*VQ& operator++(T*VQ&);
T*VQ& operator--(T*VQ&);
T* operator++(T*VQ&, int);
T* operator--(T*VQ&, int);

6 For every cv-qualified or cv-unqualified object type T, there exist candidate operator functions of the form

T& operator*(T*);

7 For every function type T that does not have cv-qualifiers or a ref-qualifier, there exist candidate operator functions of the form

T& operator*(T*);

8 For every type T there exist candidate operator functions of the form

T* operator+(T*);

9 For every promoted arithmetic type T, there exist candidate operator functions of the form

T operator+(T);
T operator-(T);

10 For every promoted integral type T, there exist candidate operator functions of the form

T operator-(T);

11 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, T is an object type or a function type, and CV1 and CV2 are cv-qualifier-seqs, there exist candidate operator functions of the form

CV12 T& operator->*(CV1 C1*, CV2 T C2::*);

where CV12 is the union of CV1 and CV2. The return type is shown for exposition only; see 5.5 for the determination of the operator’s result type.

12 For every pair of promoted arithmetic types L and R, there exist candidate operator functions of the form

LR operator*(L, R);
LR operator/(L, R);
LR operator+(L, R);
LR operator-(L, R);
bool operator<(L, R);
bool operator>(L, R);
bool operator<=(L, R);
bool operator>=(L, R);
bool operator===(L, R);
bool operator!==(L, R);

where LR is the result of the usual arithmetic conversions between types L and R.

13 For every cv-qualified or cv-unqualified object type T there exist candidate operator functions of the form
For every \( T \), where \( T \) is a pointer to object type, there exist candidate operator functions of the form

\[
\text{std::ptrdiff_t } \operatorname{operator-}(T, T);
\]

For every \( T \), where \( T \) is an enumeration type or a pointer type, there exist candidate operator functions of the form

\[
\begin{align*}
\text{bool } & \operatorname{operator<(T, T)}; \\
\text{bool } & \operatorname{operator>(T, T)}; \\
\text{bool } & \operatorname{operator<=(T, T)}; \\
\text{bool } & \operatorname{operator>=(T, T)}; \\
\text{bool } & \operatorname{operator==(T, T)}; \\
\text{bool } & \operatorname{operator!=(T, T)};
\end{align*}
\]

For every pointer to member type \( T \) or type \texttt{std::nullptr_t} there exist candidate operator functions of the form

\[
\begin{align*}
\text{bool } & \operatorname{operator==(T, T)}; \\
\text{bool } & \operatorname{operator!=(T, T)};
\end{align*}
\]

For every pair of promoted integral types \( L \) and \( R \), there exist candidate operator functions of the form

\[
\begin{align*}
\text{\( LR \)} & \operatorname{operator%}(L, R); \\
\text{\( LR \)} & \operatorname{operator&}(L, R); \\
\text{\( LR \)} & \operatorname{operator^}(L, R); \\
\text{\( LR \)} & \operatorname{operator|}(L, R); \\
\text{\( L \)} & \operatorname{operator<}(L, R); \\
\text{\( L \)} & \operatorname{operator>}(L, R);
\end{align*}
\]

where \( LR \) is the result of the usual arithmetic conversions between types \( L \) and \( R \).

For every triple \((L, VQ, R)\), where \( L \) is an arithmetic type, \( VQ \) is either \texttt{volatile} or empty, and \( R \) is a promoted arithmetic type, there exist candidate operator functions of the form

\[
\begin{align*}
\text{\( VQ \ L\& \)} & \operatorname{operator=(VQ \ L\&, R)}; \\
\text{\( VQ \ L\& \)} & \operatorname{operator*=}(VQ \ L\&, R)}; \\
\text{\( VQ \ L\& \)} & \operatorname{operator/=}(VQ \ L\&, R)}; \\
\text{\( VQ \ L\& \)} & \operatorname{operator+=}(VQ \ L\&, R)}; \\
\text{\( VQ \ L\& \)} & \operatorname{operator-=}(VQ \ L\&, R)};
\end{align*}
\]

For every pair \((T, VQ)\), where \( T \) is any type and \( VQ \) is either \texttt{volatile} or empty, there exist candidate operator functions of the form

\[
\begin{align*}
\text{\( T*\ VQ\& \)} & \operatorname{operator=(T*\ VQ\&, T)};
\end{align*}
\]

For every pair \((T, VQ)\), where \( T \) is an enumeration or pointer to member type and \( VQ \) is either \texttt{volatile} or empty, there exist candidate operator functions of the form

\[
\begin{align*}
\text{\( VQ \ T\& \)} & \operatorname{operator=(VQ \ T\&, T)};
\end{align*}
\]

For every pair \((T, VQ)\), where \( T \) is a cv-qualified or cv-unqualified object type and \( VQ \) is either \texttt{volatile} or empty, there exist candidate operator functions of the form

\[
\begin{align*}
\text{\( VQ \ T\& \)} & \operatorname{operator=(VQ \ T\&, T)};
\end{align*}
\]
For every triple \( (L, VQ, R) \), where \( L \) is an integral type, \( VQ \) is either \texttt{volatile} or empty, and \( R \) is a promoted integral type, there exist candidate operator functions of the form

\begin{align*}
VQ L & \text{ operator\%=(VQ L\&, R);} \\
VQ L & \text{ operator\%==(VQ L\&, R);} \\
VQ L & \text{ operator\%===(VQ L\&, R);} \\
VQ L & \text{ operator\%==(VQ L\&, R);}
\end{align*}

There also exist candidate operator functions of the form

\begin{align*}
\text{bool} & \text{ operator!=(bool);} \\
\text{bool} & \text{ operator&&=(bool, bool);} \\
\text{bool} & \text{ operator|=(bool, bool);}
\end{align*}

For every pair of promoted arithmetic types \( L \) and \( R \), there exist candidate operator functions of the form

\begin{align*}
LR & \text{ operator?:=(bool, L, R);} \\
LR & \text{ operator?:=(bool, L, R);} \\
LR & \text{ operator?:=(bool, L, R);}
\end{align*}

where \( LR \) is the result of the usual arithmetic conversions between types \( L \) and \( R \). [\textit{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

\begin{align*}
T & \text{ operator?:=(bool, T, T);} \\
T & \text{ operator?:=(bool, T, T);} \\
T & \text{ operator?:=(bool, T, T);}
\end{align*}
14 Templates

A template defines a family of classes or functions or an alias for a family of types.

```
template-declaration:
    template < template-parameter-list > declaration

template-parameter-list:
    template-parameter
    template-parameter-list , template-parameter
```

[Note: The > token following the template-parameter-list of a template-declaration may be the product of replacing a >> token by two consecutive > tokens (14.2). — end note]

The declaration in a template-declaration shall

1. declare or define a function, a class, or a variable, or
2. define a member function, a member class, a member enumeration, or a static data member of a class template or of a class nested within a class template, or
3. define a member template of a class or class template, or
4. be an alias-declaration.

A template-declaration is a declaration. A template-declaration is also a definition if its declaration 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:

```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]

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 (14.5.5). — end note]
In a template-declaration, explicit specialization, or explicit instantiation the init-declarator-list in the declaration shall contain at most one declarator. When such a declaration is used to declare a class template, no declarator is permitted.

A template name has linkage (3.5). 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 (14.7.3), 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 (3.2). [Note: Default arguments for function templates and for member functions of class templates are considered definitions for the purpose of template instantiation (14.5) and must also obey the one definition rule. —end note]

A class template shall not have the same name as any other template, class, function, variable, enumeration, enumerator, namespace, or type in the same scope (3.3), except as specified in (14.5.5). Except that a function template can be overloaded either by non-template functions (8.3.5) with the same name or by other function templates with the same name (14.8.3), a template name declared in namespace scope or in class scope shall be unique in that scope.

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 (14.7.1) unless the corresponding specialization is explicitly instantiated (14.7.2) in some translation unit; no diagnostic is required.

14.1 Template parameters  [temp.param]

The syntax for template-parameters is:

```
template-parameter:
  type-parameter
  parameter-declaration

  type-parameter:
    type-parameter-key ...opt identifieropt
    type-parameter-key identifieropt = type-id
    template < template-parameter-list > type-parameter-key ...opt identifieropt
    template < template-parameter-list > type-parameter-key identifieropt = id-expression

  type-parameter-key:
    class
typename
```

[Note: The > token following the template-parameter-list of a type-parameter may be the product of replacing a >> token by two consecutive > tokens (14.2). —end note]

There is no semantic difference between class and typename in a type-parameter-key. typename followed by an unqualified-id names a template type parameter. typename followed by a qualified-id denotes the type in a non-type\textsuperscript{136} parameter-declaration. A template-parameter of the form class identifier is a type-parameter.

[Example:]

```c
  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
  }
```

\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.
Here, the template f has a type-parameter called T, rather than an unnamed non-type template-parameter of class T. — end example] A storage class shall not be specified in a template-parameter declaration. Types shall not be defined in a template-parameter declaration.

3 A type-parameter whose identifier does not follow an ellipsis defines its identifier to be a typedef-name (if declared without template) or template-name (if declared with template) in the scope of the template declaration. [Note: A template argument may be a class template or alias template. For example,

```cpp
template<class T> class myarray { /* ... */ };  
```

```cpp
 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,

(4.5) — std::nullptr_t.

5 [Note: Other types are disallowed either explicitly below or implicitly by the rules governing the form of template-arguments (14.3). — end note] The top-level cv-qualifiers on the template-parameter are ignored when determining its type.

6 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& cri = i;  // OK: const reference bound to temporary
 }

— end example]
```

7 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]
```

8 A non-type template-parameter of type “array of T” or “function returning T” is adjusted to be of type “pointer to T” or “pointer to function returning T”, respectively. [Example:
template<int* a> struct R { /* ... */ };  
template<int b[5]> struct S { /* ... */ };  
int p;  
R<&p> w;       // OK  
S<&p> x;       // OK due to parameter adjustment  
int v[5];  
R<v> y;       // OK due to implicit argument conversion  
S<v> z;       // OK due to both adjustment and conversion

— end example ]

9 A default template-argument is a template-argument (14.3) 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 (14.5.3). 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.

10 The set of default template-arguments available for use with a template declaration or definition is obtained by merging the default arguments from the definition (if in scope) and all declarations in scope in the same way default function arguments are (8.3.6). [Example:

    template<class T1, class T2 = int> class A;  
    template<class T1 = int, class T2> class A;  

is equivalent to

    template<class T1 = int, class T2 = int> class A;

— end example ]

11 If a template-parameter of a class 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 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 of the function template or has a default argument (14.8.2). [Example:

    template<class T1 = int, class T2> class B;       // error

    // U can be neither deduced from the parameter-type-list nor specified
    template<class... T, class... U> void f() { }       // error
    template<class... T, class U> void g() { }         // error

— end example ]

12 A template-parameter shall not be given default arguments by two different declarations in the same scope. [ Example:

    template<class T = int> class X;  
    template<class T = int> class X { /* ... */ };       // error

— end example ]

13 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:
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 TT = float> class T {
    inline void f();
    inline void g();
};
template <template <class TT = float> 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 (8.3.5), then the template-parameter is a template parameter pack (14.5.3). 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... T> struct value_holder {
    template<T... Values> struct apply { };  // Values is a non-type template parameter pack
    // and a pack expansion
};
template<class... T, T... Values> struct static_array;  // error: Values expands template type parameter
// pack T within the same template parameter list
```
— end example]

### 14.2 Names of template specializations

A template specialization (14.7) 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 (3.4) 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]

2 For a template-name to be explicitly qualified by the template arguments, the name must be known to refer to a template.

3 After name lookup (3.4) finds that a name is a template-name or that an operator-function-id or a literal-operator-id refers to a set of overloaded functions any member of which is a function template, if this 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 >\textsuperscript{137} is taken as the ending delimiter rather than a greater-than operator. Similarly, the first non-nested >> is treated as two consecutive but distinct > tokens, the first of which is taken as the end of the template-argument-list and completes the template-id. [Note: The second > token produced by this replacement rule may terminate an enclosing template-id construct or it may be part of a different construct (e.g. a cast). — end note] [Example:

    template<int i> class X { /* ... */ };
    X< 1>2 > x1; // syntax error
    X<(1>2)> x2; // OK

    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]

4 When the name of a member template specialization appears after . or -> in a postfix-expression or after a nested-name-specifier in a qualified-id, and the object expression of the postfix-expression is type-dependent or the nested-name-specifier in the qualified-id refers to a dependent type, but the name is not a member of the current instantiation (14.6.2.1), the member template name must be prefixed by the keyword template. Otherwise the name is assumed to name a non-template. [Example:

    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]

\textsuperscript{137} A > that encloses the type-id of a \texttt{dynamic\_cast}, \texttt{static\_cast}, \texttt{reinterpret\_cast} or \texttt{const\_cast}, or which encloses the template-arguments of a subsequent template-id, is considered nested for the purpose of this description.
A name prefixed by the keyword \texttt{template} shall be a \textit{template-id} or the name shall refer to a class template. [\textit{Note}: The keyword \texttt{template} may not be applied to non-template members of class templates. — end note] [\textit{Note}: As is the case with the \texttt{typename} prefix, the \texttt{template} prefix is allowed in cases where it is not strictly necessary; i.e., when the \textit{nested-name-specifier} or the expression on the left of the \texttt{->} or \texttt{.} is not dependent on a \textit{template-parameter}, or the use does not appear in the scope of a template. — end note] [\textit{Example}:

\begin{verbatim}
    template <class T> struct A {
        void f(int);
        template <class U> void f(U);
    };

    template <class T> void f(T t) {
        A<T> a;
        a.template f<>(t);  // OK: calls template
        a.template f(t);   // error: not a template-id
    }

    template <class T> struct B {
        template <class T2> struct C { };
    };

    // OK: \texttt{T::template C} names a class template:
    template <class T, template <class X> class TT = T::template C> struct D { };
    D<B<int> > db;

    // end example]
\end{verbatim}

6 A \textit{simple-template-id} that names a class template specialization is a \textit{class-name} (Clause 9).

7 A \textit{template-id} that names an alias template specialization is a \textit{type-name}.

\section*{14.3 Template arguments} \textmd{[\texttt{temp.arg}]

There are three forms of \textit{template-argument}, corresponding to the three forms of \textit{template-parameter}: type, non-type and template. The type and form of each \textit{template-argument} specified in a \textit{template-id} shall match the type and form specified for the corresponding parameter declared by the template in its \textit{template-parameter-list}. When the parameter declared by the template is a template parameter pack (14.5.3), it will correspond to zero or more \textit{template-arguments}. [\textit{Example}:

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

\textsection{14.3}
\[v2[3] = v3.elem(4) = \text{dcomplex}(7,8);\]

— end example

2 In a template-argument, an ambiguity between a type-id and an expression is resolved to a type-id, regardless of the form of the corresponding template-parameter.\(^{138}\) [Example:

\[
\begin{align*}
\text{template<} & \text{class T> void f();} \\
\text{template<} & \text{int I> void f();} \\
\text{void g() {} & \\
\text{f<} & \text{int>()();} \quad // \text{int() is a type-id: call the first f()} \\
\text{}}
\end{align*}
\]

— end example

3 The name of a template-argument shall be accessible at the point where it is used as a template-argument. \[\text{Note: If the name of the template-argument is accessible at the point where} \]

[the resulting instantiation where the corresponding template-parameter name is used. — end note] [Example:

\[
\begin{align*}
\text{template<} & \text{class T> class X {} \\
\text{static T t;} & \\
\text{}}
\end{align*}
\]

\[
\text{class Y {} \\
\text{private:} \\
\text{struct S \{ */... */};} \\
\text{X<S> x;} \quad // \text{OK: S is accessible} \\
\text{\quad // X<Y::S> has a static member of type Y::S} \\
\text{\quad // OK: even though Y::S is private} \\
\text{}}
\]

\[
\text{X<Y::S> y;} \quad // \text{error: S not accessible}
\]

— 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:

\[
\begin{align*}
\text{template} & \text{<template <} \text{class TT> class T> class A {} \\
\text{typename } & \text{T<int>::S s;} \\
\text{}}
\end{align*}
\]

\[
\text{template <} \text{class U> class B {} \\
\text{private:} \\
\text{struct S \{ */... */};} \\
\text{}}
\]

\[
\text{A<B> b;} \quad // \text{ill-formed: A has no access to B::S}
\]

— 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:

\[\text{§ 14.3}\]

138) 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 = 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 (12.4) for an object that has a type that is a class template specialization may explicitly specify the template-arguments. [Example:

```c
template<class T> struct A {
  ~A();
};
void f(A<int>*, 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 the template in a template-id is an overloaded function template, 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 A template-argument followed by an ellipsis is a pack expansion (14.5.3).

14.3.1 Template type arguments [temp.arg.type]

1 A template-argument for a template-parameter which is a type shall be a type-id.

2 [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 (3.9). — end note]
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]

14.3.2 Template non-type arguments

A template-argument for a non-type template-parameter shall be a converted constant expression (5.20) 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 (1.8),
2. a temporary object (12.2),
3. a string literal (2.13.5),
4. the result of a typeid expression (5.2.8), or
5. a predefined __func__ variable (8.4.1).

[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 (13.4). — end note]

Example:

template<const int* pci> struct X { /* ... */};
int ai[10];
X<ai> xi; // array to pointer and qualification conversions

struct Y { /* ... */};
template<const Y& b> struct Z { /* ... */};
Y y;
Z<y> z; // no conversion, but note extra cv-qualification

template<int (&pa)[5]> struct W { /* ... */};
int b[5];
W<b> w; // no conversion

void f(char);
void f(int);

template<void (*pf)(int)> struct A { /* ... */};
A<&f> a; // selects f(int)

— end example]

3 [Note: A string literal (2.13.5) 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

§ 14.3.2
const char p[] = "Vivisectionist";
X<int,p> x2;  // OK

— end example] — end note]

4 [ 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<&s.m> x4;  // error: address of non-static member
X<&s.s> x5;  // OK: address of static member
X<&S::s> x6;  // OK: address of static member

— end example] — end note]

5 [ 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]

14.3.3 Template template arguments [temp.arg.template]

1 A template-argument for a template template-parameter shall be the name of a class template or an alias template, expressed as id-expression. When the template-argument names a class template, only primary class templates are considered when matching the template template argument with the corresponding parameter; partial specializations are not considered even if their parameter lists match that of the template template parameter.

2 Any partial specializations (14.5.5) 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 is 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;
};
C<A> c;  // V<int> within C<A> uses the primary template,
// so c.y.x has type int
A template-argument matches a template template-parameter \( P \) when each of the template parameters in the template-parameter-list of the template-argument’s corresponding class template or alias template \( A \) matches the corresponding template parameter in the template-parameter-list 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 (14.5.6.1), and for template template-parameters, each of their corresponding template-parameters matches, recursively. When \( P \)'s template-parameter-list contains a template parameter pack (14.5.3), the template parameter pack will match zero or more template parameters or template parameter packs in the template-parameter-list 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<template<class> class P> class X { /* ... */ };  
template<template<class ...> class Q> class Y { /* ... */ };  

X<A> xa;  // OK  
X<B> xb;  // ill-formed: default arguments for the parameters of a template argument are ignored  
X<C> xc;  // ill-formed: a template parameter pack does not match a template parameter  

Y<A> ya;  // OK  
Y<B> yb;  // OK  
Y<C> yc;  // OK  
```

--- end example ---

**Example:**

```cpp
template <class T> struct eval;  
template <template <class, class...> class TT, class T1, class... Rest> struct eval<TT<T1, Rest...>> { };  

template <class T1> struct A;  
template <class T1, class T2> struct B;  
template <int N> struct C;  
template <class T1, int N> struct D;  
template <class T1, class T2, int N = 17> struct E;  

eval<A<int>> eA;  // OK: matches partial specialization of eval  
eval<B<int, float>> eB;  // OK: matches partial specialization of eval  
eval<C<int>> eC;  // error: C does not match TT in partial specialization  
eval<D<int, 17>> eD;  // error: D does not match TT in partial specialization  
eval<E<int, float>> eE;  // error: E does not match TT in partial specialization  
```

--- end example ---

### 14.4 Type equivalence

Two template-ids refer to the same class, function, or variable if
— their template-names, operator-function-ids, or literal-operator-ids refer to the same template and
— their corresponding type template-arguments are the same type and
— their corresponding non-type template arguments of integral or enumeration type have identical values and
— their corresponding non-type template-arguments of pointer type refer to the same object or function or are both the null pointer value and
— 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:

```cpp
template<class E, int size> class buffer { /* ... */ };  
buffer<char,2*512> x; 
buffer<char,1024> y; 
```
declares x and y to be of the same type, and

```cpp
template<class T, void(*err_fct)()> class list { /* ... */ };  
list<int,&error_handler1> x1;  
list<int,&error_handler2> x2;  
list<int,&error_handler2> x3;  
list<char,&error_handler2> x4; 
```
declares x2 and x3 to be of the same type. Their type differs from the types of x1 and x4.

```cpp
template<class T> struct X { };  
template<class> struct Y { };  
X<Y<int> > y;  
X<Z<int> > z; 
```
declares y and z to be of the same type. — end example]

2 If an expression e involves a template parameter, decltype(e) denotes a unique dependent type. Two such decltype-specifiers refer to the same type only if their expressions are equivalent (14.5.6.1). [Note: however, it may be aliased, e.g., by a typedef-name. — end note]

### 14.5 Template declarations [temp.decls]

1 A template-id, that is, the template-name followed by a template-argument-list shall not be specified in the declaration of a primary template declaration. [Example:

```cpp
template<class T1, class T2, int I> class A<T1, T2, I> { }; // error 
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 (14.5.5). — end note]

2 For purposes of name lookup and instantiation, default arguments and exception-specifications of function templates and default arguments and exception-specifications of member functions of class templates are
considered definitions; each default argument or exception-specification is a separate definition which is unrelated to the function template definition or to any other default arguments or exception-specifications.

3 Because an alias-declaration cannot declare a template-id, it is not possible to partially or explicitly specialize an alias template.

14.5.1 Class templates

A class template defines the layout and operations for an unbounded set of related types. [Example: a single class template List might provide a common definition for list of int, list of float, and list of pointers to Shapes. — end example]

[Example: An array class template might be declared like this:

template<class T> class Array {
    T* v;
    int sz;
    public:
        explicit Array(int);
        T& operator[](int);
        T& elem(int i) { return v[i]; }
};

2 The prefix template <class T> specifies that a template is being declared and that a type-name T will be used in the declaration. In other words, Array is a parameterized type with T as its parameter. — end example]

3 When a member function, a member class, a member enumeration, a static data member or a member template of a class template is defined outside of the class template definition, the member definition is defined as a template definition in which the template-parameter names used in the definition of the member may be different from the template parameter names used in the class template definition. The names of the template parameters used in the definition of the member shall be the same as the parameter names used in the class template definition. The template argument list following the class template name in the member definition shall be the same as the one used in the template parameter list of the member. Each template parameter pack shall be expanded with an ellipsis in the template argument list. [Example:

    template<class T1, class T2> struct A {
    void f1();
    void f2();
    };

    template<class T2, class T1> void A<T2,T1>::f1() { } // OK
    template<class T2, class T1> void A<T1,T2>::f2() { } // error

    template<class ... Types> struct B {
    void f3();
    void f4();
    };

    template<class ... Types> void B<Types ...>::f3() { } // OK
    template<class ... Types> void B<Types>::f4() { } // error

— end example]

4 In a redeclaration, partial specialization, explicit specialization or explicit instantiation of a class template, the class-key shall agree in kind with the original class template declaration (7.1.6.3).
14.5.1.1 Member functions of class templates

A member function of a class template may be defined outside of the class template definition in which it is declared. [Example:

```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:

```cpp
template<class T> T& Array<T>::operator[](int i) {
    if (i<0 || sz<=i) error("Array: range error");
    return v[i];
}
```

—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.

```cpp
Array<int> v1(20);
Array<dcomplex> v2(30);

v1[3] = 7;          // Array<int>::operator[]()
v2[3] = dcomplex(7,8); // Array<dcomplex>::operator[]()
```

—end example]

14.5.1.2 Member classes of class templates

A member class of a class template may be defined outside the class template definition in which it is declared. [Note: The member class must be defined before its first use that requires an instantiation (14.7.1). For example,

```cpp
template<class T> struct A {
    class B;
};
A<int>::B* b1;             // OK: requires A to be defined but not A::B
template<class T> class A<T>::B { };  
A<int>::B b2;             // OK: requires A::B to be defined
```

—end note]

14.5.1.3 Static data members of class templates

A definition for a static data member or static data member template may be provided in a namespace scope enclosing the definition of the static member’s class template. [Example:

```cpp
template<class T> class X {
    static T s;
};
template<class T> T X<T>::s = 0;
```
struct limits {
    template<class T>
    static const T min;        // declaration
};

template<class T>
const T limits::min = { };    // definition

— end example ]

2 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:

    template <class T> struct A {
        static int i[];
    };
    template <class T> int A<T>::i[4];     // 4 elements
    template <> int A<int>::i[] = { 1 };   // OK: 1 element

— end example ]

14.5.1.4 Enumeration members of class templates [temp.mem.enum]

An enumeration member of a class template may be defined outside the class template definition. [ Example:

    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 ]

14.5.2 Member templates [temp.mem]

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 the
template-parameters of the class template followed by the template-parameters of the member template.
[ Example:

    template<class T> struct string {
        template<class T2> int compare(const T2&);
        template<class T2> string(const string<T2>& s) { /* ... */ }
    };

    template<class T> template<class T2> int string<T>::compare(const T2& s) {
    }

— end example ]

2 A local class of non-closure type shall not have member templates. Access control rules (Clause 11) apply
to member template names. A destructor shall not be a member template. A non-template member
function (8.3.5) with a given name and type and a member function template of the same name, which could
be used to generate a specialization of the same type, can both be declared in a class. When both exist, a
use of that name and type refers to the non-template member unless an explicit template argument list is
supplied. [ Example:
template <class T> struct A {
    void f(int);
    template <class T2> void f(T2);
};

template <> void A<int>::f(int) { } // non-template member function

int main() {
    A<char> ac;
    ac.f(1);  // non-template
    ac.f('c');  // template
    ac.f<>(1);  // template
}

— end example —

3 A member function template shall not be virtual. [Example:

    template <class T> struct AA {
        template <class C> virtual void g(C);  // error
        virtual void f();  // OK
    };

— end example —

4 A specialization of a member function template does not override a virtual function from a base class. [Example:

    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 —

5 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:

    struct A {
        template <class T> operator T*();
    };

    template <class T> A::operator T*() { return 0; }
    template <> A::operator char*() { return 0; }  // specialization
    template A::operator void*();  // explicit instantiation

int main() {
    A a;
    int* ip;
    ip = a.operator int*();  // explicit call to template operator
                            // A::operator int*()
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 (14.8.2.3), the resulting specialization is used as if found by name lookup.

7 A `using-declaration` in a derived class cannot refer to a specialization of a conversion function template in a base class.

8 Overload resolution (13.3.3.2) and partial ordering (14.5.6.2) are used to select the best conversion function among multiple specializations of conversion function templates and/or non-template conversion functions.

### 14.5.3 Variadic templates

A template parameter pack is a template parameter that accepts zero or more template arguments.  

```cpp
template<class ... Types> struct Tuple { }

Tuple<> t0; // Types contains no arguments
Tuple<int> t1; // Types contains one argument: int
Tuple<int, float> t2; // Types contains two arguments: int and float
Tuple<0> error; // error: 0 is not a type
```

A function parameter pack is a function parameter that accepts zero or more function arguments.  

```cpp
template<class ... Types> void f(Types ... args);

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
```

A parameter pack is either a template parameter pack or a function parameter pack.

A pack expansion consists of a pattern and an ellipsis, the instantiation of which produces zero or more instantiations of the pattern in a list (described below). The form of the pattern depends on the context in which the expansion occurs. Pack expansions can occur in the following contexts:

1. In a function parameter pack (8.3.5); the pattern is the `parameter-declaration` without the ellipsis.
2. In a template parameter pack that is a pack expansion (14.1):
   1. if the template parameter pack is a `parameter-declaration`; the pattern is the `parameter-declaration` without the ellipsis;
   2. if the template parameter pack is a `type-parameter` with a `template-parameter-list`; the pattern is the corresponding `type-parameter` without the ellipsis.
3. In an `initializer-list` (8.5); the pattern is an `initializer-clause`.
4. In a `base-specifier-list` (Clause 10); the pattern is a `base-specifier`.
5. In a `mem-initializer-list` (12.6.2) for a `mem-initializer` whose `mem-initializer-id` denotes a base class; the pattern is the `mem-initializer`.
6. In a `template-argument-list` (14.3); the pattern is a `template-argument`.

§ 14.5.3
— In a dynamic-exception-specification (15.4); the pattern is a type-id.

— In an attribute-list (7.6.1); the pattern is an attribute.

— In an alignment-specifier (7.6.2); the pattern is the alignment-specifier without the ellipsis.

— In a capture-list (5.1.2); the pattern is a capture.

— In a sizeof... expression (5.3.3); the pattern is an identifier.

— In a fold-expression (5.1.3); the pattern is the cast-expression that contains an unexpanded parameter pack.

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.

[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]

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 {};  
template<typename T1, typename T2> struct Pair {}; 

template<class ... Args1> struct zip {
  template<class ... Args2> struct with {
    typedef Tuple<Pair<Args1, Args2> ... > type;
  };
};

typedef zip<short, int>::with<unsigned short, unsigned>::type T1;
// T1 is Tuple<Pair<short, unsigned short>, Pair<int, unsigned> >
typedef zip<short>::with<unsigned short, unsigned>::type T2;
// error: different number of arguments specified for Args1 and Args2

template<class ... Args>
void g(Args ... args) {
  // OK: Args is expanded by the function parameter pack args
  f(const_cast<const Args*>(&args)...); // OK: “Args” and “args” are expanded
  f(5 ...); // error: pattern does not contain any parameter packs
  f(args); // error: parameter pack “args” is not expanded
  f(h(args ...) + args ...); // OK: first “args” expanded within h, second
  // “args” expanded within f
}
```
—end example]
The instantiation of a pack expansion that is neither a `sizeof...` expression nor a `fold-expression` produces a list $E_1, E_2, ..., E_N$, 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:

(7.1) if the pack is a template parameter pack, the element is a template parameter (14.1) of the corresponding kind (type or non-type) designating the type or value from the template argument; otherwise,

(7.2) 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.

All of the $E_i$ become elements in the enclosing list. [Note: The variety of list varies with the context: expression-list, base-specifier-list, template-argument-list, etc. — end note] When $N$ is zero, the instantiation of the expansion produces an empty list. Such an instantiation does not alter the syntactic interpretation of the enclosing construct, even in cases where omitting the list entirely would otherwise be ill-formed or would result in an ambiguity in the grammar. [Example:

```
template<class... T> struct X : T... { };  
template<class... T> void f(T... values) {
  X<T...> x(values...);  
}  

template void f<>();  // OK: X<> has no base classes  
// x is a variable of type X<> that is value-initialized
```

—end example]

The instantiation of a `sizeof...` expression (5.3.3) produces an integral constant containing the number of elements in the parameter pack it expands.

The instantiation of a `fold-expression` produces:

(9.1) $(E_1 \text{ op } E_2 \text{ op } \cdot \cdot \cdot \text{ op } E_N)$ for a unary left fold,

(9.2) $E_1 \text{ op } (\text{ \cdot \cdot \cdot } \text{ op } (E_{N-1} \text{ op } E_N))$ for a unary right fold,

(9.3) $((E \text{ op } E_1) \text{ op } E_2 \text{ op } \cdot \cdot \cdot \text{ op } E_N)$ for a binary left fold, and

(9.4) $E_1 \text{ op } (\text{ \cdot \cdot \cdot } \text{ op } (E_{N-1} \text{ op } (E_N \text{ op } E)))$ for a binary right fold.

In each case, $op$ is the fold-operator, $N$ is the number of elements in the pack expansion parameters, and each $E_i$ is generated by instantiating the pattern and replacing each pack expansion parameter with its $i$th element. For a binary fold-expression, $E$ is generated by instantiating the cast-expression that did not contain an unexpanded parameter pack. [Example:

```
template<typename ...Args>
bool all(Args ...args) { return (... && args); }  

bool b = all(true, true, true, false);  
```

Within the instantiation of `all`, the returned expression expands to `((true && true) && true) && false`, which evaluates to `false`. — end example] If $N$ is zero for a unary fold-expression, the value of the expression is shown in Table 12; if the operator is not listed in Table 12, the instantiation is ill-formed.

14.5.4 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:

§ 14.5.4 352
Table 12 — Value of folding empty sequences

<table>
<thead>
<tr>
<th>Operator</th>
<th>Value when parameter pack is empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>int()</td>
</tr>
<tr>
<td>&amp;</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>void()</td>
</tr>
</tbody>
</table>

(1.1) — if the name of the friend is a qualified or unqualified template-id, the friend declaration refers to a specialization of a function template, otherwise,

(1.2) — if the name of the friend is a qualified-id and a matching non-template function is found in the specified class or namespace, the friend declaration refers to that function, otherwise,

(1.3) — if the name of the friend is a qualified-id and a matching function template is found in the specified class or namespace, the friend declaration refers to the deduced specialization of that function template (14.8.2.6), otherwise,

(1.4) — the name shall be an unqualified-id that declares (or redeclares) a non-template function.

[Example:

```cpp
template<class T> class task;
template<class T> task<T>* preempt(task<T>**);

template<class T> class task {
    friend void next_time();
    friend void process(task<T>**);
    friend task<T>* preempt<T>(task<T>**);
    template<class C> friend int func(C);
    
    friend class task<int>;
    template<class P> friend class frd;
};
```

Here, each specialization of the task class template has the function next_time as a friend; because process does not have explicit template-arguments, each specialization of the task class template has an appropriately typed function process as a friend, and this friend is not a function template specialization; because the friend preempt has an explicit template-argument T, each specialization of the task class template has the appropriate specialization of the function template preempt as a friend; and each specialization of the task class template has all specializations of the function template func as friends. Similarly, each specialization of the task class template has the class template specialization task<int> as a friend, and has all specializations of the class template frd as friends. — end example]

2 A friend template may be declared within a class or class template. A friend function template may be defined within a class or class template, but a friend class template may not be defined in a class or class template. In these cases, all specializations of the friend class or friend function template are friends of the class or class template granting friendship. [Example:

```cpp
class A {
    template<class T> friend class B; // OK
    template<class T> friend void f(T){ /* ... */ } // OK
```
A template friend declaration specifies that all specializations of that template, whether they are implicitly instantiated (14.7.1), partially specialized (14.5.5) or explicitly specialized (14.7.3), 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]

When a function is defined in a friend function declaration in a class template, the function is instantiated when the function is odr-used (3.2). The same restrictions on multiple declarations and definitions that apply to non-template function declarations and definitions also apply to these implicit definitions.

A member of a class template may be declared to be a friend of a non-template class. In this case, the corresponding member of every specialization of the primary class template and class template partial specializations thereof is a friend of the class granting friendship. For explicit specializations and specializations of partial specializations, the corresponding member is the member (if any) that has the same name, kind (type, function, class template, or function template), template parameters, and signature as the member of the class template instantiation that would otherwise have been generated. [Example:

```cpp
template<class T> struct A {
    struct B { }; // grants friendship to A<int>::B even though
    void f(); // it is not a specialization of A<T>::B
    struct D {
        void g(); // does not grant friendship to A<int>::f()
        void g(); // because its return type does not match
    };
};

class C {
    template<class T> friend struct A<T>::B; // grants friendship to A<T>::B
    template<class T> friend void A<T>::f(); // does not grant friendship to A<int>::f()
    template<class T> friend void A<T>::D::g(); // does not grant friendship to A<int>::D::g()
        // because A<int>::D is not a specialization of A<T>::D
};
```
—end example]

[Note: A friend declaration may first declare a member of an enclosing namespace scope (14.6.5). — end note]

§ 14.5.4
A friend template shall not be declared in a local class.

Friend declarations shall not declare partial specializations. [Example:

```cpp
template<class T> class A { };  
class X {
    template<class T> friend class A<T*>;  // error
};
—end example]
```

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.

### 14.5.5 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 (14.5.5.1). 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 (14.5.5.3).

[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
```

The first declaration declares the primary (unspecialized) class template. The second and subsequent declarations declare partial specializations of the primary template. —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<T1, T2, I> { };  // error
—end note]
```

A class template partial specialization may be declared or redeclared in any namespace scope in which the corresponding primary template may be defined (7.3.1.2 and 14.5.2). [Example:

```cpp
   template<class T> struct A {
       struct C {
           template<class T2> struct B { }; 
       }; 
   };
```

§ 14.5.5
Partial specialization declarations themselves are not found by name lookup. Rather, when the primary template name is used, any previously-declared partial specializations of the primary template are also considered. One consequence is that a using-declaration which refers to a class template does not restrict the set of partial specializations which may be found through the using-declaration. [Example:

```cpp
namespace N {
    template<class T1, class T2> class A { };  // primary template
}

using N::A;  // refers to the primary template

namespace N {
    template<class T> class A<T, T*> { };  // partial specialization
}

A<int, int*> a;  // uses the partial specialization, which is found through
                 // the using declaration which refers to the primary template

— end example]

A non-type argument is non-specialized if it is the name of a non-type parameter. All other non-type arguments are specialized.

Within the argument list of a class template partial specialization, the following restrictions apply:

(8.1) A partially specialized non-type argument expression shall not involve a template parameter of the partial specialization except when the argument expression is a simple identifier. [Example:

```cpp
template <int I, int J> struct A {};  
template <int I> struct A<I+5, I*2> {};  // error

template <int I, int J> struct B {};  
template <int I> struct B<I, I> {};  // OK

— end example]
```

(8.2) 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, t>;  // error

template< int X, int (*array_ptr)[X] > class A {};  
int array[5];  
template< int X > class A<X, &array> { };  // error

— end example]
```

(8.3) The argument list of the specialization shall not be identical to the implicit argument list of the primary template.
The specialization shall be more specialized than the primary template (14.5.5.2).

The template parameter list of a specialization shall not contain default template argument values.\textsuperscript{139}

An argument shall not contain an unexpanded parameter pack. If an argument is a pack expansion (14.5.3), it shall be the last argument in the template argument list.

14.5.5.1 Matching of class template partial specializations \[\text{temp.class.spec.match}\]

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.

(1.1) If exactly one matching specialization is found, the instantiation is generated from that specialization.

(1.2) If more than one matching specialization is found, the partial order rules (14.5.5.2) 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.

(1.3) If no matches are found, the instantiation is generated from the primary template.

A partial specialization matches a given actual template argument list if the template arguments of the partial specialization can be deduced from the actual template argument list (14.8.2). \[Example:

\begin{verbatim}
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{verbatim}

\[\text{— end example}\]

A non-type template argument can also be deduced from the value of an actual template argument of a non-type parameter of the primary template. \[Example: the declaration of a2 above. — end example\]

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.

14.5.5.2 Partial ordering of class template specializations \[\text{temp.class.order}\]

For two class template partial specializations, the first is more specialized than the second if, given the following rewrite to two function templates, the first function template is more specialized than the second according to the ordering rules for function templates (14.5.6.2):

(1.1) The first function template has the same template parameters as the first partial specialization and has a single function parameter whose type is a class template specialization with the template arguments of the first partial specialization, and

(1.2) The second function template has the same template parameters as the second partial specialization and has a single function parameter whose type is a class template specialization with the template arguments of the second partial specialization.

\[Example:\]

\textsuperscript{139} There is no way in which they could be used.
The partial specialization #2 is more specialized than the partial specialization #1 because the function template B is more specialized than the function template A according to the ordering rules for function templates. — end example]

14.5.5.3 Members of class template specializations [temp.class.spec.mfunc]

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:

    // primary template
    template<class T, int I> struct A {
        void 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
    }

    — end example]
If a member template of a class template is partially specialized, the member template partial specializations
are member templates of the enclosing class template; if the enclosing class template is instantiated (14.7.1,
14.7.2), a declaration for every member template partial specialization is also instantiated as part of creating
the members of the class template specialization. If the primary member template is explicitly specialized
for a given (implicit) specialization of the enclosing class template, the partial specializations of the member
template are ignored for this specialization of the enclosing class template. If a partial specialization of the
member template is explicitly specialized for a given (implicit) specialization of the enclosing class template,
the primary member template and its other partial specializations are still considered for this specialization
of the enclosing class template. [Example:

```cpp
template<class T> struct A {
    template<class T2> struct B {}; // #1
    template<class T2> struct B<T2*> {}; // #2
};

template<> template<class T2> struct A<short>::B {}; // #3
```

```cpp
A<char>::B<int*> abcip; // uses #2
A<short>::B<int*> absip; // uses #3
A<char>::B<int> abci; // uses #1
```

—end example]

14.5.6 Function templates [temp.fct]

1 A function template defines an unbounded set of related functions. [Example: a family of sort functions
might be declared like this:

```cpp
template<class T> class Array { }
template<class T> void sort(Array<T>&);
```

—end example]

2 A function template can be overloaded with other function templates and with non-template functions (8.3.5).
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.

14.5.6.1 Function template overloading [temp.over.link]

1 It is possible to overload function templates so that two different function template specializations have the
same type. [Example:

```cpp
// file1.c
template<class T>
void f(T*);
void g(int* p) {
    f(p); // calls f<int>(int*)
}
```

```cpp
// file2.c
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 (3.2).

3 The signature of a function template is defined in 1.3. The names of the template parameters are significant
only for establishing the relationship between the template parameters and the rest of the signature. [Note:

140) 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 (3.2) in a program, it must be defined; it will not be implicitly
instantiated using the function template definition.

§ 14.5.6.1
Two distinct function templates may have identical function return types and function parameter lists, even if overload resolution alone cannot distinguish them.

```cpp
template<class T> void f();
template<int I> void f(); // OK: overloads the first template
    // distinguishable with an explicit template argument list
```

—end note—

When an expression that references a template parameter is used in the function parameter list or the return type in the declaration of a function template, the expression that references the template parameter is part of the signature of the function template. This is necessary to permit a declaration of a function template in one translation unit to be linked with another declaration of the function template in another translation unit and, conversely, to ensure that function templates that are intended to be distinct are not linked with one another. [Example:

```cpp
template <int I, int J> A<I+J> f(A<I>, A<J>); // #1
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 (3.2), 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. For determining whether two dependent names (14.6.2) are equivalent, only the name itself is considered, not the result of name lookup in the context of the template. If multiple declarations of the same function template differ in the result of this name lookup, the result for the first declaration is used. [Example:

```cpp
template <int I, int J> void f(A<I+J>); // #1
template <int K, int L> void f(A<K+L>); // same as #1
```

```cpp
template <class T> decltype(g(T())) h();
int g(int);
template <class T> decltype(g(T())) h() // redeclaration of h() uses the earlier lookup
{ return g(T()); } // ...although the lookup here does find g(int)
int i = h<int>(); // template argument substitution fails; g(int)
    // was not in scope at the first declaration of h()
```

—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 function templates are equivalent if they are declared in the same scope, have the same name, have identical template parameter lists, and have return types and parameter lists that are equivalent using the rules described above to compare expressions involving template parameters. Two function templates are functionally equivalent if they are equivalent except that one or more expressions that involve template parameters in the return types and parameter lists are functionally equivalent using the rules described above to compare expressions involving template parameters. If a program contains declarations of function templates that are functionally equivalent but not equivalent, the program is ill-formed; no diagnostic is 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>);

// Guaranteed to be different
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>);
```

— end note ]

14.5.6.2 Partial ordering of function templates [temp.func.order]

If a function template is overloaded, the use of a function template specialization might be ambiguous because template argument deduction (14.8.2) 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 (13.3.3);
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 (3.7.4.2, 5.3.4);
4. when a friend function declaration (14.5.4), an explicit instantiation (14.7.2) or an explicit specialization (14.7.3) 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.

To produce the transformed template, for each type, non-type, or template template parameter (including template parameter packs (14.5.3) 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. 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-qualification of $M$ is && or if $M$ has no ref-qualification and the first parameter of the other template has rvalue reference type. Otherwise, the new parameter is of type “lvalue reference to $cv$ $A$”.

[Note: This allows a non-static member to be ordered with respect to a nonmember function and for the results to be equivalent to the ordering of two equivalent nonmembers. — end note]

[Example:

```cpp
struct A { }
;template<class T> struct B {
    template<class R> int operator*(R&);  // #1
};
template<class T, class R> int operator*(T&, R&);  // #2
```
// The declaration of B::operator* is transformed into the equivalent of
// template<class R> int operator*(B<A>&, R&);  // #1a

int main() {
    A a;
    B<A> b;
    b * a;  // calls #1a
}

— end example]

4 Using the transformed function template’s function type, perform type deduction against the other template as described in 14.8.2.4.

[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 T> 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

14.5.7 Alias templates
[ temp.alias ]

1 A template-declaration in which the declaration is an alias-declaration (Clause 7) declares the identifier to be a 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

§ 14.5.7 363
However, if the template-id is dependent, subsequent template argument substitution still applies to the template-id. [Example:

```cpp
template<typename...> using void_t = void;
template<typename T> void_t<T::foo> f();
f<int>(); // error, int does not have a nested type foo
```

— end example]

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:

```cpp
template <class T> struct A;
template <class T> using B = typename A<T>::U;
template <class T> struct A {
    typedef B<T> U;
};
B<short> b; // error: instantiation of B<short> uses own type via A<short>::U
```

— end example]

14.6 Name resolution

Three kinds of names can be used within a template definition:

(1.1) — The name of the template itself, and names declared within the template itself.

(1.2) — Names dependent on a template-parameter (14.6.2).

(1.3) — Names from scopes which are visible within the template definition.

A name used in a template declaration or definition and that is dependent on a template-parameter is assumed not to name a type unless the applicable name lookup finds a type name or the name is qualified by the keyword typename. [Example:

```cpp
// no B declared here

class X;

// forward declaration of member class

template<class T> class Y {
    class Z;
    void f() {
        X* a1; // declare pointer to X
        T* a2; // declare pointer to T
        Y* a3; // declare pointer to Y<T>
        Z* a4; // declare pointer to Z
        typedef typename T::A TA;
        TA* a5; // declare pointer to T's A
        typename T::A* a6; // declare pointer to T's A
        T::A* a7; // T::A is not a type name:
                    // 'multiply T::A by a7; ill-formed,
                    // no visible declaration of a7
        B* a8; // B is not a type name:
                    // 'multiply B by a8; ill-formed,
                    // no visible declarations of B and a8
```
When a qualified-id is intended to refer to a type that is not a member of the current instantiation (14.6.2.1) 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, the program is ill-formed.

\[
\text{typename-specifier: }
\begin{align*}
\text{typename} & \quad \text{nested-name-specifier} & \quad \text{identifier} \\
\text{typename} & \quad \text{nested-name-specifier} & \quad \text{template} & \quad \text{opt} & \quad \text{simple-template-id}
\end{align*}
\]

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, the specialization is ill-formed. The usual qualified name lookup (3.4.3) is used to find the qualified-id even in the presence of typename. [Example:

\begin{verbatim}
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
}
\end{verbatim}

— end example]

A qualified name used as the name in a mem-initializer-id, a base-specifier, 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, 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:

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

§ 14.6
int main() {
    f<Bar>(1);    // OK
    f<Foo>(1);    // error: Foo::x is a type
}

7 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. [Note: such names can be found using unqualified name
lookup (3.4.1), class member lookup (3.4.3.1) into the current instantiation (14.6.2.1), or class member access
expression lookup (3.4.5) when the type of the object expression is the current instantiation (14.6.2.2). — end
note] [Example:

    template<class T> struct A {
        typedef int B;
        B b;              // OK, no typename required
    };

    — end example]

8 Knowing which names are type names allows the syntax of every template to be checked. No diagnostic
shall be issued for a template for which a valid specialization can be generated. If no valid specialization can
be generated for a template, and that template is not instantiated, the template is ill-formed, no diagnostic
required. If every valid specialization of a variadic template requires an empty template parameter pack,
the template is ill-formed, no diagnostic required. If 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 param-
eter, the program is ill-formed; no diagnostic is required. If 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, the program is ill-formed; no diagnostic is required. [Note: This can happen
in situations including the following:

(8.1) — a type used in a non-dependent name is incomplete at the point at which a template is defined but is
complete at the point at which an instantiation is performed, or

(8.2) — an instantiation uses a default argument or default template argument that had not been defined at
the point at which the template was defined, or

(8.3) — constant expression evaluation (5.20) within the template instantiation uses

(8.3.1) — the value of a const object of integral or unscoped enumeration type or

(8.3.2) — the value of a constexpr object or

(8.3.3) — the value of a reference or

(8.3.4) — the definition of a constexpr function,

and that entity was not defined when the template was defined, or

(8.4) — 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] [Note: If a template is instantiated, errors will be diagnosed according to the other rules in
this Standard. Exactly when these errors are diagnosed is a quality of implementation issue. — end note]
[Example:
int j;
template<class T> class X {
    void f(T t, int i, char* p) {
        t = i; // diagnosed if X::f is instantiated
            // and the assignment to t is an error
        p = i; // may be diagnosed even if X::f is
            // not instantiated
        p = j; // may be diagnosed even if X::f is
            // not instantiated
    }
    void g(T t) {
        t++; // may be diagnosed even if X::g is
            // not instantiated
    }
};
template<class... T> struct A {
    void operator++(int, T... t); // error: too many parameters
};
template<class... T> union X : T... { }; // error: union with base class
template<class... T> struct A : T..., T... { }; // error: duplicate base class

—end example

When looking for the declaration of a name used in a template definition, the usual lookup rules (3.4.1, 3.4.2) are used for non-dependent names. The lookup of names dependent on the template parameters is postponed until the actual template argument is known (14.6.2). [Example:

```cpp
#include <iostream>
using namespace std;

template<class T> class Set {
    T* p;
    int cnt;
    public:
    Set();
    Set<T>(const Set<T>&);
    void printall() {
        for (int i = 0; i<cnt; i++)
            cout << p[i] << 'n';
    }
};
```

in the example, i is the local variable i declared in printall, cnt is the member cnt declared in Set, and cout is the standard output stream declared in iostream. However, not every declaration can be found this way; the resolution of some names must be postponed until the actual template-arguments are known. For example, even though the name operator<< is known within the definition of printall() and a declaration of it can be found in <iostream>, the actual declaration of operator<< needed to print p[i] cannot be known until it is known what type T is (14.6.2). —end example]

If a name does not depend on a template-parameter (as defined in 14.6.2), a declaration (or set of declarations) for that name shall be in scope at the point where the name appears in the template definition; the name is bound to the declaration (or declarations) found at that point and this binding is not affected by declarations that are visible at the point of instantiation. [Example:

```cpp
    void f(char);
```
template<class T> void g(T t) {
    f(1);       // f(char)
    f(T(1));    // dependent
    f(t);      // dependent
    dd++;      // not dependent
            // error: declaration for dd not found
}

enum E { e };    void f(E);

double dd;
void h() {
    g(e);        // will cause one call of f(char) followed
                     // by two calls of f(E)
    g('a');      // will cause three calls of f(char)
}

— end example

11 [ Note: For purposes of name lookup, default arguments and exception-specifications of function templates and default arguments and exception-specifications of member functions of class templates are considered definitions (14.5). — end note]

14.6.1 Locally declared names [temp.local]

1 Like normal (non-template) classes, class templates have an injected-class-name (Clause 9). 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 <>.

2 Within the scope of a class template specialization or partial specialization, when the injected-class-name is used as a type-name, it is equivalent to the template-name followed by the template-arguments of the class template specialization or partial specialization enclosed in <>.

3 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. [ Example:

    template<template<class> class T> class A { };    
    template<class T> class Y;
    template<> class Y<int> {
        Y* p;       // meaning Y<int>
        Y<char>* q; // meaning Y<char>
        A<Y>* a;    // meaning A<>>::Y
    };

    class B {
        template<class T> friend class Y;  // meaning ::Y
    };

    — end example]
template<class T, template<class> class U = T::template Base> struct Third { };  
Third<Base<int>> t; // OK: default argument uses injected-class-name as a template

— end example ]

4 A lookup that finds an injected-class-name (10.2) 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. [Example:

```cpp
template <class T> struct Base { }

template <class T> struct Derived: Base<int>, Base<char> {
  typename Derived::Base b; // error: ambiguous
  typename Derived::Base<double> d; // OK
};
```
— end example ]

5 When the normal name of the template (i.e., the name from the enclosing scope, not the injected-class-name) is used, it always refers to the class template itself and not a specialization of the template. [ Example:

```cpp
template<class T> class X {
  X* p; // meaning X<T>
  X<T>* p2;
  X<int>* p3;
  ::X* p4; // error: missing template argument list
  // ::X does not refer to the injected-class-name
};
```
— end example ]

6 A template-parameter shall not be redeclared within its scope (including nested scopes). A template-parameter shall not have the same name as the template name. [ Example:

```cpp
template<class T, int i> class Y {
  int T; // error: template-parameter redeclared
  void f() {
    char T; // error: template-parameter redeclared
  }
};
```

```cpp
template<class X> class X; // error: template-parameter redeclared
```
— end example ]

7 In the definition of a member of a class template that appears outside of the class template definition, the name of a member of the class template hides the name of a template-parameter of any enclosing class templates (but not a template-parameter of the member if the member is a class or function template). [ Example:

```cpp
template<class T> struct A {
  struct B { /* ... */ }
  typedef void C;
  void f();
  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 (14.6.2.1), 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 (3.3.10). [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 —

14.6.2 Dependent names [temp.dep]

Inside a template, some constructs have semantics which may differ from one instantiation to another. Such a construct depends on the template parameters. In particular, types and expressions may depend on the type and/or value of template parameters (as determined by the template arguments) and this determines the context for name lookup for certain names. Expressions may be type-dependent (on the type of a template parameter) or value-dependent (on the value of a non-type template parameter). In an expression of the form:

    postfix-expression ( expression-list_opt )

where the postfix-expression is an unqualified-id, the unqualified-id denotes a dependent name if

(1.1) — any of the expressions in the expression-list is a pack expansion (14.5.3),
— any of the expressions or \texttt{braced-init-lists} in the \texttt{expression-list} is type-dependent (14.6.2.2), or

— if the \texttt{unqualified-id} is a \texttt{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 (14.6.4.1) in both the context of the template definition and the context of the point of instantiation.

\texttt{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 \texttt{B<T>}, the type name \texttt{T::A}, the names \texttt{B<T>::i} and \texttt{pb->j} explicitly depend on the \texttt{template-parameter}. —end example\]

\texttt{In the definition of a class or class template, the scope of a dependent base class (14.6.2.1) is not examined during unqualified name lookup either at the point of definition of the class template or member or during an instantiation of the class template or member. [Example:}

```cpp
typedef double A;
template<class T> class B {
    typedef int A;
};
template<class T> struct X : B<T> {
    A a; // a has type double
};
```

The type name \texttt{A} in the definition of \texttt{X<T>} binds to the typedef name defined in the global namespace scope, not to the typedef name defined in the base class \texttt{B<T>}. —end example] [Example:

```cpp
struct A {
    struct B { /* ... */ ;
        int a;
        int Y;
    };

    int a;

template<class T> struct Y : T {
    struct B { /* ... */ ;
        B b; // The B defined in Y
        void f(int i) { a = i; } // :a
        Y* p; // Y<T>
    };

    Y<A> ya;
```

The members \texttt{A::B}, \texttt{A::a}, and \texttt{A::Y} of the template argument \texttt{A} do not affect the binding of names in \texttt{Y<A>}. —end example]
14.6.2.1 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 9) 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 nth template parameter is a parameter pack, the nth template argument is a pack expansion (14.5.3) 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 nth template argument has the value of the nth template parameter of the class template. If the nth template parameter is a template parameter pack (14.5.3), the nth template argument is a pack expansion (14.5.3) whose pattern is the name of the template parameter pack.

3 A template argument that is equivalent to a template parameter (i.e., has the same constant value or the same type as the template parameter) can be used in place of that template parameter in a reference to the current instantiation. In the case of a non-type template argument, the argument must have been given the value of the template parameter and not an expression in which the template parameter appears as a subexpression. [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; // refers to 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;
    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:

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 (5.2.5) for which the type of the object expression is the current instantiation, and the id-expression, when looked up (3.4.5), refers to at least one member of a class that is the current instantiation or a non-dependent base class thereof.  

[Note: if no such member is found, and the current instantiation has any dependent base classes, then the id-expression is a member of an unknown specialization; see below. — end note]

[Example:

template <class T> class A {
    static const int i = 5;
    int n1[i]; // i refers to a member of the current instantiation
    int n2[A::i]; // A::i refers to a member of the current instantiation
    int n3[A<T>::i]; // A<T>::i refers to a member of the current instantiation
    int f();
};

template <class T> int A<T>::f() {
    return i; // i refers to a member of the current instantiation
}

— end example]

A name is a dependent member of the current instantiation if it is a member of the current instantiation that, when looked up, refers to at least one member of a class that is the current instantiation.

§ 14.6.2.1
A name is a *member of an unknown specialization* if it is

(6.1) — A *qualified-id* in which the *nested-name-specifier* names a dependent type that is not the current instantiation.

(6.2) — A *qualified-id* in which the *nested-name-specifier* refers to the current instantiation, the current instantiation has at least one dependent base class, and name lookup of the *qualified-id* does not find any member of a class that is the current instantiation or a non-dependent base class thereof.

(6.3) — An *id-expression* denoting the member in a class member access expression (5.2.5) in which either

(6.3.1) — the type of the object expression is the current instantiation, the current instantiation has at least one dependent base class, and name lookup of the *id-expression* does not find a member of a class that is the current instantiation or a non-dependent base class thereof; or

(6.3.2) — the type of the object expression is 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]

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. [Note: the result of name lookup differs only when the member of the current instantiation was found in a non-dependent base class of the current instantiation and a member with the same name is also introduced by the substitution for a dependent base class of the current instantiation. — end note]

A type is dependent if it is

(9.1) — a template parameter,

(9.2) — a member of an unknown specialization,

(9.3) — a nested class or enumeration that is a dependent member of the current instantiation,

(9.4) — a cv-qualified type where the cv-unqualified type is dependent,

(9.5) — a compound type constructed from any dependent type,

(9.6) — an array type constructed from any dependent type or whose size is specified by a constant expression that is value-dependent,
1. 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

2. denoted by `decltype(expression)`, where `expression` is type-dependent (14.6.2.2).

10 [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]

14.6.2.2 Type-dependent expressions

1 Except as described below, an expression is type-dependent if any subexpression is type-dependent.

2 `this` is type-dependent if the class type of the enclosing member function is dependent (14.6.2.1).

3 An `id-expression` is type-dependent if it contains

- an identifier associated by name lookup with one or more declarations declared with a dependent type,
- 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 (7.1.6.4),
- the identifier `__func__` (8.4.1), where any enclosing function is a template, a member of a class template, or a generic lambda,
- a `template-id` that is dependent,
- a `conversion-function-id` that specifies a dependent type, or
- a `nested-name-specifier` or a `qualified-id` that names a member of an unknown specialization;

or if it names a dependent member of the current instantiation that is a static data member of type “array of unknown bound of T” for some T (14.5.1.3). 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:

- `simple-type-specifier (expression-list_opt)`
- `::opt new new-placement_opt new-type-id new-initializer_opt`
- `::opt new new-placement_opt (type-id) new-initializer_opt`
- `dynamic_cast < type-id > (expression)`
- `static_cast < type-id > (expression)`
- `const_cast < type-id > (expression)`
- `reinterpret_cast < type-id > (cast-expression)`

4 Expressions of the following forms are never type-dependent (because the type of the expression cannot be dependent):

- `literal`
- `postfix-expression . pseudo-destructor-name`
- `postfix-expression -> pseudo-destructor-name`
- `sizeof unary-expression`
- `sizeof (type-id)`
- `sizeof ... (identifier)`
- `alignof (type-id)`
- `typeid (expression)`
- `typeid (type-id)`
- `::opt delete cast-expression`
- `::opt delete [ ] cast-expression`
- `throw assignment-expression_opt`
A class member access expression (5.2.5) 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 \texttt{x.y} or \texttt{xp->y} the type of the expression is usually the type of the member \texttt{y} of the class of \texttt{x} (or the class pointed to by \texttt{xp}). However, if \texttt{x} or \texttt{xp} refers to a dependent type that is not the current instantiation, the type of \texttt{y} is always dependent. If \texttt{x} or \texttt{xp} refers to a non-dependent type or refers to the current instantiation, the type of \texttt{y} is the type of the class member access expression. —end note]

A \textit{braced-init-list} is type-dependent if any element is type-dependent or is a pack expansion.

A \textit{fold-expression} is type-dependent.

### 14.6.2.3 Value-dependent expressions

C Except as described below, a constant expression is value-dependent if any subexpression is value-dependent.

An \textit{id-expression} is value-dependent if:

1. it is a name declared with a dependent type,
2. it is the name of a non-type template parameter,
3. it names a member of an unknown specialization,
4. it names a static data member that is a dependent member of the current instantiation and is not initialized in a \textit{member-declarator},
5. it names a static member function that is a dependent member of the current instantiation, or
6. it 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 \textit{unary-expression} or \textit{expression} is type-dependent or the \textit{type-id} is dependent:

\begin{verbatim}
sizeof unary-expression
sizeof (type-id)
typeid (expression)
typeid (type-id)
alignof (type-id)
noexcept (expression)
\end{verbatim}

[Note: For the standard library macro \texttt{offsetof}, see 18.2. —end note]

Expressions of the following form are value-dependent if either the \textit{type-id} or \textit{simple-type-specifier} is dependent or the \textit{expression} or \textit{cast-expression} is value-dependent:

\begin{verbatim}
simple-type-specifier (expression-listopt)
static_cast <type-id> (expression)
const_cast <type-id> (expression)
reinterpret_cast <type-id> (expression)
(type-id) cast-expression
\end{verbatim}

Expressions of the following form are value-dependent:

\begin{verbatim}
sizeof ... (identifier)
fold-expression
\end{verbatim}

An expression of the form \&\texttt{qualified-id} where the \textit{qualified-id} names a dependent member of the current instantiation is value-dependent.
14.6.2.4 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.

14.6.3 Non-dependent names

Non-dependent names used in a template definition are found using the usual name lookup and bound at the point they are used. [Example:

```cpp
void g(double);
void h();

template<class T> class Z {
    public:
        void f() {
            g(1); // calls g(double)
            h++;
            // ill-formed: cannot increment function;
            // this could be diagnosed either here or
            // at the point of instantiation
        }
    }

    void g(int);  // not in scope at the point of the template
                  // definition, not considered for the call g(1)

    — end example]
```

14.6.4 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 (14.6.4.1) and from the definition context.

14.6.4.1 Point of instantiation

For a function template specialization, a member function template specialization, or a specialization for a member function or static data member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization and the context from which it is referenced depends on a template parameter, the point of instantiation of the specialization is the point of instantiation of the enclosing specialization. Otherwise, the point of instantiation for such a specialization immediately follows the namespace scope declaration or definition that refers to the specialization.

If a function template or member function of a class template is called in a way which uses the definition of a default argument of that function template or member function, the point of instantiation of the default argument is the point of instantiation of the function template or member function specialization.

For an exception-specification of a function template specialization or specialization of a member function of a class template, if the exception-specification 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 \textit{exception-specification} is the point of instantiation of the specialization that requires it. Otherwise, the point of instantiation for such an \textit{exception-specification} immediately follows the namespace scope declaration or definition that requires the \textit{exception-specification}.

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.

If a virtual function is implicitly instantiated, its point of instantiation is immediately following the point of instantiation of its enclosing class template specialization.

An explicit instantiation definition is an instantiation point for the specialization or specializations specified by the explicit instantiation.

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 (3.2), the program is ill-formed, no diagnostic required.

14.6.4.2 \textbf{Candidate functions} \[\text{temp.dep.candidate}\]

For a function call where the \textit{postfix-expression} is a dependent name, the candidate functions are found using the usual lookup rules (3.4.1, 3.4.2) except that:

(1.1) For the part of the lookup using unqualified name lookup (3.4.1), only function declarations from the template definition context are found.

(1.2) For the part of the lookup using associated namespaces (3.4.2), 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.

14.6.5 \textbf{Friend names declared within a class template} \[\text{temp.inject}\]

Friend classes or functions can be declared within a class template. When a template is instantiated, the names of its friends are treated as if the specialization had been explicitly declared at its point of instantiation.

As with non-template classes, the names of namespace-scope friend functions of a class template specialization are not visible during an ordinary lookup unless explicitly declared at namespace scope (11.3). Such names may be found under the rules for associated classes (3.4.2).\footnote{Friend declarations do not introduce new names into any scope, either when the template is declared or when it is instantiated.} Example:
```cpp
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 ]

### 14.7 Template instantiation and specialization

1 The act of instantiating a function, a class, a member of a class template or a member template is referred to as **template instantiation**.

2 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.

3 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 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
  // U is deduced from the parameter type
template<> void g<int>(int) { } // specialize for U == int
template<> int A<char>::x = 0; // specialize for T == char

template<class T = int> struct B {
  static int x;
};
template<> int B<>::x = 1; // specialize for T == int
```

— end example ]

4 An instantiated template specialization can be either implicitly instantiated (14.7.1) for a given argument list or be explicitly instantiated (14.7.2). A specialization is a class, function, or class member that is either instantiated or explicitly specialized (14.7.3).

5 For a given template and a given set of `template-arguments`,
— an explicit instantiation definition shall appear at most once in a program,

— an explicit specialization shall be defined at most once in a program (according to 3.2), and

— 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 Each class template specialization instantiated from a template has its own copy of any static members.

[Example:

```c
template<class T> class X {
  static T s;
};
template<class T> T X<T>::s = 0;
X<int> aa;
X<char*> bb;
```

X<int> has a static member s of type int and X<char*> has a static member s of type char*. — end example]

### 14.7.1 Implicit instantiation

1 Unless a class template specialization has been explicitly instantiated (14.7.2) or explicitly specialized (14.7.3), 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: Within a template declaration, a local class 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, exception-specifications, and non-static data member initializers, if any). 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 exception-specifications of the class member functions, member classes, scoped member enumerations, static data members and member templates; 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 of a member is valid according to 9.2, a declaration that corresponds to a definition in the template is considered to be a definition. [Example:

```c
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. — end example]

2 Unless a member of a class template or a member template has been explicitly instantiated or explicitly specialized, the specialization of the member is implicitly instantiated when the specialization is referenced
in a context that requires the member definition to exist; 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.

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. 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.

[Example:

```cpp
template<class T> struct Z {
    void f();
    void g();
};

void h() {
    Z<int> a;              // instantiation of class Z<int> required
    Z<char>* p;            // instantiation of class Z<char> not required
    Z<double>* q;          // instantiation of class Z<double> not required

    a.f();                 // instantiation of Z<int>::f() required
    p->g();                // instantiation of class Z<char> required, and
                            // instantiation of Z<char>::g() required
}
```

Nothing in this example requires class Z<double>, Z<int>::g(), or Z<char>::f() to be implicitly instantiated. —end example]

Unless a variable template specialization has been explicitly instantiated or explicitly specialized, the variable template specialization is implicitly instantiated when the specialization is used. 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.

A class template specialization is implicitly instantiated if the class type is used in a context that requires a completely-defined object type or if the completeness of the class type might affect 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 conversion between pointer to class types 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
}
```

§ 14.7.1 381
If the overload resolution process can determine the correct function to call without instantiating a class template definition, it is unspecified whether that instantiation actually takes place. [Example:

```cpp
template <class T> struct S {
    operator int();
};
void f(int);
void f(S<int>&);
void f(S<float>);
void g(S<int>& sr) {
    f(sr);  // instantiation of S<int> allowed but not required
    f(S<float>& sr);  // instantiation of S<float> allowed but not required
};
```
—end example]

If an implicit instantiation of a class template specialization is required and the template is declared but not defined, the program is ill-formed. [Example:

```cpp
template<class T> class X;
X<char> ch;  // error: definition of X required
```
—end example]

The implicit instantiation of a class template does not cause any static data members of that class to be implicitly instantiated.

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 (14.8.3).

An implementation shall not implicitly instantiate a function template, a variable template, a member template, a non-virtual member function, a member class, or a static data member of a class template that does not require instantiation. 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;
```
a call of \texttt{lt.get()} from \texttt{Map<const char*,int>::get()} would place \texttt{List<int>::get()} in the namespace \texttt{N} rather than in the global namespace. — end example] 

13 If a function template \texttt{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 \texttt{f} used at that point, except that the scope in which a closure type is declared (5.1.2) – and therefore its associated namespaces – remain as determined from the context of the definition for the default argument. This analysis is called \textit{default argument instantiation}. The instantiated default argument is then used as the argument of \texttt{f}.

14 Each default argument is instantiated independently. [\textit{Example:}
\begin{verbatim}
template<class T> void f(T x, T y = ydef(T()), T z = zdef(T()));

class A { }
A zdef(A);

void g(A a, A b, A c) {
    f(a, b, c);            // no default argument instantiation
    f(a, b);              // default argument \texttt{z} = \texttt{zdef(T())} instantiated
    f(a);                 // ill-formed; \texttt{ydef} is not declared
}
\end{verbatim}
— end example]

15 The \textit{exception-specification} of a function template specialization is not instantiated along with the function declaration; it is instantiated when needed (15.4). If such an \textit{exception-specification} 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 \textit{exception-specification} is done as if it were being done as part of instantiating the declaration of the specialization at that point.

16 [\textit{Note: 14.6.4.1 defines the point of instantiation of a template specialization. — end note}] 

17 There is an implementation-defined quantity that specifies the limit on the total depth of recursive instantiations, which could involve more than one template. The result of an infinite recursion in instantiation is undefined. [\textit{Example:}
\begin{verbatim}
template<class T> class X {
    X<T>* p;            // OK
    X<T**> a;           // implicit generation of \texttt{X<T>} requires
                      // the implicit instantiation of \texttt{X<T**>} which requires
                      // the implicit instantiation of \texttt{X<T***>} which ...
};
\end{verbatim}
— end example]
14.7.2 Explicit instantiation

1 A class, function, variable, or member template specialization can be explicitly instantiated from its template. A member function, member class or static data member of a class template can be explicitly instantiated from the member definition associated with its class template. An explicit instantiation of a function template or member function of a class template shall not use the `inline` or `constexpr` specifiers.

2 The syntax for explicit instantiation is:

```cpp
explicit-instantiation:
  extern_opt template declaration
```

There are two forms of explicit instantiation: an explicit instantiation definition and an explicit instantiation declaration. An explicit instantiation declaration begins with the `extern` keyword.

3 If the explicit instantiation is for a class or member class, the `elaborated-type-specifier` in the declaration shall include a `simple-template-id`. If the explicit instantiation is for a function or member function, the `unqualified-id` in the declaration shall be either a `template-id` or, where all template arguments can be deduced, a `template-name` or `operator-function-id`. [Note: The declaration may declare a `qualified-id`, in which case the `unqualified-id` of the `qualified-id` must be a `template-id`. — end note] If the explicit instantiation is for a member function, a member class or a static data member of a class template specialization, the name of the class template specialization in the `qualified-id` for the member name shall be a `simple-template-id`. If the explicit instantiation is for a variable, the `unqualified-id` in the declaration shall be a `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 (7.3.1), any namespace from its enclosing namespace set. [Note: Regarding qualified names in declarators, see 8.3. — end note] [Example:

```cpp
template<class T> class Array { void mf(); }
template class Array<char>;template void Array<int>::mf();

template<class T> void sort(Array<T>& v) { /* ... */ }
template void sort(Array<char>&);  // argument is deduced here

namespace N {
template<class T> void f(T&) { }
}
template void N::f<int>(int&);
```

— end example]

4 A declaration of a function template, a variable template, a member function or static data member of a class template, or a member function template of a class or class template shall precede an explicit instantiation of that entity. A definition of a class template, a member class of a class template, or a member class template of a class or class template shall precede an explicit instantiation of that entity unless the explicit instantiation is preceded by an explicit specialization of the entity with the same template arguments. If the `declaration` of the explicit instantiation names an implicitly-declared special member function (Clause 12), the program is ill-formed.

5 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.

6 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 class Y<int>; // error: class template Y not visible

using N::Y;
template class Y<int>; // error: explicit instantiation outside of the

// namespace of the template

template class N::Y<char*>;

// OK: explicit instantiation in namespace N

template void N::Y<double>::mf(); // OK: explicit instantiation

— end example |

7 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

// instantiate sort(Array<int>&) - template-argument deduced
template void sort<> (Array<int>&); // OK: explicit instantiation

— end example |

8 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, except as described below. [ Note: In addition, it will typically be

an explicit instantiation of certain implementation-dependent data about the class. — end note ]

9 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.

10 Except for inline functions, declarations with types deduced from their initializer or return value (7.1.6.4),

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 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 (3.2) 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 ]

11 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 (14.7.1) in the translation unit shall be the subject of an explicit instantiation definition some-

where 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

§ 14.7.2 385
the body. —end note] An explicit instantiation declaration shall not name a specialization of a template with internal linkage.

12 The usual access checking rules do not apply to names used to specify explicit instantiations. [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 which would normally not be accessible and the template may be a member template or member function which would not normally be accessible. —end note]

13 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]

14.7.3 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:

`explicit-specialization:
`template <= > declaration`

[Example:

```c
template<class T> class stream;
template<> class stream<char> { /* ... */ };
template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v) { /* ... */ }
template<> void sort<char*>(Array<char*>&) ;
```

Given these declarations, `stream<char>` will be used as the definition of streams of `chars`; other streams will be handled by class template specializations instantiated from the class template. Similarly, `sort<char*>` will be used as the sort function for arguments of type `Array<char*>`; other `Array` types will be sorted by functions generated from the template. —end example]

§ 14.7.3 386
An explicit specialization shall be declared in a namespace enclosing the specialized template. An explicit specialization whose declarator-id is not qualified shall be declared in the nearest enclosing namespace of the template, or, if the namespace is inline (7.3.1), any namespace from its enclosing namespace set. Such a declaration may also be a definition. If the declaration is not a definition, the specialization may be defined later (7.3.1.2).

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:

```cpp
template<> class X<int> { /* ... */ }; // error: X not a template
template<class T> class X;

template<> class X<char*> { /* ... */ }; // OK: X is a template
```
—end example]

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 12), the program is ill-formed.

A member of an explicitly specialized class is not implicitly instantiated from the member declaration of the class template; instead, the member of the class template specialization shall itself be explicitly defined if its definition is required. In this case, the definition of the class template explicit specialization shall be in scope at the point at which the member is defined. The definition of an explicitly specialized class is unrelated to the definition of a generated specialization. That is, its members need not have the same names, types, etc. as the members of a generated specialization. Members of an explicitly specialized class template are defined in the same manner as members of normal classes, and not using the template<> syntax. The same is true when defining a member of an explicitly specialized member class. However, template<> is used in defining a member of an explicitly specialized member class template that is specialized as a class template. [Example:

```cpp
template<class T> struct A {
   struct B { };  
   template<class U> struct C { };  
};  

template<> struct A<int> {
   void f(int);
};

void h() {
   A<int> a;
   a.f(16); // A<int>::f must be defined somewhere
}
```

// template<> not used for a member of an explicitly specialized class template

// template<> struct A<char>::B {  

§ 14.7.3 387
void f();
};
// template<> also not used when defining a member of an explicitly specialized member class
void A<char>::B::f() { /* ... */ }

template<> template<class U> struct A<char>::C {
  void f();
};
// template<> is used when defining a member of an explicitly specialized member class template specialized as a class template
template<>
template<class U> void A<char>::C<U>::f() { /* ... */ }

template<> struct A<short>::B {
  void f();
};
template<> void A<short>::B::f() { /* ... */ } // error: template<> not permitted

template<> template<class U> struct A<short>::C {
  void f();
};
template<class U> void A<short>::C<U>::f() { /* ... */ } // error: template<> required
— end example

6 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:

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 }; // well-formed, A<char>::E was instantiated
// when A<char> was instantiated

template<class T> enum A<T>::S : T { sT };
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 function templates of member classes of class templates, static data member templates of non-template classes, member function templates of member classes of non-template classes, 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:

namespace N {
    template<class T> class X { /* ... */ };  // OK: specialization
    template<class T> class Y { /* ... */ };  // in same namespace

    template<> class X<int> { /* ... */ };  // forward declare intent to
    // specialize for double

}  // OK: specialization

template<> class N::Y<double> {   // in enclosing namespace
    /* ... */

}  // OK: specialization

template<> class N::Y<short> {   // 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 (3.9). [ Example:

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>

— end example]

10 A trailing template-argument can be left unspecified in the template-id naming an explicit function template specialization provided it can be deduced from the function argument type. [ Example:

template<class T> class Array { /* ... */ };  // explicit specialization for sort(Array<int>&)
    template<class T> void sort(Array<T>& v);  // with deduced template-argument of type int
    template<> void sort(Array<int>&);
A function with the same name as a template and a type that exactly matches that of a template special-
ization is not an explicit specialization (14.5.6).

An explicit specialization of a function template is inline only if it is declared with the inline specifier or
defined as deleted, and independently of whether its function template is inline. [Example:

```
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:

```
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:

```
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]

§ 14.7.3
A member or a member template may be nested within many enclosing class templates. In an explicit specialization for such a member, the member declaration shall be preceded by a template<> for each enclosing class template that is explicitly specialized. [Example:

```cpp
template<class T1> class A {
    template<class T2> class B {
        void mf();
    };
    template<> template<> class A<int>::B<double>;
    template<> template<> void A<char>::B<char>::mf();

    — end example]

In an explicit specialization declaration for a member of a class template or a member template that appears in namespace scope, the member template and some of its enclosing class templates may remain unspecialized, except that the declaration shall not explicitly specialize a class member template if its enclosing class templates are not explicitly specialized as well. In such explicit specialization declaration, the keyword template followed by a template-parameter-list shall be provided instead of the template<> preceding the explicit specialization declaration of the member. The types of the template-parameters in the template-parameter-list shall be the same as those specified in the primary template definition. [Example:

```cpp
template <class T1> class A {
    template<class T2> class B {
        template<class T3> void mf1(T3);
        void mf2();
    };
    template <> template <class X>
    class A<int>::B {
        template <class T> void mf1(T);
    }
    template <> template <> template<class T>
    void A<int>::B<double>::mf1(T t) { } // ill-formed; B<double> is specialized but
    template <class Y> template <>
    void A<Y>::B<double>::mf2() { } // 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:

(19.1) — the explicit specialization of a function template;
(19.2) — the explicit specialization of a member function template;
(19.3) — 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]
14.8 Function template specializations

A function instantiated from a function template is called a function template specialization; so is an explicit specialization of a function template. Template arguments can be explicitly specified when naming the function template specialization, deduced from the context (e.g., deduced from the function arguments in a call to the function template specialization, see 14.8.2), or obtained from default template arguments.

Each function template specialization instantiated from a template has its own copy of any static variable.

Example:

```c
template<class T> void f(T* p) {
    static T s;
};

void g(int a, char* b) {
    f(&a); // calls f<int>(int*)
    f(&b); // calls f<char*>(char**)  
}
```

Here `f<int>(int*)` has a static variable `s` of type `int` and `f<char*>(char**)` has a static variable `s` of type `char*`. — end example 

14.8.1 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:

```c
template<class T> void sort(Array<T>& v);
void f(Array<dcomplex>& cv, Array<int>& ci) {
    sort<dcomplex>(cv); // sort(Array<dcomplex>&)
    sort<int>(ci);     // sort(Array<int>&)  
}
```

and

```c
template<class U, class V> U convert(V v);
void g(double d) {
    int i = convert<int,double>(d);    // int convert(double)
    char c = convert<char,double>(d);  // char convert(double)  

    — end example 
```

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 (14.8.2) or obtained from default template-arguments may be omitted from the list of explicit template-arguments. A trailing template parameter pack (14.5.3) 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<int,void>(f<int, bool>); // Y for outer f deduced to be int (*)(bool)
    f<int,void>(); // 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<int,void>(g<int, bool>); // Y for outer f is deduced to be int (*)(bool), Z is deduced to an empty sequence
}
```
—end example]

[Note: An empty template argument list can be used to indicate that a given use refers to a specialization of a function template even when a non-template function (8.3.5) is visible that would otherwise be used. For example:

```cpp
template <class T> int f(T); // #1
int f(int); // #2
int k = f(1); // uses #2
int l = f<>(1); // uses #1
```
—end note]

Template arguments that are present shall be specified in the declaration order of their corresponding template-parameters. The template argument list shall not specify more template-argument than there are corresponding template-parameters unless one of the template-parameters is a template parameter pack. [Example:

```cpp
template<class X, class Y, class Z> X f(Y,Z);
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]

Implicit conversions (Clause 4) 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 —

[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]

[Note: For simple function names, argument dependent lookup (3.4.2) applies even when the function name is not visible within the scope of the call. This is because the call still has the syntactic form of a function call (3.4.1). But when a function template with explicit template arguments is used, the call does not have the correct syntactic form unless there is a function template with that name visible at the point of the call. If no such name is visible, the call is not syntactically well-formed and argument-dependent lookup does not apply. If some such name is visible, argument dependent lookup applies and additional function templates may be found in other namespaces. [Example:

namespace A {
    struct B { }
    template<int X> void f(B);
}
namespace C {
    template<class T> void f(T t);
}
void g(A::B b) {
    f<3>(b);  // ill-formed: not a function call
    A::f<3>(b);  // well-formed
    C::f<3>(b);  // ill-formed; argument dependent lookup
    // applies only to unqualified names
    using C::f;
    f<3>(b);  // well-formed because C::f is visible; then A::f is found by argument dependent lookup
}
— end example] — end note]

Template argument deduction can extend the sequence of template arguments corresponding to a template parameter pack, even when the sequence contains explicitly specified template arguments. [Example:

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]
14.8.2 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:

```c
void f(Array<dcomplex>& cv, Array<int>& ci) {
    sort(cv);            // calls sort(Array<dcomplex>&)
    sort(ci);            // calls sort(Array<int>&)
}
```

and

```c
void g(double d) {
    int i = convert<int>(d);    // calls convert<int,double>(double)
    int c = convert<char>(d);   // calls convert<char,double>(double)
}
```

— end example]

2 When an explicit template argument list is specified, the template arguments must be compatible with the template parameter list and must result in a valid function type as described below; otherwise type deduction fails. Specifically, the following steps are performed when evaluating an explicitly specified template argument list with respect to a given function template:

1. The specified template arguments must match the template parameters in kind (i.e., type, non-type, template). There must not be more arguments than there are parameters unless at least one parameter is a template parameter pack, and there shall be an argument for each non-pack parameter. Otherwise, type deduction fails.

2. Non-type arguments must match the types of the corresponding non-type template parameters, or must be convertible to the types of the corresponding non-type parameters as specified in 14.3.2, otherwise type deduction fails.

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 8.3.5 are performed. [Example: A parameter type of "void ()(const int, int[5])" becomes "void(*)(int,int*)".

— end example] [Note: A top-level qualifier in a function parameter declaration does not affect the function type but still affects the type of the function parameter variable within the function. — end note] [Example:

```c
template <class T> void f(T t);
template <class X> void g(const X x);
template <class Z> void h(Z, Z*);

int main() {
    // #1: function type is f(int), t is non const
    f<int>(1);

    // #2: function type is f(int), t is const
    f<const int>(1);

    // #3: function type is g(int), x is const
    g<int>(1);

    // #4: function type is g(int), x is const
    g<const int>(1);
```

§ 14.8.2
// #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:

```cpp
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.

6 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.

7 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 `exception-specification` is instantiated, at which point a program is ill-formed if the substitution results in an invalid type or expression. — end note]

[Example:

```cpp
template <class T> struct A { using X = typename T::X; };  
template <class T> typename T::X f(typename A<T>::X);
template <class T> void f(...) { }
template <class T> 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
```
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 evaluation of the substituted types and expressions can result in side effects such as the instantiation of class template specializations and/or function template specializations, the generation of implicitly-defined functions, etc. Such side effects are not in the “immediate context” and can result in the program being ill-formed. —end note]

[Example:

```cpp
g<int>(0); // error, substituting parameter type instantiates A<int>
}
— end example]
```

8

§ 14.8.2
template <int I> struct X { };
template <template <class T> class> struct Z { };

template <class T> void f(typename T::Y*){}
template <class T> void g(X<T::N>*){}
template <class T> void h(Z<T::template TT>*){}

struct A {}
struct B { int Y; }
struct C {
    typedef int N;
};
struct D {
    typedef int TT;
};

int main() {
    // Deduction fails in each of these cases:
    f<A>(0);   // A does not contain a member Y
    f<B>(0);   // The Y member of B is not a type
    g<C>(0);   // The N member of C is not a non-type
    h<D>(0);   // The TT member of D is not a template
}

— end example ]

(8.5) — Attempting to create a pointer to reference type.
(8.6) — Attempting to create a reference to void.
(8.7) — Attempting to create “pointer to member of T” when T is not a class type. [ Example:
    template <class T> int f(int T::*);
    int i = f<int>(0);

    — end example ]
(8.8) — Attempting to give an invalid type to a non-type template parameter. [ Example:
    template <class T, T> struct S {}
    template <class T> int f(S<T, T>()*)
    struct X {}
    int i0 = f<X>(0);

    — end example ]
(8.9) — Attempting to perform an invalid conversion in either a template argument expression, or an expression used in the function declaration. [ Example:
    template <class T, T*> int f(int);
    int i2 = f<int,1>(0);      // can't conv 1 to int*

    — end example ]
(8.10) — Attempting to create a function type in which a parameter has a type of void, or in which the return type is a function type or array type.
(8.11) — Attempting to create a function type in which a parameter type or the return type is an abstract class type (10.4).
Example: In the following example, assuming a signed char cannot represent the value 1000, a narrowing conversion (8.5.4) would be required to convert the template-argument of type int to signed char, therefore substitution fails for the second template (14.3.2).

```cpp
template <int> int f(int);
template <signed char> int f(int);
int i1 = f<1000>(0);        // OK
int i2 = f<1>(0);           // ambiguous; not narrowing
```

— end example —

14.8.2.1 Deducing template arguments from a function call

Template argument deduction is done by comparing each function template parameter type (call it $P$) with the type of the corresponding argument of the call (call it $A$) as described below. If $P$ is a dependent type, removing references and cv-qualifiers from $P$ gives $\text{std::initializer_list}\langle P' \rangle$ or $P'[N]$ for some $P'$ and $N$ and the argument is a non-empty initializer list (8.5.4), then deduction is performed instead for each element of the initializer list, taking $P'$ 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 (14.8.2.5).

```cpp
Example:

```cpp
template<class T> void f(std::initializer_list<T>);
f({1,2,3});          // error: deduction fails, no conversion from int to Aggr
k({1,"asdf"});      // error: no argument deduced for T

struct Aggr { int i; int j; };
```

--- end example ---
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:

(2.1) — If A is an array type, the pointer type produced by the array-to-pointer standard conversion (4.2) is used in place of A for type deduction; otherwise,

(2.2) — If A is a function type, the pointer type produced by the function-to-pointer standard conversion (4.3) is used in place of A for type deduction; otherwise,

(2.3) — If A is a cv-qualified type, the top level cv-qualifiers of A's type are ignored for type deduction.

3 If P is a cv-qualified type, the top level cv-qualifiers of P's type are ignored for type deduction. If P is a reference type, the type referred to by P is used for type deduction. A forwarding reference is an rvalue reference to a cv-unqualified template parameter. 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:

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

—end example]

4 In general, the deduction process attempts to find template argument values that will make the deduced A identical to A (after the type A is transformed as described above). However, there are three cases that allow a difference:

(4.1) — If the original P is a reference type, the deduced A (i.e., the type referred to by the reference) can be more cv-qualified than the transformed A.

(4.2) — The transformed A can be another pointer or pointer to member type that can be converted to the deduced A via a qualification conversion (4.4).

(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.

[Note: as specified in 14.8.1, implicit conversions will be performed on a function argument to convert it to the type of the corresponding function parameter if the parameter contains no template-parameters that participate in template argument deduction. Such conversions are also allowed, in addition to the ones described in the preceding list. —end note]

5 These alternatives are considered only if type deduction would otherwise fail. If they yield more than one possible deduced A, the type deduction fails. [Note: If a template-parameter is not used in any of the

§ 14.8.2.1
function parameters of a function template, or is used only in a non-deduced context, its corresponding
_template-argument cannot be deduced from a function call and the _template-argument must be explicitly
specified. — end note

6 When P is a function type, pointer to function 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.

[ Example:

// Only one function of an overload set matches the call so the function
// parameter is a deduced context.
template <class T> int f(T (*)(T));
int g(int);
int g(char);
int i = f(g); // calls f(int (*)(int))

— end example ]

[ Example:

// Ambiguous deduction causes the second function parameter to be a
// non-deduced context.
template <class T> int f(T, T (*)(T));
int g(int);
char g(char);
int i = f(1, g); // calls f(int, int (*)(int))

— end example ]

[ 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 (*)(T));
char g(char);
template <class T> T g(T);
int i = f(1, g); // calls f(int, int (*)(int))

— end example ]

14.8.2.2 Deducing template arguments taking the address of a function template
[temp.deduct.funcaddr]

1 Template arguments can be deduced from the type specified when taking the address of an overloaded
function (13.4). 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 14.8.2.5.

2 A placeholder type (7.1.6.4) 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.
14.8.2.3 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 8.5, 13.3.1.5, and 13.3.1.6 for the determination of that type) as described in 14.8.2.5.

1 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 section.

2 If \( A \) is not a reference type:

   - If \( P \) is an array type, the pointer type produced by the array-to-pointer standard conversion (4.2) is used in place of \( P \) for type deduction; otherwise,
   
   - If \( P \) is a function type, the pointer type produced by the function-to-pointer standard conversion (4.3) is used in place of \( P \) for type deduction; otherwise,

   - If \( P \) is a cv-qualified type, the top level cv-qualifiers of \( P \)’s type are ignored for type deduction.

3 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.

4 In general, the deduction process attempts to find template argument values that will make the deduced \( A \) identical to \( A \). However, there are two cases that allow a difference:

   - If the original \( A \) is a reference type, \( A \) can be more cv-qualified than the deduced \( A \) (i.e., the type referred to by the reference)

5 The deduced \( A \) can be another pointer or pointer to member type that can be converted to \( A \) via a qualification conversion.

6 These alternatives are considered only if type deduction would otherwise fail. If they yield more than one possible deduced \( A \), the type deduction fails.

7 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

   \[ cv_{1,0} \text{ “pointer to . . .” } cv_{1,n-1} \text{ “pointer to” } cv_{1,n} T1 \]

   and \( P \) is a type

   \[ cv_{2,0} \text{ “pointer to . . .” } cv_{2,n-1} \text{ “pointer to” } cv_{2,n} T2 \]

   The cv-unqualified \( T1 \) and \( T2 \) are used as the types of \( A \) and \( P \) respectively for type deduction. [Example:

   ```
   struct A {
   template <class T> operator T***();
   };
   A a;
   const int * const * const * p1 = a;  // T is deduced as int, not const int
   ```

   — end example]
14.8.2.4 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 14.5.6.2. — 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:

(3.1) — In the context of a function call, the types used are those function parameter types for which the function call has arguments.\(^\text{142}\)

(3.2) — In the context of a call to a conversion function, the return types of the conversion function templates are used.

(3.3) — In other contexts (14.5.6.2) 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 \(A\).

Before the partial ordering is done, certain transformations are performed on the types used for partial ordering:

(5.1) — If \(P\) is a reference type, \(P\) is replaced by the type referred to.

(5.2) — If \(A\) is a reference type, \(A\) is replaced by the type referred to.

If both \(P\) and \(A\) were reference types (before being replaced with the type referred to above), determine which of the two types (if any) is more cv-qualified than the other; otherwise the types are considered to be equally cv-qualified for partial ordering purposes. The result of this determination will be used below.

Remove any top-level cv-qualifiers:

(7.1) — If \(P\) is a cv-qualified type, \(P\) is replaced by the cv-unqualified version of \(P\).

(7.2) — If \(A\) is a cv-qualified type, \(A\) is replaced by the cv-unqualified version of \(A\).

If \(A\) was transformed from a function parameter pack and \(P\) is not a parameter pack, type deduction fails. Otherwise, using the resulting types \(P\) and \(A\), the deduction is then done as described in 14.8.2.5. 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. 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:

\[
\begin{align*}
\text{template<class... Args>} & \quad \text{void f(Args... args);} \quad // \#1 \\
\text{template<class T1, class... Args>} & \quad \text{void f(T1 a1, Args... args);} \quad // \#2 \\
\text{template<class T1, class T2>} & \quad \text{void f(T1 a1, T2 a2);} \quad // \#3 \\
\end{align*}
\]

\(fO\); \quad // calls \#1
\]

\[142\) Default arguments are not considered to be arguments in this context; they only become arguments after a function has been selected.
f(1, 2, 3);  // calls #2
f(1, 2);    // calls #3; non-variadic template #3 is more
            // specialized than the variadic templates #1 and #2

— end example

9 If, for a given type, deduction succeeds in both directions (i.e., the types are identical after the transfor-
mations above) and both P and A were reference types (before being replaced with the type referred to
above):

(9.1) — if the type from the argument template was an lvalue reference and the type from the parameter
template was not, the parameter type is not considered to be at least as specialized as the argument
type; otherwise,

(9.2) — if the type from the argument template is more cv-qualified than the type from the parameter template
(as described above), the parameter type is not considered to be at least as specialized as the argument
type.

10 Function template \( F \) is at least as specialized as function template \( G \) if, for each pair of types used to
determine the ordering, the type from \( F \) is at least as specialized as the type from \( G \). \( F \) is more specialized
than \( G \) if \( F \) is at least as specialized as \( G \) and \( G \) is not at least as specialized as \( F \).

11 In most cases, all template parameters must have values in order for deduction to succeed, but for partial
ordering purposes a template parameter may remain without a value provided it is not used in the types
being used for partial ordering. [ Note: A template parameter used in a non-deduced context is considered
used. — end note ] [ Example:

\[
\begin{align*}
\text{template } \langle \text{class } T \rangle & \ T \ f(\text{int}); \quad \text{ // #1} \\
\text{template } \langle \text{class } T, \text{ class } U \rangle & \ T \ f(U); \quad \text{ // #2} \\
\text{void } g() \ \{ \\
\quad \ f<\text{int}>(1); \quad \text{ // calls #1} \\
\} 
\end{align*}
\]

— end example ]

12 [ 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:

\[
\begin{align*}
\text{template<class ...> struct Tuple } & \ { } ; \\
\text{template<class ... Types> void } g(\text{Tuple<Types ...>}); \quad \text{ // #1} \\
\text{template<class T1, class ... Types> void } g(\text{Tuple<T1, Types ...>}); \quad \text{ // #2} \\
\text{template<class T1, class ... Types> void } g(\text{Tuple<T1, Types& ...>}); \quad \text{ // #3} \\
\text{g(Tuple<>());} \quad \text{ // calls #1} \\
\text{g(Tuple<int, float>();} \quad \text{ // calls #2} \\
\text{g(Tuple<int, float&>();} \quad \text{ // calls #3} \\
\text{g(Tuple<int>();} \quad \text{ // calls #3} 
\end{align*}
\]

— end example ]

14.8.2.5 Deducing template arguments from a type [temp.deduct.type]

Template arguments can be deduced in several different contexts, but in each case a type that is specified
in terms of template parameters (call it P) is compared with an actual type (call it A), and an attempt
is made to find template argument values (a type for a type parameter, a value for a non-type parameter, or
a template for a template parameter) that will make P, after substitution of the deduced values (call it the
deduced 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.

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., \( \mathcal{A}<\text{int}\rangle \)) 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 the value so determined must be 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.

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 (13.4), and one or more of the following apply:
   1. more than one function matches the function parameter type (resulting in an ambiguous deduction), or
   2. no function matches the function parameter type, or
   3. the set of functions supplied as an argument contains one or more function templates.
6. A function parameter for which the associated argument is an initializer list (8.5.4) but the parameter does not have a type for which deduction from an initializer list is specified (14.8.2.1). [Example:

   ```cpp
   template<class T> void g(T);
   g({1,2,3}); // error: no argument deduced for T
   ```

   — end example]
7. A function parameter pack that does not occur at the end of the parameter-declaration-list.
When a type name is specified in a way that includes a non-deduced context, all of the types that comprise that type name are also non-deduced. However, a compound type can include both deduced and non-deduced types. [Example: If a type is specified as \( A<T>::B<T2> \), both \( T \) and \( T2 \) are non-deduced. Likewise, if a type is specified as \( A<I+J>::X<T> \), \( I \), \( J \), and \( T \) are non-deduced. If a type is specified as \( \text{void } f(\text{typename } A<T>::B, A<T>) \), the \( T \) in \( A<T>::B \) is non-deduced but the \( T \) in \( A<T> \) is deduced. — end example]

Here is an example in which different parameter/argument pairs produce inconsistent template argument deductions:

```cpp
template<class T> void f(T x, T y) { /* ... */ }
struct A { /* ... */ };
struct B : A { /* ... */ };
void g(A a, B b) {
    f(a,b);  // error: \( T \) could be \( A \) or \( B \)
    f(b,a);  // error: \( T \) could be \( A \) or \( B \)
    f(a,a);  // OK: \( T \) is \( A \)
    f(b,b);  // OK: \( T \) is \( B \)
}
```

Here is an example where two template arguments are deduced from a single function parameter/argument pair. This can lead to conflicts that cause type deduction to fail:

```cpp
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> void f(B<T>&){}
void t() {
    D<int> d;
    D2 d2;
    f(d);  // calls \( f(B<int>&) \)
    f(d2);  // calls \( f(B<int>&) \)
}
```
A template type argument $T$, a template template argument $TT$ or a template non-type argument $i$ can be deduced if $P$ and $A$ have one of the following forms:

$$
\begin{align*}
T & \text{ cu-list } T \\
T* & \\
T& \\
T&& \\
T[\text{integer-constant}] & \\
template-name<T> & \text{ (where } template-name \text{ refers to a class template)} \\
type(T) & \\
T() & \\
T(T) & \\
T type::* & \\
type T type::* & \\
T.T::* & \\
type (T::*)(T) & \\
type T (T::*)(T) & \\
type[i] & \\
template-name<i> & \text{ (where } template-name \text{ refers to a class template)} \\
TT<T> & \\
TT<i> & \\
TT<> & \\
\end{align*}
$$

where (T) represents a parameter-type-list where at least one parameter type contains a $T$, and () represents a parameter-type-list where no parameter type contains a $T$. Similarly, $<$T$>$ represents template argument lists where at least one argument contains a $T$, $<$i$>$ represents template argument lists where at least one argument contains an $i$ and $<$> represents template argument lists where no argument contains a $T$ or an $i$.

If $P$ has a form that contains $<$T$>$ or $<$i$>$, then each argument $P_i$ of the respective template argument list $P$ is compared with the corresponding argument $A_i$ of the corresponding template argument list of $A$. If the template argument list of $P$ contains a pack expansion that is not the last template argument, the entire template argument list is a non-deduced context. If $P_i$ is a pack expansion, then the pattern of $P_i$ is compared with each remaining argument in the template argument list of $A$. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by $P_i$. During partial ordering (14.8.2.4), if $A_i$ was originally a pack expansion:

1. if $P$ does not contain a template argument corresponding to $A_i$ then $A_i$ is ignored;
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
int, const int& s; // both #2 and #3 match; #3 is more specialized
```

```cpp
template<class T, class... U> struct A { }; // #1
template<class T1, class T2, class... U> struct A<T1, T2*, U...> { }; // #2
```

§ 14.8.2.5
Similarly, if \( P \) has a form that contains \((T)\), then each parameter type \( P_i \) of the respective parameter-type-list 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 (14.8.2.2) or when deducing template arguments from a function declaration (14.8.2.6) 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 (14.8.2.1) 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 \( X\) rather than \( T\&\), 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&), i.e., #2
}
```

— end example]

If the parameter-declaration corresponding to \( P_i \) is a function parameter pack, then the type of its declarator-id is compared with each remaining parameter type in the parameter-type-list of \( A \). Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack. During partial ordering (14.8.2.4), if \( A_i \) was originally a function parameter pack:

10.1 — if \( P \) does not contain a function parameter type corresponding to \( A_i \) then \( A_i \) is ignored;

10.2 — otherwise, if \( P_i \) is not a function parameter pack, template argument deduction fails.

[Example:

```cpp
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]

12 Template arguments cannot be deduced from function arguments involving constructs other than the ones specified above.

13 A template type argument cannot be deduced from the type of a non-type template-argument.

[Example:

\[§ 14.8.2.5\]
template<class T, T i> void f(double a[10][i]);
int v[10][20];
f(v); // error: argument for template-parameter T cannot be deduced

— end example]

[Note: Except for reference and pointer types, a major array bound is not part of a function parameter type and cannot be deduced from an argument:

    template<int i> void f1(int a[10][i]);
    template<int i> void f2(int a[i][20]);
    template<int i> void f3(int (&a)[i][20]);

    void g() {
      int v[10][20];
      f1(v); // OK: i deduced to be 20
      f2(v); // OK
      f2<10>(v); // error: cannot deduce template-argument i
      f3(v); // OK: i deduced to be 10
    }

].

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:

    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,

    template<int i, typename T> 
    T deduce(typename A<T>::X x, 
             T t, 
             typename B<i>::Y y); 
    // T is not deduced here
    // but T is deduced here
    // i is not deduced here
    A<int> a;
    B<77> b;

    int  x = deduce<77>(a.xm, 62, b.ym);  
    // T is deduced to be int, a.xm must be convertible to
    // A<int>::X
    // i is explicitly specified to be 77, b.ym must be convertible
    // to B<77>::Y

] — end note]

If P has a form that contains <i>, and if the type of the corresponding value of A differs from the type of i, deduction fails. If P has a form that contains [i], and if the type of i is not an integral type, deduction
fails.143  [Example:]
    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 
    }

template<const short cs> class B { }; 
    template<short s> void g(B<s>); 
    void k2() {
        B<1> b; 
        g(b);  // OK: cv-qualifiers are ignored on template parameter types 
    }

— end example]

18 A template-argument can be deduced from a function, pointer to function, or pointer to member function type. [Example:]
    template<class T> void f(void(*)(T,int)); 
    template<class T> void foo(T,int); 
    void g(int,int); 
    void g(char,int); 
    void h(int,int,int); 
    void h(char,int); 

    void m() {
        f(&g);  // error: ambiguous 
        f(&h);  // OK: void h(char,int) is a unique match 
        f(&foo);  // error: type deduction fails because foo is a template 
    }

— end example]

19 A template type-parameter cannot be deduced from the type of a function default argument. [Example:]
    template <class T> void f(T = 5, T = 7); 
    void g() {
        f(1);  // OK: call f<int>(1,7) 
        f();    // error: cannot deduce T 
        f<int>();  // OK: call f<int>(5,7) 
    }

— end example]

20 The template-argument corresponding to a template template-parameter is deduced from the type of the template-argument of a class template specialization used in the argument list of a function call. [Example:
    template <template <class T> class X> struct A { }; 
    template <template <class T> class X> void f(A<X>) { }
    template<class T> struct B { }; 

143) 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 non-zero.
A<B> ab;
f(ab); // calls f(A<B>)

—end example

Note: Template argument deduction involving parameter packs (14.5.3) can deduce zero or more arguments for each parameter pack. —end note

Example:

template<class> struct X { }
template<class R, class ... ArgTypes> struct X<R(int, ArgTypes ...)> { }
template<class T, class ... Types> struct Y { }

template<class ... Types> int f(void (*)(Types ...));
void g(int, float);

X<int> x1; // uses primary template
X<int(int, float, double)> x2; // uses partial specialization; ArgTypes contains float, double
X<int(float, int)> x3; // uses primary template
Y<> 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

14.8.2.6 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 (14.7.2), explicit specializations (14.7.3), and certain friend declarations (14.5.4). This is also done to determine whether a deallocation function template specialization matches a placement operator new (3.7.4.2, 5.3.4). 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 5.3.4. The deduction is done as described in 14.8.2.5.

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 (14.5.6.2), deduction fails and, in the declaration cases, the program is ill-formed.

14.8.3 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 (14.8.2) and checking of any explicit template arguments (14.3) 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, 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 13.3.3.144

[Example:

```cpp
template<class T> T max(T a, T b) { return a>b?a:b; }
```

void f(int a, int b, char c, char d) {
    int m1 = max(a,b); // max(int, int)
    char m2 = max(c,d); // max(char, char)
    int m3 = max(a,c); // error: cannot generate max(int, char)
}

2 Adding the non-template function

```cpp
int max(int, int);
```

to the example above would resolve the third call, by providing a function that could be called for `max(a, c)` after using the standard conversion of `char` to `int` for `c`.

3 Here is an example involving conversions on a function argument involved in template-argument deduction:

```cpp
template<class T> struct B { /* ... */ };
template<class T> struct D : public B<T> { /* ... */ };  
template<class T> void f(B<T>&);
void g(B<int>& bi, D<int>& di) {
    f(bi);  // f(bi)
    f(di);  // f((B<int>&)di)
}
```

4 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*>}
```

144) 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 13.3.3 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.

§ 14.8.3
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 \( f<\text{const char}*> \), either implicitly or explicitly generated, is present in some translation unit. — end example}
15 Exception handling [except]

1 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 (15.3).

2 A try-block is a statement (Clause 6). [Note: Within this Clause “try block” is taken to mean both try-block and function-try-block. — end note]

3 A goto or switch statement shall not be used to transfer control into a try block or into a handler.

```
void f() {
  goto 11; // Ill-formed
  goto 12; // Ill-formed
  try {
    goto 11; // OK
    goto 12; // Ill-formed
    11: ;
  } catch (...) {
    12: ;
    goto 11; // Ill-formed
    goto 12; // OK
  }
}
```

—end example] A goto, break, return, or continue statement can be used to transfer control out of a try block or handler. When this happens, each variable declared in the try block will be destroyed in the context that directly contains its declaration. [Example:

```
lab: try {
  T1 t1;
  try {
    T2 t2;
    if (condition)
      goto lab;
```

Exception handling 414
Here, executing `goto lab;` will destroy first `t2`, then `t1`, assuming the condition does not declare a variable. Any exception raised while destroying `t2` will result in executing `handler 2`; any exception raised 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]

15.1 Throwing an exception [except.throw]

Throwing an exception transfers control to a handler. [Note: An exception can be thrown from one of the following contexts: throw-expressions (5.17), allocation functions (3.7.4.1), dynamic_cast (5.2.7), typeid (5.2.8), new-expressions (5.3.4), and standard library functions (17.5.1.4). — 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
	ry {
    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 (15.3); “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 (8.5, 12.8) a temporary object, called the exception object. The temporary is an lvalue and is used to initialize the variable declared in the matching handler (15.3). If the type of the exception object would be an incomplete type or a pointer to an incomplete type other than (possibly cv-qualified) void the program is ill-formed.

4 The memory for the exception object is allocated in an unspecified way, except as noted in 3.7.4.1. If a handler exits by rethrowing, control is passed to another handler for the same exception. The exception object is destroyed after either the last remaining active handler for the exception exits by any means other than rethrowing, or the last object of type std::exception_ptr (18.8.5) that refers to the exception object is destroyed, whichever is later. In the former case, the destruction occurs when the handler exits, immediately after the destruction of the object declared in the exception-declaration in the handler, if any. In the latter case, the destruction occurs before the destructor of std::exception_ptr returns. 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 18.8.5 and 30.6. —end note]

5 When the thrown object is a class object, the constructor selected for the copy-initialization and the destructor shall be accessible, even if the copy/move operation is elided (12.8).

6 An exception is considered caught when a handler for that exception becomes active (15.3). [Note: An exception can have active handlers and still be considered uncaught if it is rethrown. —end note]

7 If the exception handling mechanism, after completing the initialization of the exception object but before the activation of a handler for the exception, calls a function that exits via an exception, std::terminate is called (15.5.1). [Example:]

struct C {
    C() { }  
    C(const C& k) { 
        if (std::uncaught_exceptions()) { 
            throw 0;  // throw during copy to handler’s exception-declaration object (15.3)
        }
    }
};

int main() {
    try {
        throw C();  // calls std::terminate() if construction of the handler’s
        // exception-declaration object is not elided (12.8)
    } catch(C) { }
15.2 Constructors and destructors

As control passes from the point where an exception is thrown to a handler, destructors are invoked by a process, specified in this section, called stack unwinding. If a destructor directly invoked by stack unwinding exits with an exception, `std::terminate` is called (15.5.1). [Note: Consequently, destructors should generally catch exceptions and not let them propagate out of the destructor. — end note]

The destructor is invoked for each automatic object of class type constructed since the try block was entered. The automatic objects are destroyed in the reverse order of the completion of their construction.

For an object of class type of any storage duration whose initialization or destruction is terminated by an exception, the destructor is invoked for each of the object’s fully constructed subobjects, that is, for each subobject for which the principal constructor (12.6.2) has completed execution and the 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.

Similarly, if the non-delegating constructor for an object has completed execution and a delegating constructor for that object exits with 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 (5.3.4), the matching deallocation function (3.7.4.2, 5.3.4, 12.5), if any, is called to free the storage occupied by the object. — end note]

15.3 Handling an exception

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*`.

A handler of type “array of T” or “function returning T” is adjusted to be of type “pointer to T” or “pointer to function returning T”, respectively.

A handler is a match for an exception object of type E if

1. The handler is of type `cv T` or `cv T&` and E and T are the same type (ignoring the top-level cv-qualifiers), or
2. the handler is of type `cv T` or `cv T&` and T is an unambiguous public base class of E, or
3. the handler is of type `cv T` or `const T&` where T is a pointer type and E is a pointer type that can be converted to T by either or both of
   3.1. a standard pointer conversion (4.10) not involving conversions to pointers to private or protected or ambiguous classes
   3.2. a qualification conversion, or
   3.3. the handler is of type `cv T` or `const T&` where T is a pointer or pointer to member type and E is `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. — end note]
[Example:

```cpp
class Matherr { /* ... */ virtual void vf(); };  
class Overflow: public Matherr { /* ... */ };  
class Underflow: public Matherr { /* ... */ };  
class Zerodivide: public Matherr { /* ... */ };  

void f() {
  try {
    g();
  } catch (Overflow oo) {
    // ...
  } catch (Matherr mm) {
    // ...
  }
}
```

Here, the `Overflow` handler will catch exceptions of type `Overflow` and the `Matherr` handler will catch exceptions of type `Matherr` and of all types publicly derived from `Matherr` including exceptions of type `Underflow` and `Zerodivide.` — end example]

4 The handlers for a try block are tried in order of appearance. That makes it possible to write handlers that can never be executed, for example by placing a handler for a derived class after a handler for a corresponding base class.

5 A ... in a handler’s `exception-declaration` functions similarly to ... in a function parameter declaration; it specifies a match for any exception. If present, a ... handler shall be the last handler for its try block.

6 If no match is found among the handlers for a try block, the search for a matching handler continues in a dynamically surrounding try block of the same thread.

7 A handler is considered active when initialization is complete for the parameter (if any) of the catch clause. [Note: The stack will have been unwound at that point. — end note] Also, an implicit handler is considered active when `std::terminate()` or `std::unexpected()` is entered due to a throw. A handler is no longer considered active when the catch clause exits or when `std::unexpected()` exits after being entered due to a throw.

8 The exception with the most recently activated handler that is still active is called the `currently handled exception`.

9 If no matching handler is found, the function `std::terminate()` is called; whether or not the stack is unwound before this call to `std::terminate()` is implementation-defined (15.5.1).

10 Referring to any non-static member or base class of an object in the handler for a function-try-block of a constructor or destructor for that object results in undefined behavior.

11 The scope and lifetime of the parameters of a function or constructor extend into the handlers of a function-try-block.

12 Exceptions thrown in destructors of objects with static storage duration or in constructors of namespace-scope objects with static storage duration are not caught by a function-try-block on `main()`. Exceptions thrown in destructors of objects with thread storage duration or in constructors of namespace-scope objects with thread storage duration are not caught by a function-try-block on the initial function of the thread.

13 If a return statement appears in a handler of the function-try-block of a constructor, the program is ill-formed.

14 The currently handled exception is rethrown if control reaches the end of a handler of the function-try-block of a constructor or destructor. Otherwise, a function returns when control reaches the end of a handler for the function-try-block (6.6.3). Flowing off the end of a function-try-block is equivalent to a `return` with no value; this results in undefined behavior in a value-returning function (6.6.3).
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 (8.5) from the corresponding base class subobject of the exception object;

15.2 otherwise, the variable is copy-initialized (8.5) 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.

15.4 Exception specifications

The exception specification of a function is a (possibly empty) set of types, indicating that the function might exit via an exception that matches a handler of one of the types in the set; the (conceptual) set of all types is used to denote that the function might exit via an exception of arbitrary type. If the set is empty, the function is said to have a non-throwing exception specification. The exception specification is either defined explicitly by using an exception-specification as a suffix of a function declaration’s declarator (8.3.5) or implicitly.

```
exception-specification:
  dynamic-exception-specification
  noexcept-specification

dynamic-exception-specification:
  throw ( type-id-list_opt )

type-id-list:
  type-id ...opt
  type-id-list, type-id ...opt

noexcept-specification:
  noexcept ( constant-expression )
  noexcept
```

In a noexcept-specification, the constant-expression, if supplied, shall be a constant expression (5.20) that is contextually converted to bool (Clause 4). A (token that follows noexcept is part of the noexcept-specification and does not commence an initializer (8.5).

An exception-specification shall appear only on a function declarator for a function type, pointer to function type, reference to function type, or pointer to member function type that is the top-level type of a declaration or definition, or on such a type appearing as a parameter or return type in a function declarator. An exception-specification shall not appear in a typedef declaration or alias-declaration. [Example:

```
void f() throw(int); // OK
void (*fp)() throw (int); // OK
void g(void pfa() throw(int)); // OK
typedef int (*pf)() throw(int); // ill-formed
```

-- end example]

A type denoted in a dynamic-exception-specification shall not denote an incomplete type or an rvalue reference type. A type denoted in a dynamic-exception-specification shall not denote a pointer or reference to an incomplete type, other than “pointer to cv void”. A type cv \( T \), “array of \( T \)”, or “function returning \( T \)” denoted in a dynamic-exception-specification is adjusted to type \( T \), “pointer to \( T \)”, or “pointer to function returning \( T \)”, respectively. A dynamic-exception-specification denotes an exception specification that is the set of adjusted types specified thereby.
The *exception-specification* `noexcept` or `noexcept(constant-expression)`, where the `constant-expression` yields `true`, denotes an exception specification that is the empty set. The *exception-specification* `noexcept(constant-expression)`, where the `constant-expression` yields `false`, or the absence of an *exception-specification* in a function declarator other than that for a destructor (12.4) or a deallocation function (3.7.4.2) denotes an exception specification that is the set of all types.

Two *exception-specifications* are *compatible* if the sets of types they denote are the same.

If any declaration of a function has an *exception-specification* that is not a `noexcept-specification` allowing all exceptions, all declarations, including the definition and any explicit specialization, of that function shall have a compatible *exception-specification*. If any declaration of a pointer to function, reference to function, or pointer to member function has an *exception-specification*, all occurrences of that declaration shall have a compatible *exception-specification*. If a declaration of a function has an implicit exception specification, other declarations of the function shall not specify an *exception-specification*. In an explicit instantiation an *exception-specification* may be specified, but is not required. If an *exception-specification* is specified in an explicit instantiation directive, it shall be compatible with the *exception-specifications* of other declarations of that function. A diagnostic is required only if the *exception-specifications* are not compatible within a single translation unit.

If a virtual function has an exception specification, all declarations, including the definition, of any function that overrides that virtual function in any derived class shall only allow exceptions that are allowed by the exception specification of the base class virtual function, unless the overriding function is defined as deleted.

```c
struct B {
    virtual void f() throw (int, double);
    virtual void g();
};

struct D: B {
    void f(); // ill-formed
    void g() throw (int); // OK
};
```

The declaration of `D::f` is ill-formed because it allows all exceptions, whereas `B::f` allows only `int` and `double`. —end example] A similar restriction applies to assignment to and initialization of pointers to functions, pointers to member functions, and references to functions: the target entity shall allow at least the exceptions allowed by the source value in the assignment or initialization.

```c
class A { /* ... */ }; // no exception specification
void (*pf1)();
void (*pf2)() throw(A);

void f() {
    pf1 = pf2; // OK: pf1 is less restrictive
    pf2 = pf1; // error: pf2 is more restrictive
}
```

—end example]

In such an assignment or initialization, *exception-specifications* on return types and parameter types shall be compatible. In other assignments or initializations, *exception-specifications* shall be compatible.

An *exception-specification* can include the same type more than once and can include classes that are related by inheritance, even though doing so is redundant. [Note: An *exception-specification* can also include the class `std::bad_exception` (18.8.2). —end note]
A function is said to *allow an exception* of type \( E \) if its exception specification contains a type \( T \) for which a handler of type \( T \) would be a match (15.3) for an exception of type \( E \). A function is said to *allow all exceptions* if its exception specification is the set of all types.

Whenever an exception of type \( E \) is thrown and the search for a handler (15.3) encounters the outermost block of a function with an exception specification that does not allow \( E \), then,

1. if the function definition has a *dynamic-exception-specification*, the function `std::unexpected()` is called (15.5.2),
2. otherwise, the function `std::terminate()` is called (15.5.1).

[Example:
```cpp
class X { }
class Y { }
class Z: public X { }
class W { }

void f() throw (X, Y) {
    int n = 0;
    if (n) throw X();    // OK
    if (n) throw Z();    // also OK
    throw W();           // will call std::unexpected()
}
```
— end example]

[Note: A function can have multiple declarations with different non-throwing *exception-specifications*; for this purpose, the one on the function definition is used. — end note]

An implementation shall not reject an expression merely because when executed it throws or might throw an exception that the containing function does not allow. [Example:
```cpp
extern void f() throw(X, Y);

void g() throw(X) {
    f();    // OK
}
```
the call to \( f \) is well-formed even though when called, \( f \) might throw exception \( Y \) that \( g \) does not allow. — end example]

[Note: An exception specification is not considered part of a function's type; see 8.3.5. — end note]

A *potential exception* of a given context is either a type that might be thrown as an exception or a pseudo-type, denoted by “any”, that represents the situation where an exception of an arbitrary type might be thrown. A subexpression \( e1 \) of an expression \( e \) is an immediate subexpression if there is no subexpression \( e2 \) of \( e \) such that \( e1 \) is a subexpression of \( e2 \).

The set of potential exceptions of a function, function pointer, or member function pointer \( f \) is defined as follows:

1. If the exception specification of \( f \) is the set of all types, the set consists of the pseudo-type “any”.
2. Otherwise, the set consists of every type in the exception specification of \( f \).

The set of potential exceptions of an expression \( e \) is empty if \( e \) is a core constant expression (5.20). Otherwise, it is the union of the sets of potential exceptions of the immediate subexpressions of \( e \), including default argument expressions used in a function call, combined with a set \( S \) defined by the form of \( e \), as follows:
If \( e \) is a function call (5.2.2):

- If its postfix-expression is a (possibly parenthesized) id-expression (5.1.1), class member access (5.2.5), or pointer-to-member operation (5.5) whose cast-expression is an id-expression, \( S \) is the set of potential exceptions of the entity selected by the contained id-expression (after overload resolution, if applicable).

- Otherwise, \( S \) contains the pseudo-type “any”.

If \( e \) implicitly invokes a function (such as an overloaded operator, an allocation function in a new-expression, or a destructor if \( e \) is a full-expression (1.9)), \( S \) is the set of potential exceptions of the function.

If \( e \) is a throw-expression (5.17), \( S \) consists of the type of the exception object that would be initialized by the operand, if present, or the pseudo-type “any” otherwise.

If \( e \) is a dynamic_cast expression that casts to a reference type and requires a run-time check (5.2.7), \( S \) consists of the type `std::bad_cast`.

If \( e \) is a typeid expression applied to a glvalue expression whose type is a polymorphic class type (5.2.8), \( S \) consists of the type `std::bad_typeid`.

If \( e \) is a new-expression with a non-constant expression in the noptr-new-declarator (5.3.4), \( S \) consists of the type `std::bad_array_new_length`.

[Example: Given the following declarations

```cpp
define void f() throw(int);
define void g();
define struct A { A(); };
define struct B { B() noexcept; };
define struct D() { D() throw (double); };
```

the set of potential exceptions for some sample expressions is:

- for \( f() \), the set consists of `int`;
- for \( g() \), the set consists of “any”;
- for `new A`, the set consists of “any”;
- for `B()`, the set is empty;
- for `new D`, the set consists of “any” and `double`.

[end example]

Given a member function \( f \) of some class \( X \), where \( f \) is an inheriting constructor (12.9) or an implicitly-declared special member function, the set of potential exceptions of the implicitly-declared member function \( f \) consists of all the members from the following sets:

- if \( f \) is a constructor,
  - the sets of potential exceptions of the constructor invocations
    - for \( X \)’s non-variant non-static data members,
    - for \( X \)’s direct base classes, and
    - if \( X \) is non-abstract (10.4), for \( X \)’s virtual base classes,
(including default argument expressions used in such invocations) as selected by overload resolution for the implicit definition of \( f \) (12.1).  

\[ \text{Note: Even though destructors for fully-constructed subobjects are invoked when an exception is thrown during the execution of a constructor (15.2), their exception specifications do not contribute to the exception specification of the constructor, because an exception thrown from such a destructor could never escape the constructor (15.1, 15.5.1). — end note} \]

— the sets of potential exceptions of the initialization of non-static data members from \textit{brace-or-equal-initializers} that are not ignored (12.6.2);

— if \( f \) is an assignment operator, the sets of potential exceptions of the assignment operator invocations for \( X \)'s non-variant non-static data members and for \( X \)'s direct base classes (including default argument expressions used in such invocations), as selected by overload resolution for the implicit definition of \( f \) (12.8);

— if \( f \) is a destructor, the sets of potential exceptions of the destructor invocations for \( X \)'s non-variant non-static data members and for \( X \)'s virtual and direct base classes.

An inheriting constructor (12.9) and an implicitly-declared special member function (Clause 12) are considered to have an implicit exception specification, as follows, where \( S \) is the set of potential exceptions of the implicitly-declared member function:

— if \( S \) contains the pseudo-type "any", the implicit exception specification is the set of all types;

— otherwise, the implicit exception specification contains all the types in \( S \).

\[ \text{Note: An instantiation of an inheriting constructor template has an implied exception specification as if it were a non-template inheriting constructor. — end note} \]  

\[ \text{Example:} \]

\begin{verbatim}
struct A {
    A(int = (A(5), 0)) noexcept;
    A(const A&) throw();
    A(A&&) throw();
    ~A() throw(X);
};
struct B {
    B() throw();
    B(const B&) = default; // exception specification contains no types
    B(B&&, int = (throw Y(), 0)) noexcept;
    ~B() throw(Y);
};
int n = 7;
struct D : public A, public B {
    int * p = new (std::nothrow) int[n];
    // exception specification of D::D() contains X and std::bad_array_new_length
    // exception specification of D::D(const D&) contains no types
    // exception specification of D::D(D&&) contains Y
    // exception specification of D::~D() contains X and Y
};
\end{verbatim}

Furthermore, if \( A::\neg A() \) or \( B::\neg B() \) were virtual, \( D::\neg D() \) would not be as restrictive as that of \( A::\neg A \), and the program would be ill-formed since a function that overrides a virtual function from a base class shall have an \textit{exception-specification} at least as restrictive as that in the base class. — end example]

\[ \text{A deallocation function (3.7.4.2) with no explicit \textit{exception-specification} has an exception specification that is the empty set.} \]

\[ \text{An \textit{exception-specification} is considered to be \textit{needed} when:} \]

\[ \text{§ 15.4} \]
— in an expression, the function is the unique lookup result or the selected member of a set of overloaded functions (3.4, 13.3, 13.4);

— the function is odr-used (3.2) or, if it appears in an unevaluated operand, would be odr-used if the expression were potentially-evaluated;

— the exception-specification is compared to that of another declaration (e.g., an explicit specialization or an overriding virtual function);

— the function is defined; or

— the exception-specification is needed for a defaulted special member function that calls the function.  
  [ Note: A defaulted declaration does not require the exception-specification of a base member function to be evaluated until the implicit exception-specification of the derived function is needed, but an explicit exception-specification 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 exception-specification of a specialization of a function template or member function of a class template is instantiated only when needed.

20 In a dynamic-exception-specification, a type-id followed by an ellipsis is a pack expansion (14.5.3).

21 [ Note: The use of dynamic-exception-specifications is deprecated (see Annex D). — end note ]

15.5 Special functions

The functions std::terminate() (15.5.1) and std::unexpected() (15.5.2) are used by the exception handling mechanism for coping with errors related to the exception handling mechanism itself. The function std::current_exception() (18.8.5) and the class std::nested_exception (18.8.6) can be used by a program to capture the currently handled exception.

15.5.1 The std::terminate() function

1 In some situations exception handling must be abandoned for less subtle error handling techniques.  [ Note: These situations are:

— when the exception handling mechanism, after completing the initialization of the exception object but before activation of a handler for the exception (15.1), calls a function that exits via an exception, or

— when the exception handling mechanism cannot find a handler for a thrown exception (15.3), or

— when the search for a handler (15.3) encounters the outermost block of a function with a noexcept-specification that does not allow the exception (15.4), or

— when the destruction of an object during stack unwinding (15.2) terminates by throwing an exception, or

— when initialization of a non-local variable with static or thread storage duration (3.6.2) exits via an exception, or

— when destruction of an object with static or thread storage duration exits via an exception (3.6.3), or

— when execution of a function registered with std::atexit or std::at_quick_exit exits via an exception (18.5), or

— when a throw-expression (5.17) with no operand attempts to rethrow an exception and no exception is being handled (15.1), or

§ 15.5.1
— when \texttt{std::unexpected} exits via an exception of a type that is not allowed by the previously violated exception specification, and \texttt{std::bad\_exception} is not included in that exception specification (15.5.2), or

— when the implementation’s default unexpected exception handler is called (D.8.1), or

— when the function \texttt{std::nested\_exception::rethrow\_nested} is called for an object that has captured no exception (18.8.6), or

— when execution of the initial function of a thread exits via an exception (30.3.1.2), or

— when the destructor or the copy assignment operator is invoked on an object of type \texttt{std::thread} that refers to a joinable thread (30.3.1.3, 30.3.1.4).

— end note

In such cases, \texttt{std::terminate()} is called (18.8.3). In the situation where no matching handler is found, it is implementation-defined whether or not the stack is unwound before \texttt{std::terminate()} is called. In the situation where the search for a handler (15.3) encounters the outermost block of a function with a \texttt{noexcept\_specification} that does not allow the exception (15.4), it is implementation-defined whether the stack is unwound, unwound partially, or not unwound at all before \texttt{std::terminate()} is called. In all other situations, the stack shall not be unwound before \texttt{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 \texttt{std::terminate()}.

15.5.2 The \texttt{std::unexpected()} function

1 If a function with a \texttt{dynamic\_exception\_specification} exits via an exception of a type that is not allowed by its exception specification, the function \texttt{std::unexpected()} is called (D.8) immediately after completing the stack unwinding for the former function.

2 [Note: By default, \texttt{std::unexpected()} calls \texttt{std::terminate()}, but a program can install its own handler function (D.8.2). In either case, the constraints in the following paragraph apply. — end note]

3 The \texttt{std::unexpected()} function shall not return, but it can throw (or rethrow) an exception. If it throws a new exception which is allowed by the exception specification which previously was violated, then the search for another handler will continue at the call of the function whose exception specification was violated. If it exits via an exception of a type that the \texttt{dynamic\_exception\_specification} does not allow, then the following happens: If the \texttt{dynamic\_exception\_specification} does not include the class \texttt{std::bad\_exception} (18.8.2) then the function \texttt{std::terminate()} is called, otherwise the thrown exception is replaced by an implementation-defined object of type \texttt{std::bad\_exception} and the search for another handler will continue at the call of the function whose \texttt{dynamic\_exception\_specification} was violated.

4 [Note: Thus, a \texttt{dynamic\_exception\_specification} guarantees that a function exits only via an exception of one of the listed types. If the \texttt{dynamic\_exception\_specification} includes the type \texttt{std::bad\_exception} then any exception type not on the list may be replaced by \texttt{std::bad\_exception} within the function \texttt{std::unexpected()}. — end note]

15.5.3 The \texttt{std::uncaught\_exceptions()} function

1 An exception is considered uncaught after completing the initialization of the exception object (15.1) until completing the activation of a handler for the exception (15.3). This includes stack unwinding. If the exception is rethrown (15.1), it is considered uncaught from the point of rethrow until the rethrown exception is caught again. The function \texttt{std::uncaught\_exceptions()} (18.8.4) returns the number of uncaught exceptions.
16 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.

preprocessing-file:
  group_opt
group:
  group-part
  group group-part
group-part:
  if-section
  control-line
  text-line
  # non-directive
if-section:
  if-group elif-groups_opt else-group_opt endif-line
if-group:
  # if constant-expression new-line group_opt
  # ifdef identifier new-line group_opt
  # ifndef identifier new-line group_opt
elif-groups:
  elif-group
  elif-groups elif-group
elif-group:
  # elif constant-expression new-line group_opt
else-group:
  # else new-line group_opt
endif-line:
  # endif new-line
control-line:
  # include pp-tokens new-line
  # define identifier replacement-list new-line
  # define identifier (paren identifier-list_opt) replacement-list new-line
  # define identifier (paren . . ) replacement-list new-line
  # define identifier (paren identifier-list, . . ) replacement-list new-line
  # undef identifier new-line
  # line pp-tokens new-line
  # error pp-tokens_opt new-line
  # pragma pp-tokens_opt new-line
  # new-line

145) 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 16.3.2, for example).
A text line shall not begin with a # preprocessing token. A non-directive shall not begin with any of the directive names appearing in the syntax.

When in a group that is skipped (16.1), the directive syntax is relaxed to allow any sequence of preprocessing tokens to occur between the directive name and the following new-line character.

The only white-space characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the introducing # preprocessing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other white-space characters in translation phase 3).

The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called preprocessing, because conceptually they occur before translation of the resulting translation unit.

The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

[Example: In:

```cpp
#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]

16.1 Conditional inclusion [cpp.cond]

The expression that controls conditional inclusion shall be an integral constant expression except that identifiers (including those lexically identical to keywords) are interpreted as described below and it may contain unary operator expressions of the form

```cpp
defined identifier
```

or

```cpp
defined ( identifier )
```

Because the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names — there simply are no keywords, enumeration constants, etc.
which evaluate to 1 if the identifier is currently defined as a macro name (that is, if it is predefined or if it has been the subject of a \#define preprocessing directive without an intervening \#undef directive with the same subject identifier), 0 if it is not.

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 (2.6).

Preprocessing directives of the forms

\[
\begin{align*}
\text{# if} & \quad \text{constant-expression new-line group}_{\text{opt}} \\
\text{# elif} & \quad \text{constant-expression new-line group}_{\text{opt}}
\end{align*}
\]

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 the defined unary operator 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. The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 5.20 using arithmetic that has at least the ranges specified in 18.3. 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 (18.4). 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. 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

\[
\begin{align*}
\text{# ifdef} & \quad \text{identifier new-line group}_{\text{opt}} \\
\text{# ifndef} & \quad \text{identifier new-line group}_{\text{opt}}
\end{align*}
\]

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. If none of the conditions evaluates to true, and there is a \#else directive, the group controlled by the \#else is processed; lacking a \#else directive, all the groups until the \#endif are skipped.

---

147) An alternative token (2.5) is not an identifier, even when its spelling consists entirely of letters and underscores. Therefore it is not subject to this replacement.

148) 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 (2.2).

149) 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.

\[
\begin{align*}
\text{#if 'z' - 'a' == 25} \\
\text{if ('z' - 'a' == 25)}
\end{align*}
\]

150) 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.
16.2 Source file inclusion

1 A `#include` directive shall identify a header or source file that can be processed by the implementation.

2 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.

3 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.

4 A preprocessing directive of the form

```
#include pp-tokens new-line
```

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `include` in the directive are processed just as in normal text (i.e., each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined.\(^{151}\) The method by which a sequence of preprocessing tokens between a `<` and a `>` preprocessing token pair or a pair of `"` characters is combined into a single header name preprocessing token is implementation-defined.

5 The implementation shall provide unique mappings for sequences consisting of one or more `nondigits` or `digits` (2.10) followed by a period (\`) and a single `nondigit`. The first character shall not be a `digit`. The implementation may ignore distinctions of alphabetical case.

6 A `#include` preprocessing directive may appear in a source file that has been read because of a `#include` directive in another file, up to an implementation-defined nesting limit.

7 [Note: Although an implementation may provide a mechanism for making arbitrary source files available to the `< >` search, in general programmers should use the `< >` form for headers provided with the implementation, and the `"` form for sources outside the control of the implementation. For instance:]

```
#include <stdio.h>
#include <unistd.h>
#include "usefullib.h"
#include "myprog.h"
```

—end note]

8 [Example: This illustrates macro-replaced `#include` directives:]

```
#if VERSION == 1
#define INCFILE "vers1.h"
#elif VERSION == 2
#define INCFILE "vers2.h" // and so on
#else
#define INCFILE "versN.h"
```

\(^{151}\) Note that adjacent string literals are not concatenated into a single string literal (see the translation phases in 2.2); thus, an expansion that results in two string literals is an invalid directive.
16.3 Macro replacement

Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

An identifier currently defined as an object-like macro 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 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.

There shall be white-space between the identifier and the replacement list in the definition of an object-like macro.

If the identifier-list in the macro definition does not end with an ellipsis, the number of arguments (including those arguments consisting of no preprocessing tokens) in an invocation of a function-like macro shall equal the number of parameters in the macro definition. Otherwise, there shall be more arguments in the invocation than there are parameters in the macro definition (excluding the ...). There shall exist a ) preprocessing token that terminates the invocation.

The identifier __VA_ARGS__ shall occur only in the replacement-list of a function-like macro that uses the ellipsis notation in the parameters.

A parameter identifier in a function-like macro shall be uniquely declared within its scope.

The identifier immediately following the define is called the macro name. There is one name space for macro names. Any white-space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro.

If a # preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive could begin, the identifier is not subject to macro replacement.

A preprocessing directive of the form

```
#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.

A preprocessing directive of the form

```
#define identifier (paren identifier-list, ... ) 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

---

152) Since, by macro-replacement time, all character literals and string literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see 2.2, translation phases), they are never scanned for macro names or parameters.

153) An alternative token (2.5) is not an identifier, even when its spelling consists entirely of letters and underscores. Therefore it is not possible to define a macro whose name is the same as that of an alternative token.
the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is terminated by the matching ) preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives, the behavior is undefined.

If there is a ... immediately preceding the ) in the function-like macro definition, then the trailing arguments, 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 one more than the number of parameters in the macro definition (excluding the ...).

16.3.1 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.

16.3.2 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.

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.

16.3.3 The ## operator

A ## preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

If, in the replacement list of a function-like macro, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument’s preprocessing token.

154) Despite the name, a non-directive is a preprocessing directive.
sequence; however, if an argument consists of no preprocessing tokens, the parameter is replaced by a placemarker preprocessing token instead.\(^\text{155}\)

For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a `##` preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemarker preprocessing tokens are handled specially: concatenation of two placemakers results in a single placemarker preprocessing token, and concatenation of a placemaker 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]

### 16.3.4 Rescanning and further replacement

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.

If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file’s preprocessing tokens), it is not replaced. Furthermore, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.

The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one, but all pragma unary operator expressions within it are then processed as specified in 16.9 below.

### 16.3.5 Scope of macro definitions

A macro definition lasts (independent of block structure) until a corresponding `#undef` directive is encountered or (if none is encountered) until the end of the translation unit. Macro definitions have no significance after translation phase 4.

A preprocessing directive of the form

\(^{155}\) Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.
# 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]

```
#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.]

```
#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]

```
#define x 3
#define f(a) f(x * (a))
#undef x
#define x 2
#define f 2
#define g f
#define z[0]
#define h g(-
#define m(a) a(w)
#define w 0,1
#define t(a) a
#define p() int
#define q(x) x
#define r(x, y) x ## y
#define str(x) # x

f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m
```
results in

```
f(2 * (y+1)) + f(2 * (f(2 * (z[0])))) % f(2 * (0)) + t(1);
f(2 * (2+3,4)-0,1)) | f(2 * (~ 5)) & f(2 * (0,1))~m(0,1);
```

6 [Example: To illustrate the rules for creating character string literals and concatenating tokens, the sequence]
#define INCFILE(n) vers ## n
#define glue(a, b) a ## b
#define xglue(a, b) glue(a, b)
#define HIGHLOW "hello"
define LOW LOW ", world"
debug(1, 2);
fputs(str(strncmp("abc\0d", "abc", '\4') // this goes away
    == 0) str(: @
), s);
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW)

results in

printf("x" "1" = %d, x" "2" = %s", x1, x2);
fputs("strncmp("abc\0d", "abc", '\4') == 0": @
), s);
#include "vers2.h" (after macro replacement, before file access)
"hello";
"hello": "", world"
or, after concatenation of the character string literals,

printf("x1=" "d, x2= "%s", x1, x2);
fputs("strncmp("abc\0d", "abc", '\4') == 0": @
), s);
#include "vers2.h" (after macro replacement, before file access)
"hello";
"hello": ", world"

Space around the # and ## tokens in the macro definition is optional. — end example]

Example: To illustrate the rules for placemarker preprocessing tokens, the sequence

#define t(x,y,z) x ## y ## z
int j[] = { t(1,2,3), t(,4,5), t(6,,7), t(8,9,)
    t(10,,), t(,11,), t(,,12), t(,,) }

results in

int j[] = { 123, 45, 67, 89,
    10, 11, 12, };

— end example]

Example: To demonstrate the redefinition rules, the following sequence is valid.

#define OBJ_LIKE (1-1)
#define OBJ_LIKE /* white space */ (1-1) /* other */
#define FUNC_LIKE(a) ( a )
#define FUNC_LIKE(b) ( a ) /* note the white space */
    /* other stuff on this line
    */

But the following redefinitions are invalid:

#define OBJ_LIKE (0) // different token sequence
#define OBJ_LIKE (1 - 1) // different white space
#define FUNC_LIKE(b) ( a ) // different parameter usage
#define FUNC_LIKE(b) ( b ) // different parameter spelling

§ 16.3.5
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__))

debug("Flag");
debug("X = %d\n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

results in

```c
fprintf(stderr, "Flag");
fprintf(stderr, "X = %d\n", x);
puts("The first, second, and third items.");
((x>y) ? puts("x>y") : printf("x is %d but y is %d", x, y));
```

— end example ]

16.4 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 (2.2) while processing the source file to the current token.

A preprocessing directive of the form

```c
# 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

```c
# 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

```c
# 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.

16.5 Error directive

A preprocessing directive of the form

```c
# 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.

16.6 Pragma directive

A preprocessing directive of the form

```c
§ 16.6 435
```
# 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.

## 16.7 Null directive [cpp.null]

A preprocessing directive of the form

```cpp
#pragma new-line
```

has no effect.

## 16.8 Predefined macro names [cpp.predefined]

The following macro names shall be defined by the implementation:

### `__cplusplus`

The name `__cplusplus` is defined to the value `201402L` when compiling a C++ translation unit.\(^{156}\)

### `__DATE__`

The date of translation of the source file: a character string literal of the form "Mmm dd yyyy", where the names of the months are the same as those generated by the `asctime` function, and the first character of `dd` is a space character if the value is less than 10. If the date of translation is not available, an implementation-defined valid date shall be supplied.

### `__FILE__`

The presumed name of the current source file (a character string literal).\(^{157}\)

### `__LINE__`

The presumed line number (within the current source file) of the current source line (an integer literal).\(^{158}\)

### `__STDC_HOSTED__`

The integer literal `1` if the implementation is a hosted implementation or the integer literal `0` if it is not.

### `__TIME__`

The time of translation of the source file: a character string literal of the form "hh:mm:ss" as in the time generated by the `asctime` function. If the time of translation is not available, an implementation-defined valid time shall be supplied.

The following macro names are conditionally defined by the implementation:

### `__STDC__`

Whether `__STDC__` is predefined and if so, what its value is, are implementation-defined.

### `__STDC_MB_MIGHT_NEQ_WC__`

The integer literal `1`, intended to indicate that, in the encoding for `wchar_t`, a member of the basic character set need not have a code value equal to its value when used as the lone character in an ordinary character literal.

### `__STDC_VERSION__`

Whether `__STDC_VERSION__` is predefined and if so, what its value is, are implementation-defined.

\(^{156}\) It is intended that future versions of this 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.

\(^{157}\) The presumed source file name can be changed by the `#line` directive.

\(^{158}\) The presumed line number can be changed by the `#line` directive.
__STDC_ISO_10646__
An integer literal of the form \texttt{yyyyMMddL} (for example, \texttt{199712L}). If this symbol is defined, then every character in the Unicode required set, when stored in an object of type 	exttt{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.

__STDCPP STRICT_POINTER_SAFETY__
Defined, and has the value integer literal 1, if and only if the implementation has strict pointer safety (3.7.4.3).

__STDCPP THREADS__
Defined, and has the value integer literal 1, if and only if a program can have more than one thread of execution (1.10).

3 The values of the predefined macros (except for __FILE__ and __LINE__) remain constant throughout the translation unit.

4 If any of the pre-defined macro names in this subclause, or the identifier \texttt{defined}, is the subject of a \texttt{#define} or a \texttt{#undef} preprocessing directive, the behavior is undefined. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

16.9 Pragma operator
A unary operator expression of the form:
\begin{small}
\begin{verbatim}
Pragma ( string-literal )
\end{verbatim}
\end{small}
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 \texttt{\"} by a double-quote, and replacing each escape sequence \texttt{\\} by a single backslash. The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the \textit{pp-tokens} in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

[Example:
\texttt{#pragma listing on ..\listing.dir}]

can also be expressed as:
\begin{small}
\begin{verbatim}
Pragma ( "listing on \\
..\listing.dir"")
\end{verbatim}
\end{small}
The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:
\begin{small}
\begin{verbatim}
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x) _Pragma(#x)
LISTING( ..\listing.dir )
\end{verbatim}
\end{small}
— end example]
17 Library introduction

17.1 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 (17.3), method of description (17.5), and organization (17.6.1) of the library. Clause 17.6, Clauses 18 through 30, 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 Clauses 18–30, as shown in Table 13.

Table 13 — Library categories

<table>
<thead>
<tr>
<th>Clause</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>Language support library</td>
</tr>
<tr>
<td>19</td>
<td>Diagnostics library</td>
</tr>
<tr>
<td>20</td>
<td>General utilities library</td>
</tr>
<tr>
<td>21</td>
<td>Strings library</td>
</tr>
<tr>
<td>22</td>
<td>Localization library</td>
</tr>
<tr>
<td>23</td>
<td>Containers library</td>
</tr>
<tr>
<td>24</td>
<td>Iterators library</td>
</tr>
<tr>
<td>25</td>
<td>Algorithms library</td>
</tr>
<tr>
<td>26</td>
<td>Numerics library</td>
</tr>
<tr>
<td>27</td>
<td>Input/output library</td>
</tr>
<tr>
<td>28</td>
<td>Regular expressions library</td>
</tr>
<tr>
<td>29</td>
<td>Atomic operations library</td>
</tr>
<tr>
<td>30</td>
<td>Thread support library</td>
</tr>
</tbody>
</table>

The language support library (Clause 18) provides components that are required by certain parts of the C++ language, such as memory allocation (5.3.4, 5.3.5) and exception processing (Clause 15).

The diagnostics library (Clause 19) provides a consistent framework for reporting errors in a C++ program, including predefined exception classes.

The general utilities library (Clause 20) includes components used by other library elements, such as a predefined storage allocator for dynamic storage management (3.7.4), and components used as infrastructure in C++ programs, such as a tuples, function wrappers, and time facilities.

The strings library (Clause 21) provides support for manipulating text represented as sequences of type `char`, sequences of type `char16_t`, sequences of type `char32_t`, sequences of type `wchar_t`, and sequences of any other character-like type.

The localization library (Clause 22) provides extended internationalization support for text processing.

The containers (Clause 23), iterators (Clause 24), and algorithms (Clause 25) libraries provide a C++ program with access to a subset of the most widely used algorithms and data structures.

The numerics library (Clause 26) provides numeric algorithms and complex number components that extend support for numeric processing. The `valarray` component provides support for n-at-a-time processing, potentially implemented as parallel operations on platforms that support such processing. The random number component provides facilities for generating pseudo-random numbers.

§ 17.1 438
The input/output library (Clause 27) 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 28) provides regular expression matching and searching.

The atomic operations library (Clause 29) allows more fine-grained concurrent access to shared data than is possible with locks.

The thread support library (Clause 30) provides components to create and manage threads, including mutual exclusion and interthread communication.

### 17.2 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 signatures and semantics of those functions. In all such cases, any use of the `restrict` qualifier shall be omitted.

### 17.3 Definitions

#### 17.3.1 arbitrary-positional stream

a stream (described in Clause 27) that can seek to any integral position within the length of the stream

[Note: Every arbitrary-positional stream is also a repositional stream. — end note]

#### 17.3.2 block

place a thread in the blocked state

#### 17.3.3 blocked thread

a thread that is waiting for some condition (other than the availability of a processor) to be satisfied before it can continue execution

#### 17.3.4 character

any object which, when treated sequentially, can represent text

[Note: 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]

#### 17.3.5 character container type

a class or a type used to represent a `character`

[Note: It is used for one of the template parameters of the string, iostream, and regular expression class templates. A character container type is a POD (3.9) type. — end note]

#### 17.3.6 comparison

---

159) This definition is taken from POSIX.
comparison function
an operator function (13.5) for any of the equality (5.10) or relational (5.9) operators

17.3.7 component
a group of library entities directly related as members, parameters, or return types
[Note: 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]

17.3.8 deadlock
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

17.3.9 default behavior
<implementation> any specific behavior provided by the implementation, within the scope of the required behavior

17.3.10 default behavior
<specification> a description of replacement function and handler function semantics

17.3.11 handler function
a non-reserved function whose definition may be provided by a C++ program
[Note: 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 18). —end note]

17.3.12 iostream class templates
templates, defined in Clause 27, that take two template arguments
[Note: 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]

17.3.13 modifier function
a class member function (9.3) other than a constructor, assignment operator, or destructor that alters the state of an object of the class

17.3.14 move construction
direct-initialization of an object of some type with an rvalue of the same type

17.3.15 § 17.3
move assignment
assignment of an rvalue of some object type to a modifiable lvalue of the same type

17.3.16
object state
the current value of all non-static class members of an object (9.2)
[Note: The state of an object can be obtained by using one or more observer functions. — end note]

17.3.17
NTCTS
a sequence of values that have character type that precede the terminating null character type value \texttt{charT()}

17.3.18
observer function
a class member function (9.3) that accesses the state of an object of the class but does not alter that state
[Note: Observer functions are specified as \texttt{const} member functions (9.3.2). — end note]

17.3.19
referenceable type
An object type, a function type that does not have cv-qualifiers or a ref-qualifier, or a reference type.
[Note: The term describes a type to which a reference can be created, including reference types. — end note]

17.3.20
replacement function
a non-reserved function whose definition is provided by a C++ program
[Note: 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 (2.2) and resolving the definitions of all translation units (3.5). — end note]

17.3.21
repositionable stream
a stream (described in Clause 27) that can seek to a position that was previously encountered

17.3.22
required behavior
a 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: If such a function defined in a C++ program fails to meet the required behavior when it executes, the behavior is undefined. — end note]

17.3.23
reserved function
a function, specified as part of the C++ standard library, that must be defined by the implementation
[Note: If a C++ program provides a definition for any reserved function, the results are undefined. — end note]

17.3.24
stable algorithm

§ 17.3
441
an algorithm that preserves, as appropriate to the particular algorithm, the order of elements
[Note: Requirements for stable algorithms are given in 17.6.5.7. — end note]

17.3.25
traits class
a 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
[Note: Traits classes defined in Clauses 21, 22 and 27 are character traits, which provide the character
handling support needed by the string and iostream classes. — end note]

17.3.26
unblock
place a thread in the unblocked state

17.3.27
valid but unspecified state
an object state 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 \textit{false}. — end example]

17.4 Additional definitions
[defs.additional]

1.3 defines additional terms used elsewhere in this International Standard.

17.5 Method of description (Informative)
[description]
1 This subclause describes the conventions used to specify the C++ standard library. 17.5.1 describes the
structure of the normative Clauses 18 through 30 and Annex D. 17.5.2 describes other editorial conventions.

17.5.1 Structure of each clause
[structure]
17.5.1.1 Elements
[structure.elements]

1 Each library clause contains the following elements, as applicable:

(1.1) — Summary
(1.2) — Requirements
(1.3) — Detailed specifications
(1.4) — References to the Standard C library

17.5.1.2 Summary
[structure.summary]
1 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 Paragraphs labeled “Note(s):” or “Example(s):” are informative, other paragraphs are normative.

3 The contents of the summary and the detailed specifications include:

(3.1) — macros

§ 17.5.1.2
17.5.1.3 Requirements

Requirements describe constraints that shall be met by a C++ program that extends the standard library. Such extensions are generally one of the following:

(1.1) — Template arguments
(1.2) — Derived classes
(1.3) — Containers, iterators, and algorithms that meet an interface convention

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 25) 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 17.5.2.1.

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.

17.5.1.4 Detailed specifications

The detailed specifications each contain the following elements:

(1.1) — name and brief description
(1.2) — synopsis (class definition or function declaration, as appropriate)
(1.3) — restrictions on template arguments, if any
(1.4) — description of class invariants
(1.5) — description of function semantics

Descriptions of class member functions follow the order (as appropriate):

(2.1) — constructor(s) and destructor
(2.2) — copying, moving & assignment functions
Descriptions of function semantics contain the following elements (as appropriate):\(^{163}\)

- Requires: the preconditions for calling the function
- Effects: the actions performed by the function
- Synchronization: the synchronization operations (1.10) applicable to the function
- 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.
- Notes: non-normative comments about 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:, Postconditions:, Returns:, Throws:, Complexity:, Remarks:, Error conditions:, and Notes: 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 Returns: elements 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 18 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.\(^{164}\)

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 (19.5).

\(^{163}\) 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.

\(^{164}\) This simplifies the presentation of complexity requirements in some cases.
17.5.1.5 C library

Paragraphs labeled “See also:” contain cross-references to the relevant portions of this International Standard and the ISO C standard, which is incorporated into this International Standard by reference.

17.5.2 Other conventions

This subclause describes several editorial conventions used to describe the contents of the C++ standard library. These conventions are for describing implementation-defined types (17.5.2.1), and member functions (17.5.2.2).

17.5.2.1 Type descriptions

17.5.2.1.1 General

The Requirements subclauses may describe names that are used to specify constraints on template arguments. These names are used in library Clauses to describe the types that may be supplied as arguments by a C++ program when instantiating template components from the library.

2 Certain types defined in Clause 27 are used to describe implementation-defined types. They are based on other types, but with added constraints.

17.5.2.1.2 Enumerated types

Several types defined in Clause 27 are enumerated types. Each enumerated type may be implemented as an enumeration or as a synonym for an enumeration.

The enumerated type enumerated can be written:

```cpp
eenum enumerated { V0, V1, V2, V3, ..... };
```

```cpp
static const enumerated C0 (V0);
static const enumerated C1 (V1);
static const enumerated C2 (V2);
static const enumerated C3 (V3);
......
```

3 Here, the names C0, C1, etc. represent enumerated elements for this particular enumerated type. All such elements have distinct values.

17.5.2.1.3 Bitmask types

Several types defined in Clauses 18 through 30 and Annex D are bitmask types. Each bitmask type can be implemented as an enumerated type that overloads certain operators, as an integer type, or as a bitset (20.6).

2 The bitmask type bitmask can be written:

```cpp
// 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, ..... }
```

```cpp
constexpr bitmask C0(V0);
constexpr bitmask C1(V1);
```

165) Examples from 17.6.3 include: EqualityComparable, LessThanComparable, CopyConstructible. Examples from 24.2 include: InputIterator, ForwardIterator, Function, Predicate.

166) Such as an integer type, with constant integer values (3.9.1).

§ 17.5.2.1.3
constexpr bitmask C2(V2);
constexpr bitmask C3(V3);
......

constexpr bitmask operator&(bitmask X, bitmask Y) {
    return static_cast<bitmask>(
        static_cast<int_type>(X) & static_cast<int_type>(Y));
}
constexpr bitmask operator|(bitmask X, bitmask Y) {
    return static_cast<bitmask>(
        static_cast<int_type>(X) | static_cast<int_type>(Y));
}
constexpr bitmask operator^ (bitmask X, bitmask Y) {
    return static_cast<bitmask>(
        static_cast<int_type>(X) ^ static_cast<int_type>(Y));
}
constexpr bitmask operator~ (bitmask X) {
    return static_cast<bitmask> (~static_cast<int_type>(X));
}

bitmask& operator&=(bitmask& X, bitmask Y){
    X = X & Y; return X;
}
bitmask& operator|=(bitmask& X, bitmask Y) {
    X = X | Y; return X;
}
bitmask& operator^=(bitmask& X, bitmask Y) {
    X = X ^ Y; return X;
}

3 Here, the names C0, C1, etc. represent bitmask elements for this particular bitmask type. All such elements have distinct, nonzero values such that, for any pair Ci and Cj where i != j, Ci & Cj is nonzero and Ci & Cj is zero. Additionally, the value 0 is used to represent an empty bitmask, in which no bitmask elements are set.

4 The following terms apply to objects and values of bitmask types:

(4.1) — To set a value Y in an object X is to evaluate the expression X |= Y.
(4.2) — To clear a value Y in an object X is to evaluate the expression X &= ~Y.
(4.3) — The value Y is set in the object X if the expression X & Y is nonzero.

17.5.2.1.4 Character sequences [character.seq]

The C standard library makes widespread use of characters and character sequences that follow a few uniform conventions:

(1.1) — A letter is any of the 26 lowercase or 26 uppercase letters in the basic execution character set.167
(1.2) — The decimal-point character is the (single-byte) character used by functions that convert between a (single-byte) character sequence and a value of one of the floating-point types. It is used in the character sequence to denote the beginning of a fractional part. It is represented in Clauses 18 through 30 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*),168 or by a change to a locale object, as described in Clauses 22.3 and 27.

167) Note that this definition differs from the definition in ISO C 7.1.1.
168) declared in <locale> (22.6).
A character sequence is an array object (8.3.4) \( A \) that can be declared as \( T \ A \ [N] \), where \( T \) is any of the types \( \text{char} \), \( \text{unsigned char} \), or \( \text{signed char} \) (3.9.1), optionally qualified by any combination of \( \text{const} \) or \( \text{volatile} \). 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.

### 17.5.2.1.4.1 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.\(^{169}\)

1. The length of an NTBS is the number of elements that precede the terminating null character. An empty NTBS has a length of zero.
2. The value of an NTBS is the sequence of values of the elements up to and including the terminating null character.
3. A static NTBS is an NTBS with static storage duration.\(^{170}\)

### 17.5.2.1.4.2 Multibyte strings

Multibyte strings are sequences of valid multibyte characters, beginning and ending in the initial shift state.\(^{171}\)

1. A null-terminated multibyte string, or NTMBS, is an NTBS that constitutes a sequence of valid multibyte characters.
2. A static NTMBS is an NTMBS with static storage duration.

### 17.5.2.2 Functions within classes

For the sake of exposition, Clauses 18 through 30 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 (12.1, 12.4, 12.8).

1. It is unspecified whether the implementation provides explicit definitions for such member function signatures, or for virtual destructors that can be generated by default.

### 17.5.2.3 Private members

Clauses 18 through 30 and Annex D do not specify the representation of classes, and intentionally omit specification of class members (9.2). An implementation may define static or non-static class members, or both, as needed to implement the semantics of the member functions specified in Clauses 18 through 30 and Annex D.

2. Objects of certain classes are sometimes required by the external specifications of their classes to store data, apparently in member objects. For the sake of exposition, some subclauses provide representative declarations, and semantic requirements, for private member objects of classes that meet the external specifications of the classes. The declarations for such member objects and the definitions of related member types are followed by a comment that ends with \textit{exposition only}, as in:

   \[\text{streambuf\* sb; // exposition only}\]

3. An implementation may use any technique that provides equivalent external behavior.

---

\(169\) Many of the objects manipulated by function signatures declared in \texttt{<cstring>} (21.8) are character sequences or NTBS. The size of some of these character sequences is limited by a length value, maintained separately from the character sequence.

\(170\) A string literal, such as "abc", is a static NTBS.

\(171\) An NTBS that contains characters only from the basic execution character set is also an NTMBS. Each multibyte character then consists of a single byte.
17.6 Library-wide requirements

This subclause specifies requirements that apply to the entire C++ standard library. Clauses 18 through 30 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, 17.6.1 describes the library’s contents and organization, 17.6.2 describes how well-formed C++ programs gain access to library entities, 17.6.3 describes constraints on types and functions used with the C++ standard library, 17.6.4 describes constraints on well-formed C++ programs, and 17.6.5 describes constraints on conforming implementations.

17.6.1 Library contents and organization

17.6.1.1 describes the entities defined in the C++ standard library. 17.6.1.2 lists the standard library headers and some constraints on those headers. 17.6.1.3 lists requirements for a freestanding implementation of the C++ standard library.

17.6.1.1 Library contents

The C++ standard library provides definitions for the following types of entities: macros, values, types, templates, classes, functions, objects.

All library entities except macros, operator new and operator delete are defined within the namespace std or namespaces nested within namespace std. It is unspecified whether names declared in a specific namespace are declared directly in that namespace or in an inline namespace inside that namespace.

Whenever a name \( x \) defined in the standard library is mentioned, the name \( x \) is assumed to be fully qualified as ::std::x, unless explicitly described otherwise. For example, if the Effects section for library function \( F \) is described as calling library function \( G \), the function ::std::G is meant.

17.6.1.2 Headers

Each element of the C++ standard library is declared or defined (as appropriate) in a header.

The C++ standard library provides 53 C++ library headers, as shown in Table 14.

```
| <algorithm> | <fstream> | <list> | <regex> | <tuple> |
| <array>     | <functional> | <locale> | <scoped_allocator> | <type_traits> |
| <atomic>    | <future> | <map> | <set> | <typeinfo> |
| <bitset>    | <initializer_list> | <memory> | <sstream> | <typeinfo> |
| <chrono>   | <iomanip> | <mutex> | <stack> | <unordered_map> |
| <codecvt>   | <ios> | <new> | <stdexcept> | <unordered_set> |
| <complex>  | <iosfwd> | <numeric> | <streambuf> | <utility> |
| <condition_variable> | <iostream> | <ostream> | <string> | <valarray> |
| <deque>    | <iterator> | <random> | <system_error> |
| <exception> | <limits> | <ratio> | <thread> |
```

The facilities of the C standard Library are provided in 26 additional headers, as shown in Table 15.

Except as noted in Clauses 18 through 30 and Annex D, the contents of each header \( c \)name shall be the same as that of the corresponding header \( name.h \), as specified in the C standard library (1.2) or the C Unicode

---

\(^{172}\) The C standard library headers (Annex D.5) also define names within the global namespace, while the C++ headers for C++
Table 15 — C++ headers for C library facilities

| <cassert> | <cinttypes> | <csignal> | <cstdio> | <cwchar> |
| <ccomplex> | <ciso646> | <cstdalign> | <cstdlib> | <cwctype> |
| <cctype> | <climits> | <cstdarg> | <cstring> |    |
| <cerrno> | <clocale> | <cstddef> | <ctime> |    |
| <cfenv> | <cmath> | <cstdint> | <cuchar> |    |
| <cstring> | | | |    |

TR, as appropriate, as if by inclusion. In the C++ standard library, however, the declarations (except for names which are defined as macros in C) are within namespace scope (3.3.6) of the namespace std. It is unspecified whether these names are first declared within the global namespace scope and are then injected into namespace std by explicit using-declarations (7.3.3).

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]

Identifiers that are keywords or operators in C++ shall not be defined as macros in C++ standard library

D.5, C standard library headers, describes the effects of using the name.h (C header) form in a C++ program.

17.6.1.3 Freestanding implementations

Two kinds of implementations are defined: hosted and freestanding (1.4). For a hosted implementation, this International Standard 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 16.

The supplied version of the header <cstdlib> shall declare at least the functions abort, atexit, at_quick_exit, exit, and quick_exit (18.5). The other headers listed in this table shall meet the same requirements as for a hosted implementation.

17.6.2 Using the library

17.6.2.1 Overview

This section describes how a C++ program gains access to the facilities of the C++ standard library. 17.6.2.2 describes effects during translation phase 4, while 17.6.2.3 describes effects during phase 8 (2.2).

17.6.2.2 Headers

The entities in the C++ standard library are defined in headers, whose contents are made available to a translation unit when it contains the appropriate #include preprocessing directive (16.2).

library facilities (17.6.1.2) may also define names within the global namespace.

This gives implementers freedom to use inline namespaces to support multiple configurations of the library.

A header is not necessarily a source file, nor are the sequences delimited by < and > in header names necessarily valid source file names (16.2).

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.

In particular, including the standard header <iso646.h> or <ciso646> has no effect.

The .h headers dump all their names into the global namespace, whereas the newer forms keep their names in namespace std. Therefore, the newer forms are the preferred forms for all uses except for C++ programs which are intended to be strictly compatible with C.
Table 16 — C++ headers for freestanding implementations

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.2 Types</td>
<td>&lt;ciso646&gt;</td>
</tr>
<tr>
<td>18.3 Implementation properties</td>
<td>&lt;cfloat&gt; &lt;limits&gt; &lt;climits&gt;</td>
</tr>
<tr>
<td>18.4 Integer types</td>
<td>&lt;cstdint&gt;</td>
</tr>
<tr>
<td>18.5 Start and termination</td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>18.6 Dynamic memory management</td>
<td>&lt;new&gt;</td>
</tr>
<tr>
<td>18.7 Type identification</td>
<td>&lt;typeinfo&gt;</td>
</tr>
<tr>
<td>18.8 Exception handling</td>
<td>&lt;exception&gt;</td>
</tr>
<tr>
<td>18.9 Initializer lists</td>
<td>&lt;initializer_list&gt;</td>
</tr>
<tr>
<td>18.10 Other runtime support</td>
<td>&lt;cstdalign&gt; &lt;cstdarg&gt; &lt;cstdbool&gt;</td>
</tr>
<tr>
<td>20.10 Type traits</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>29 Atomics</td>
<td>&lt;atomic&gt;</td>
</tr>
</tbody>
</table>

2 A translation unit may include library headers in any order (Clause 2). Each may be included more than once, with no effect different from being included exactly once, except that the effect of including either `<cassert>` or `<assert.h>` depends each time on the lexically current definition of `NDEBUG`.\(^{178}\)

3 A translation unit shall include a header only outside of any external 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.

17.6.2.3 Linkage \([\text{using.linkage}]\)

1 Entities in the C++ standard library have external linkage (3.5). Unless otherwise specified, objects and functions have the default `extern "C++"` linkage (7.5).

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.\(^{179}\)

3 Objects and functions defined in the library and required by a C++ program are included in the program prior to program startup.

See also: replacement functions (17.6.4.6), run-time changes (17.6.4.7).

17.6.3 Requirements on types and expressions \([\text{utility.requirements}]\)

1 17.6.3.1 describes requirements on types and expressions used to instantiate templates defined in the C++ standard library. 17.6.3.2 describes the requirements on swappable types and swappable expressions. 17.6.3.3 describes the requirements on pointer-like types that support null values. 17.6.3.4 describes the requirements on hash function objects. 17.6.3.5 describes the requirements on storage allocators.

17.6.3.1 Template argument requirements \([\text{utility.arg.requirements}]\)

1 The template definitions in the C++ standard library refer to various named requirements whose details are set out in tables 17–24. 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.

\(^{178}\) This is the same as the Standard C library.

\(^{179}\) The only reliable way to declare an object or function signature from the Standard C library is by including the header that declares it, notwithstanding the latitude granted in 7.1.4 of the C Standard.

§ 17.6.3.1
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 (8.5) if one of those signatures is called using the default argument (8.3.6).

### Table 17 — EqualityComparable requirements [equalitycomparable]

<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 18 — LessThanComparable requirements [lessthancomparable]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a &lt; b )</td>
<td>convertible to bool</td>
<td>(&lt;) is a strict weak ordering relation (25.4)</td>
</tr>
</tbody>
</table>

### Table 19 — DefaultConstructible requirements [defaultconstructible]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T t; )</td>
<td>object ( t ) is default-initialized</td>
</tr>
<tr>
<td>( T u; )</td>
<td>object ( u ) is value-initialized or aggregate-initialized</td>
</tr>
<tr>
<td>( T() )</td>
<td>a temporary object of type ( T ) is value-initialized or aggregate-initialized</td>
</tr>
</tbody>
</table>

### Table 20 — MoveConstructible requirements [moveconstructible]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T u = rv; )</td>
<td>( u ) is equivalent to the value of ( rv ) before the construction</td>
</tr>
<tr>
<td>( T(rv) )</td>
<td>( T(rv) ) is equivalent to the value of ( rv ) before the construction</td>
</tr>
<tr>
<td>( rv )'s state is unspecified [Note: ( rv ) must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified whether ( rv ) has been moved from or not. — end note]</td>
<td></td>
</tr>
</tbody>
</table>

### Table 21 — CopyConstructible requirements (in addition to MoveConstructible) [copyconstructible]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T u = v; )</td>
<td>the value of ( v ) is unchanged and is equivalent to ( u )</td>
</tr>
<tr>
<td>( T(v) )</td>
<td>the value of ( v ) is unchanged and is equivalent to ( T(v) )</td>
</tr>
</tbody>
</table>
### Table 22 — MoveAssignable requirements [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 = rv )</td>
<td>T&amp;</td>
<td>t</td>
<td>( t ) is equivalent to the value of ( rv ) before the assignment</td>
</tr>
</tbody>
</table>

**Note:** \( rv \)’s state is unspecified. \[Note: \( rv \) must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified whether \( rv \) has been moved from or not. — end note\]\n
### Table 23 — CopyAssignable requirements (in addition to MoveAssignable) [copyassignable]

<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 24 — Destructible requirements [destructible]

<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>

#### 17.6.3.2 Swappable requirements [swappable.requirements]

1. This subclause provides definitions for swappable types and expressions. In these definitions, let \( t \) denote an expression of type \( T \), and let \( u \) denote an expression of type \( U \).

2. An object \( t \) is swappable with an object \( u \) if and only if:

   - the expressions \( \text{swap}(t, u) \) and \( \text{swap}(u, t) \) are valid when evaluated in the context described below, and

   - these expressions have the following effects:

   - the object referred to by \( t \) has the value originally held by \( u \) and

   - the object referred to by \( u \) has the value originally held by \( t \).

3. The context in which \( \text{swap}(t, u) \) and \( \text{swap}(u, t) \) are evaluated shall ensure that a binary non-member function named “swap” is selected via overload resolution (13.3) on a candidate set that includes:

   - the two \( \text{swap} \) function templates defined in <utility> (20.2) and

   - the lookup set produced by argument-dependent lookup (3.4.2).

   \[Note: If \( T \) and \( U \) are both fundamental types or arrays of fundamental types and the declarations from the header <utility> are in scope, the overall lookup set described above is equivalent to that of the qualified name lookup applied to the expression \text{std::swap}(t, u) \) or \text{std::swap}(u, t) as appropriate. — end note\]

   \[Note: It is unspecified whether a library component that has a swappable requirement includes the header <utility> to ensure an appropriate evaluation context. — end note\]

4. An rvalue or lvalue \( t \) is swappable if and only if \( t \) is swappable with any rvalue or lvalue, respectively, of type \( T \).

5. A type \( X \) satisfying any of the iterator requirements (24.2) satisfies the requirements of ValueSwappable if, for any dereferenceable object \( x \) of type \( X \), \( \ast x \) is swappable.

\[Example: User code can ensure that the evaluation of \text{swap} \) calls is performed in an appropriate context under the various conditions as follows:\n
§ 17.6.3.2
#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

17.6.3.3 NullablePointer requirements [nullablepointer.requirements]

A NullablePointer type is a pointer-like type that supports null values. A type P meets the requirements of NullablePointer if:

1.1 P satisfies the requirements of EqualityComparable, DefaultConstructible, CopyConstructible, CopyAssignable, and Destructible.

1.2 Lvalues of type P are swappable (17.6.3.2),

1.3 the expressions shown in Table 25 are valid and have the indicated semantics, and

1.4 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 4). 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 25, 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 25 — NullablePointer requirements [nullablepointer]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>P u(np);</td>
<td></td>
<td>post: u == nullptr</td>
</tr>
<tr>
<td>P u = np;</td>
<td></td>
<td>post: P(np) == nullptr</td>
</tr>
<tr>
<td>P(np)</td>
<td></td>
<td>post: P(np) == nullptr</td>
</tr>
<tr>
<td>t = np P&amp;</td>
<td></td>
<td>post: t == nullptr</td>
</tr>
<tr>
<td>a != b</td>
<td>contextually convertible to bool</td>
<td>!(a == b)</td>
</tr>
<tr>
<td>a == np</td>
<td>contextually convertible to bool</td>
<td>a == P()</td>
</tr>
<tr>
<td>np == a</td>
<td>contextually convertible to bool</td>
<td>!(a == np)</td>
</tr>
<tr>
<td>np != a</td>
<td>contextually convertible to bool</td>
<td>!(a == np)</td>
</tr>
</tbody>
</table>

#### 17.6.3.4 Hash requirements [hash.requirements]

1. A type H meets the Hash requirements if:

   (1.1) — it is a function object type (20.9),
   (1.2) — it satisfies the requirements of CopyConstructible and Destructible (17.6.3.1), and
   (1.3) — the expressions shown in Table 26 are valid and have the indicated semantics.

2. Given Key is an argument type for function objects of type H, in Table 26 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 26 — Hash requirements [hash]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>h(k)</td>
<td>size_t</td>
<td>The value returned shall depend only on the argument k for the duration of the program. [Note: Thus all evaluations of the expression h(k) with the same value for k yield the same result for a given execution of the program. —end note] [Note: For two different values t1 and t2, the probability that h(t1) and h(t2) compare equal should be very small, approaching 1.0 / numeric_limits&lt;size_t&gt;::max(). —end note]</td>
</tr>
<tr>
<td>h(u)</td>
<td>size_t</td>
<td>Shall not modify u</td>
</tr>
</tbody>
</table>

#### 17.6.3.5 Allocator requirements [allocator.requirements]

1. The library describes a standard set of requirements for allocators, which are class-type objects that encapsulate the information about an allocation model. This information includes the knowledge of pointer types, the type of their difference, the type of the size of objects in this allocation model, as well as the memory allocation and deallocation primitives for it. All of the string types (Clause 21), containers (Clause 23) (except array), string buffers and string streams (Clause 27), and match_results (Clause 28) are parameterized in terms of allocators.
The template struct `allocator_traits` (20.7.8) supplies a uniform interface to all allocator types. Table 27 describes the types manipulated through allocators. Table 28 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 28 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 27 and 28, the use of `move` and `forward` always refers to `std::move` and `std::forward`, respectively.

### Table 27 — Descriptive variable definitions

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>T, U, C</td>
<td>any non-const object type (3.9)</td>
</tr>
<tr>
<td>V</td>
<td>a type convertible to T</td>
</tr>
<tr>
<td>X</td>
<td>an Allocator class for type T</td>
</tr>
<tr>
<td>Y</td>
<td>the corresponding Allocator class for type U</td>
</tr>
<tr>
<td>XX</td>
<td>the type <code>allocator_traits&lt;X&gt;</code></td>
</tr>
<tr>
<td>YY</td>
<td>the type <code>allocator_traits&lt;Y&gt;</code></td>
</tr>
<tr>
<td>t</td>
<td>a value of type <code>const T&amp;</code></td>
</tr>
<tr>
<td>a, a1, a2</td>
<td>values of type <code>X&amp;</code></td>
</tr>
<tr>
<td>a3</td>
<td>an rvalue of type <code>X</code></td>
</tr>
<tr>
<td>b</td>
<td>a value of type <code>Y</code></td>
</tr>
<tr>
<td>c</td>
<td>a pointer of type <code>C*</code> through which indirection is valid</td>
</tr>
<tr>
<td>p</td>
<td>a value of type <code>XX::pointer</code>, obtained by calling <code>a1.allocate</code>, where <code>a1 == a</code></td>
</tr>
<tr>
<td>q</td>
<td>a value of type <code>XX::const_pointer</code> obtained by conversion from a value <code>p</code></td>
</tr>
<tr>
<td>w</td>
<td>a value of type <code>XX::void_pointer</code> obtained by conversion from a value <code>p</code></td>
</tr>
<tr>
<td>z</td>
<td>a value of type <code>XX::const_void_pointer</code> obtained by conversion from a value <code>q</code> or a value <code>w</code></td>
</tr>
<tr>
<td>r</td>
<td>a value of type <code>T&amp;</code> obtained by the expression <code>*p</code></td>
</tr>
<tr>
<td>s</td>
<td>a value of type <code>const T&amp;</code> obtained by the expression <code>*q</code> or by conversion from a value <code>r</code></td>
</tr>
<tr>
<td>u</td>
<td>a value of type <code>XX::const_void_pointer</code> obtained by conversion from a result value of <code>YY::allocate</code>, or else a value of type (possibly <code>const</code>) <code>std::nullptr_t</code></td>
</tr>
<tr>
<td>v</td>
<td>a value of type <code>V</code></td>
</tr>
<tr>
<td>n</td>
<td>a value of type <code>XX::size_type</code></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 <code>Args&amp;&amp;</code></td>
</tr>
</tbody>
</table>

### Table 28 — Allocator requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X::pointer</code></td>
<td><code>T*</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

§ 17.6.3.5
Table 28 — Allocator requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Default</th>
</tr>
</thead>
</table>
| X::const_pointer               | X::pointer is convertible to X::const_pointer | pointer_traits<X::pointer>::rebind<
|                                | X::void_pointer, X::void_pointer and Y::void_pointer are the same type. | X::const_void_pointer, X::pointer, and X::void_pointer are convertible to X::const_void_pointer. X::const_void_pointer and Y::const_void_pointer are the same type. | X::const_void_pointer and Y::const_void_pointer are the same type. |
| X::value_type                  | Identical to T                           | a type that can represent the size of the largest object in the allocation model. | make_unsigned_t<X::difference_type> |
| X::size_type                   | unsigned integer type                   |                                  | pointer_traits<X::void_pointer>::difference_type |
| X::difference_type             | signed integer type                     | a type that can represent the difference between any two pointers in the allocation model. | pointer_traits<X::void_pointer>::difference_type |
| typename                       | Y                                       | For all U (including T), Y::template rebind<T>::other is X. | See Note A, below. |
| X::template rebind<U>::other   |                                        |                                   |         |
| *p                             | T&                                      | *q refers to the same object as *p |         |
| *q                             | const T&                                |                                   |         |
| p->m                           | type of T::m                             | pre: (*p).m is well-defined. equivalent to (*p).m |         |
| q->m                           | type of T::m                             | pre: (*q).m is well-defined. equivalent to (*q).m |         |
| static_cast<X::pointer>(w)    | X::pointer                              | static_cast<X::pointer>(w) == p |         |
| static_cast<X::const_pointer>(z)| X::const_pointer                        | static_cast<X::const_pointer>(z) == q |         |
Table 28 — Allocator requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.allocate(n)</td>
<td>X::pointer</td>
<td>Memory is allocated for n objects of type T but objects are not constructed. allocate may raise an appropriate exception. [Note: If n == 0, the return value is unspecified.] [-end note]</td>
<td>a.allocate(n)</td>
</tr>
<tr>
<td>a.allocate(n, u)</td>
<td>X::pointer</td>
<td>Same as a.allocate(n). The use of u is unspecified, but it is intended as an aid to locality.</td>
<td>a.allocate(n)</td>
</tr>
<tr>
<td>a.deallocate(p,n)</td>
<td>(not used)</td>
<td>All n T objects in the area pointed to by p shall be destroyed prior to this call. n shall match the value passed to allocate to obtain this memory. Does not throw exceptions. [Note:p shall not be singular.] [-end note]</td>
<td></td>
</tr>
<tr>
<td>a.max_size()</td>
<td>X::size_type</td>
<td>the largest value that can meaningfully be passed to X::allocate()</td>
<td>numeric_limits&lt;size_type&gt;::max()</td>
</tr>
<tr>
<td>a1 == a2</td>
<td>bool</td>
<td>returns true only if storage allocated from each can be deallocated via the other. [operator== shall be reflexive, symmetric, and transitive, and shall not exit via an exception.]</td>
<td></td>
</tr>
<tr>
<td>a1 != a2</td>
<td>bool</td>
<td>same as !(a1 == a2)</td>
<td></td>
</tr>
<tr>
<td>a == b</td>
<td>bool</td>
<td>same as a == Y::rebind&lt;T&gt;::other(b)</td>
<td></td>
</tr>
<tr>
<td>a != b</td>
<td>bool</td>
<td>same as !(a == b)</td>
<td></td>
</tr>
<tr>
<td>X a1(a);</td>
<td></td>
<td>Shall not exit via an exception.</td>
<td></td>
</tr>
<tr>
<td>X a1 = a;</td>
<td></td>
<td>post: a1 == a</td>
<td></td>
</tr>
<tr>
<td>X a(b);</td>
<td></td>
<td>Shall not exit via an exception.</td>
<td></td>
</tr>
<tr>
<td>post: Y(a) == b, a == X(b)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X a1(move(a));</td>
<td></td>
<td>Shall not exit via an exception.</td>
<td></td>
</tr>
<tr>
<td>X a1 = move(a);</td>
<td></td>
<td>post: a1 equals the prior value of a.</td>
<td></td>
</tr>
<tr>
<td>X a(move(b));</td>
<td></td>
<td>Shall not exit via an exception.</td>
<td></td>
</tr>
<tr>
<td>post: a equals the prior value of X(b).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a.construct(c, args)</td>
<td>(not used)</td>
<td>Effect: Constructs an object of type C at c ::new ((void*)c) C(forward&lt; Args&gt;(args)...)</td>
<td></td>
</tr>
</tbody>
</table>

§ 17.6.3.5
Table 28 — Allocator requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.destroy(c)</td>
<td>(not used)</td>
<td>Effect: Destroys the object at c</td>
</tr>
<tr>
<td>a.select_on_container_copy_construction()</td>
<td>X</td>
<td>Typically returns either a or X()</td>
</tr>
<tr>
<td>X::propagate_on_container_copy_assignment</td>
<td>Identical to or derived from true_type or false_type</td>
<td>true_type only if an allocator of type X should be copied when the client container is copy-assigned. See Note B, below.</td>
</tr>
<tr>
<td>X::propagate_on_container_move_assignment</td>
<td>Identical to or derived from true_type or false_type</td>
<td>true_type only if an allocator of type X should be moved when the client container is move-assigned. See Note B, below.</td>
</tr>
<tr>
<td>X::propagate_on_container_swap</td>
<td>Identical to or derived from true_type or false_type</td>
<td>true_type only if an allocator of type X should be swapped when the client container is swapped. See Note B, below.</td>
</tr>
<tr>
<td>X::is_always_equal</td>
<td>Identical to or derived from true_type or false_type</td>
<td>true_type only if the expression a1 == a2 is guaranteed to be true for any two (possibly const) values a1, a2 of type X.</td>
</tr>
</tbody>
</table>

3 Note A: The member class template rebind in the table above is effectively a typedef template. [Note: In general, if the name Allocator is bound to SomeAllocator<T>, then Allocator::rebind<U>::other is the same type as SomeAllocator<U>, where SomeAllocator<T>::value_type is T and SomeAllocator<U>::value_type is U. — end note] If Allocator is a class template instantiation of the form SomeAllocator<T, Args>, where Args is zero or more type arguments, and Allocator does not supply a rebind member template, the standard allocator_traits template uses SomeAllocator<U, Args> in place of Allocator::rebind<U>::other by default. For allocator types that are not template instantiations of the above form, no default is provided.

4 Note B: If X::propagate_on_container_copy_assignment::value is true, X shall satisfy the CopyAssignable requirements (Table 23) and the copy operation shall not throw exceptions. If X::propagate_on_container_move_assignment::value is true, X shall satisfy the MoveAssignable requirements (Table 22) and the move operation shall not throw exceptions. If X::propagate_on_container_swap::value is true, lvalues of type X shall be swappable (17.6.3.2) and the swap operation shall not throw exceptions.

5 An allocator type X shall satisfy the requirements of CopyConstructible (17.6.3.1). The X::pointer, X::const_pointer, X::void_pointer, and X::const_void_pointer types shall satisfy the requirements of NullablePointer (17.6.3.3). No constructor, comparison operator, copy operation, move operation, or swap operation on these types shall exit via an exception. X::pointer and X::const_pointer shall also satisfy the requirements for a random access iterator (24.2).

180) 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.
Let $x_1$ and $x_2$ denote objects of (possibly different) types $X::\text{void\_pointer}$, $X::\text{const\_void\_pointer}$, $X::\text{pointer}$, or $X::\text{const\_pointer}$. Then, $x_1$ and $x_2$ are \textit{equivalently-valued} pointer values, if and only if both $x_1$ and $x_2$ can be explicitly converted to the two corresponding objects $px_1$ and $px_2$ of type $X::\text{const\_pointer}$, using a sequence of \texttt{static\_cast}s using only these four types, and the expression $px_1 == px_2$ evaluates to \texttt{true}.

Let $w_1$ and $w_2$ denote objects of type $X::\text{void\_pointer}$. Then for the expressions

\begin{verbatim}
 w1 == w2
 w1 != w2
\end{verbatim}

either or both objects may be replaced by an equivalently-valued object of type $X::\text{const\_void\_pointer}$ with no change in semantics.

Let $p_1$ and $p_2$ denote objects of type $X::\text{pointer}$. Then for the expressions

\begin{verbatim}
 p1 == p2
 p1 != p2
 p1 < p2
 p1 <= p2
 p1 > p2
 p1 >= p2
 p1 - p2
\end{verbatim}

either or both objects may be replaced by an equivalently-valued object of type $X::\text{const\_pointer}$ with no change in semantics.

An allocator may constrain the types on which it can be instantiated and the arguments for which its \texttt{construct} member may be called. If a type cannot be used with a particular allocator, the allocator class or the call to \texttt{construct} may fail to instantiate.

[Example: the following is an allocator class template supporting the minimal interface that satisfies the requirements of Table 28:]

\begin{verbatim}
 template <class Tp>
 struct SimpleAllocator {
    typedef Tp value_type;
    SimpleAllocator(ctor args);
    
    template <class T> SimpleAllocator(const SimpleAllocator<T>& other);
    
    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>&);
 template <class T, class U>
 bool operator!=(const SimpleAllocator<T>&, const SimpleAllocator<U>&);

—end example]

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 \texttt{allocate} for that type may fail by throwing an object of type \texttt{std::bad\_alloc}. —end note]
17.6.4 Constraints on programs

17.6.4.1 Overview

This section 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 (17.6.4.2.1), its use of various reserved names (17.6.4.3), its use of headers (17.6.4.4), its use of standard library classes as base classes (17.6.4.5), its definitions of replacement functions (17.6.4.6), and its installation of handler functions during execution (17.6.4.7).

17.6.4.2 Namespace use

17.6.4.2.1 Namespace std

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.\(^{181}\)

The behavior of a C++ program is undefined if it declares

1. an explicit specialization of any member function of a standard library class template, or
2. an explicit specialization of any member function template of a standard library class or class template, or
3. an explicit or partial specialization of any member class template of a standard library class or class template.

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.

A translation unit shall not declare namespace std to be an inline namespace (7.3.1).

17.6.4.2.2 Namespace posix

The behavior of a C++ program is undefined if it adds declarations or definitions to namespace posix or to a namespace within namespace posix unless otherwise specified. The namespace posix is reserved for use by ISO/IEC 9945 and other POSIX standards.

17.6.4.3 Reserved names

The C++ standard library reserves the following kinds of names:

1. macros
2. global names
3. names with external linkage

If a program declares or defines a name in a context where it is reserved, other than as explicitly allowed by this Clause, its behavior is undefined.

\(^{181}\) 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 the Standard.
17.6.4.3.1 Macro names

A translation unit that includes a standard library header shall not `#define` or `#undef` names declared in any standard library header.

A translation unit shall not `#define` or `#undef` names lexically identical to keywords, to the identifiers listed in Table 2, or to the attribute-tokens described in 7.6.

17.6.4.3.2 External linkage

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, \(^{182}\) both in namespace std and in the global namespace.

Each global function signature declared with external linkage in a header is reserved to the implementation to designate that function signature with external linkage. \(^{183}\)

Each name from the Standard C 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.

Each function signature from the Standard C library declared with external linkage is reserved to the implementation for use as a function signature with both `extern "C"` and `extern "C++"` linkage, \(^{184}\) or as a name of namespace scope in the global namespace.

17.6.4.3.3 Types

For each type T from the Standard C library, \(^{185}\) the types `::T` and `std::T` are reserved to the implementation and, when defined, `::T` shall be identical to `std::T`.

17.6.4.3.4 User-defined literal suffixes

Literal suffix identifiers (13.5.8) that do not start with an underscore are reserved for future standardization.

17.6.4.4 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 (16.2), the behavior is undefined.

17.6.4.5 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 (10.3).

17.6.4.6 Replacement functions

Clauses 18 through 30 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 (17.3).

A C++ program may provide the definition for any of twelve dynamic memory allocation function signatures declared in header `<new>` (3.7.4, 18.6):

(2.1) — `operator new(std::size_t)`

(2.2) — `operator new(std::size_t, const std::nothrow_t&)`

\(^{182}\) The list of such reserved names includes `errno`, declared or defined in `<cerrno>`.

\(^{183}\) The list of such reserved function signatures with external linkage includes `setjmp(jmp_buf)`, declared or defined in `<csetjmp>`, and `va_end(va_list)`, declared or defined in `<stdarg>`.

\(^{184}\) The function signatures declared in `<cuchar>`, `<cwchar>`, and `<cwctype>` are always reserved, notwithstanding the restrictions imposed in subclause 4.5.1 of Amendment 1 to the C Standard for these headers.

\(^{185}\) 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`. 

§ 17.6.4.6
The program’s definitions are used instead of the default versions supplied by the implementation (18.6). Such replacement occurs prior to program startup (3.2, 3.6). The program’s declarations shall not be specified as inline. No diagnostic is required.

17.6.4.7 Handler functions

The C++ standard library provides default versions of the following handler functions (Clause 18):

1. unexpected_handler
2. terminate_handler

A C++ program may install different handler functions during execution, by supplying a pointer to a function defined in the program or the library as an argument to (respectively):

1. set_new_handler
2. set_unexpected
3. set_terminate

See also: subclauses 18.6.2, Storage allocation errors, and 18.8, Exception handling.

A C++ program can get a pointer to the current handler function by calling the following functions:

1. get_new_handler
2. get_unexpected
3. 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.
17.6.4.8 Other functions

1 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, the Standard places no requirements on the implementation.

2 In particular, the effects are undefined in the following cases:

(2.1) for replacement functions (18.6.1), if the installed replacement function does not implement the semantics of the applicable Required behavior: paragraph.

(2.2) for handler functions (18.6.2.3, 18.8.3.1, D.8.1), if the installed handler function does not implement the semantics of the applicable Required behavior: paragraph.

(2.3) for types used as template arguments when instantiating a template component, if the operations on the type do not implement the semantics of the applicable Requirements subclause (17.6.3.5, 23.2, 24.2, 26.2). Operations on such types can report a failure by throwing an exception unless otherwise specified.

(2.4) if any replacement function or handler function or destructor operation exits via an exception, unless specifically allowed in the applicable Required behavior: paragraph.

(2.5) if an incomplete type (3.9) is used as a template argument when instantiating a template component, unless specifically allowed for that component.

17.6.4.9 Function arguments

1 Each of the following applies to all arguments to functions defined in the C++ standard library, unless explicitly stated otherwise.

(1.1) If an argument to a function has an invalid value (such as a value outside the domain of the function or a pointer invalid for its intended use), the behavior is undefined.

(1.2) If a function argument is described as being an array, the pointer actually passed to the function shall have a value such that all address computations and accesses to objects (that would be valid if the pointer did point to the first element of such an array) are in fact valid.

(1.3) If a function argument binds to an rvalue reference parameter, the implementation may assume that this parameter is a unique reference to this argument. [Note: If the parameter is a generic parameter of the form T&& and an lvalue of type A is bound, the argument binds to an lvalue reference (14.8.2.1) 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 move(x)), the program is effectively asking that function to treat that lvalue as a temporary. The implementation is free to optimize away aliasing checks which might be needed if the argument was an lvalue. —end note]

17.6.4.10 Shared objects and the library

1 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 17.6.5.9. [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 sharable without data races or the user supplies a locking mechanism. —end note]

2 [Note: In particular, the program is required to ensure that completion of the constructor of any object of a class type defined in the standard library happens before any other member function invocation on that object and, unless otherwise specified, to ensure that completion of any member function invocation other
than destruction on such an object happens before destruction of that object. This applies even to objects such as mutexes intended for thread synchronization. — end note]

17.6.4.11 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.

17.6.5 Conforming implementations

17.6.5.1 Overview

This section describes the constraints upon, and latitude of, implementations of the C++ standard library.

1 An implementation’s use of headers is discussed in 17.6.5.2, its use of macros in 17.6.5.3, global functions in 17.6.5.4, member functions in 17.6.5.5, data race avoidance in 17.6.5.9, access specifiers in 17.6.5.10, class derivation in 17.6.5.11, and exceptions in 17.6.5.12.

17.6.5.2 Headers

1 A C++ header may include other C++ headers. A C++ header shall provide the declarations and definitions that appear in its synopsis. A C++ header shown in its synopsis as including other C++ headers shall provide the declarations and definitions that appear in the synopses of those other headers.

2 Certain types and macros are defined in more than one header. Every such entity shall be defined such that any header that defines it may be included after any other header that also defines it (3.2).

3 The C standard headers (D.5) shall include only their corresponding C++ standard header, as described in 17.6.1.2.

17.6.5.3 Restrictions on macro definitions

1 The names and global function signatures described in 17.6.1.1 are reserved to the implementation.

2 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.

17.6.5.4 Global and non-member functions

1 It is unspecified whether any global or non-member functions in the C++ standard library are defined as inline (7.1.2).

2 A call to a global or non-member function signature described in Clauses 18 through 30 and Annex D shall behave as if the implementation declared no additional global or non-member function signatures.\(^\text{186}\)

3 An implementation shall not declare a global or non-member function signature with additional default arguments.

4 Unless otherwise specified, global and non-member functions in the standard library shall not use functions from another namespace which are found through argument-dependent name lookup (3.4.2). [Note: The phrase “unless otherwise specified” is intended to allow argument-dependent lookup in cases like that of ostream_iterator::operator= (24.6.2.2):]

Effects:

*out_stream << value;
if (delim != 0)
  *out_stream << delim;
return *this;

\(^{186}\) A valid C++ program always calls the expected library global or non-member function. An implementation may also define additional global or non-member functions that would otherwise not be called by a valid C++ program.
17.6.5.5 Member functions

It is unspecified whether any member functions in the C++ standard library are defined as inline (7.1.2). An implementation may declare additional non-virtual member function signatures within a class:

1. by adding arguments with default values to a member function signature;¹
2. by replacing a member function signature with default values by two or more member function signatures with equivalent behavior; and
3. by adding a member function signature for a member function name.

A call to a member function signature described in the C++ standard library behaves as if the implementation declares no additional member function signatures.²

17.6.5.6 constexpr functions and constructors

This standard explicitly requires that certain standard library functions are constexpr (7.1.5). 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.

17.6.5.7 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).

17.6.5.8 Reentrancy

Except where explicitly specified in this standard, it is implementation-defined which functions in the Standard C++ library may be recursively reentered.

17.6.5.9 Data race avoidance

This section specifies requirements that implementations shall meet to prevent data races (1.10). 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 (1.10) 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 (1.10) 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.

¹ Hence, the address of a member function of a class in the C++ standard library has an unspecified type.
² A valid C++ program always calls the expected library member function, or one with equivalent behavior. An implementation may also define additional member functions that would otherwise not be called by a valid C++ program.
[Note: This means, for example, that implementations can’t use a static object for internal purposes without synchronization because it could cause a data race even in programs that do not explicitly share objects between threads. — end note]

A C++ standard library function shall not access objects indirectly accessible via its arguments or via elements of its container arguments except by invoking functions required by its specification on those container elements.

Operations on iterators obtained by calling a standard library container or string member function may access the underlying container, but shall not modify it. [Note: In particular, container operations that invalidate iterators conflict with operations on iterators associated with that container. — end note]

Implementations may share their own internal objects between threads if the objects are not visible to users and are protected against data races.

Unless otherwise specified, C++ standard library functions shall perform all operations solely within the current thread if those operations have effects that are visible (1.10) to users.

[Note: This allows implementations to parallelize operations if there are no visible side effects. — end note]

17.6.5.10 Protection within classes

It is unspecified whether any function signature or class described in Clauses 18 through 30 and Annex D is a friend of another class in the C++ standard library.

17.6.5.11 Derived classes

An implementation may derive any class in the C++ standard library from a class with a name reserved to the implementation.

Certain classes defined in the C++ standard library are required to be derived from other classes in the C++ standard library. An implementation may derive such a class directly from the required base or indirectly through a hierarchy of base classes with names reserved to the implementation.

In any case:

(3.1) — Every base class described as virtual shall be virtual;
(3.2) — Every base class described as non-virtual shall not be virtual;
(3.3) — Unless explicitly stated otherwise, types with distinct names shall be distinct types.\(^\text{189}\)

17.6.5.12 Restrictions on exception handling

Any of the functions defined in the C++ standard library can report a failure by throwing an exception of a type described in its Throws: paragraph. An implementation may strengthen the exception specification for a non-virtual function by adding a non-throwing noexcept-specification.

A function may throw an object of a type not listed in its Throws clause if its type is derived from a type named in the Throws clause and would be caught by an exception handler for the base type.

Functions from the C standard library shall not throw exceptions\(^\text{190}\) except when such a function calls a program-supplied function that throws an exception.\(^\text{191}\)

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. Any other functions

\(^{189}\) There is an implicit exception to this rule for types that are described as synonyms for basic integral types, such as size_t (18.2) and streamoff (27.5.2).

\(^{190}\) 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.

\(^{191}\) The functions qsort() and bsearch() (25.5) meet this condition.
defined in the C++ standard library that do not have an exception-specification may throw implementation-defined exceptions unless otherwise specified.\(^{192}\) An implementation may strengthen this implicit exception-specification by adding an explicit one.\(^{193}\)

### 17.6.5.13 Restrictions on storage of pointers  
[ref: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 (3.7.4.3). [Note: Other libraries are strongly encouraged to do the same, since not doing so may result in accidental use of pointers that are not safely derived. Libraries that store pointers outside the user’s address space should make it appear that they are stored and retrieved from a traceable pointer location. — end note]

### 17.6.5.14 Value of error codes  
[ref:value.error.codes]

Certain functions in the C++ standard library report errors via a `std::error_code` (19.5.2.1) object. That object’s `category()` member shall return `std::system_category()` for errors originating from the operating system, or a reference to an implementation-defined `error_category` object for errors originating elsewhere. The implementation shall define the possible values of `value()` for each of these error categories. [Example: For operating systems that are based on POSIX, implementations are encouraged to 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 are encouraged to 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]

### 17.6.5.15 Moved-from state of library types  
[ref:lib.types.movedfrom]

Objects of types defined in the C++ standard library may be moved from (12.8). 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.

---

\(^{192}\) 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` (18.6.2.1). Library implementations should report errors by throwing exceptions of or derived from the standard exception classes (18.6.2.1, 18.8, 19.2).

\(^{193}\) That is, an implementation may provide an explicit exception-specification that defines the subset of “any” exceptions thrown by that function. This implies that the implementation may list implementation-defined types in such an exception-specification.
18  Language support library  
[language.support]

18.1 General  
[support.general]

1 This Clause describes the function signatures that are called implicitly, and the types of objects generated implicitly, during the execution of some C++ programs. It also describes the headers that declare these function signatures and define any related types.

2 The following subclauses describe common type definitions used throughout the library, characteristics of the predefined types, functions supporting start and termination of a C++ program, support for dynamic memory management, support for dynamic type identification, support for exception processing, support for initializer lists, and other runtime support, as summarized in Table 29.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.2  Types</td>
<td>&lt;cstdint&gt;</td>
</tr>
<tr>
<td>18.3  Implementation properties</td>
<td>&lt;limits&gt;</td>
</tr>
<tr>
<td>18.4  Integer types</td>
<td>&lt;cstddef&gt;</td>
</tr>
<tr>
<td>18.5  Start and termination</td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>18.6  Dynamic memory management</td>
<td>&lt;new&gt;</td>
</tr>
<tr>
<td>18.7  Type identification</td>
<td>&lt;typeinfo&gt;</td>
</tr>
<tr>
<td>18.8  Exception handling</td>
<td>&lt;exception&gt;</td>
</tr>
<tr>
<td>18.9  Initializer lists</td>
<td>&lt;initializer_list&gt;</td>
</tr>
<tr>
<td>18.10 Other runtime support</td>
<td>&lt;cstdarg&gt;</td>
</tr>
</tbody>
</table>

18.2 Types  
[support.types]

1 Table 30 describes the header <cstdint>.

Table 30 — Header <cstdint> synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td>NULL</td>
</tr>
<tr>
<td>Types:</td>
<td>ptrdiff_t</td>
</tr>
<tr>
<td></td>
<td>max_align_t</td>
</tr>
</tbody>
</table>

2 The contents are the same as the Standard C library header <stddef.h>, with the following changes:
The macro `NULL` is an implementation-defined C++ null pointer constant in this International Standard (4.10).\textsuperscript{194}

The macro `offsetof(type, member-designator)` accepts a restricted set of `type` arguments in this International Standard. If `type` is not a standard-layout class (Clause 9), the results are undefined.\textsuperscript{195} The expression `offsetof(type, member-designator)` is never type-dependent (14.6.2.2) and it is value-dependent (14.6.2.3) if and only if `type` is dependent. The result of applying the `offsetof` macro to a field that is 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 5.7.

The type `size_t` is an implementation-defined unsigned integer type that is large enough to contain the size in bytes of any object.

\[ Note: \text{It is recommended that implementations choose types for `ptrdiff_t` and `size_t` whose integer conversion ranks (4.13) are no greater than that of `signed long int` unless a larger size is necessary to contain all the possible values.} \text{— end note} \]

The type `max_align_t` is a POD type whose alignment requirement is at least as great as that of every scalar type, and whose alignment requirement is supported in every context.

The type `nullptr_t` is defined as follows:

```cpp
namespace std {
    typedef decltype(nullptr) nullptr_t;
}
```

The type for which `nullptr_t` is a synonym has the characteristics described in 3.9.1 and 4.10. \[ Note: \text{Although `nullptr_t`'s address cannot be taken, the address of another `nullptr_t` object that is an lvalue can be taken.} \text{— end note} \]

See also: Alignment (3.11), Sizeof (5.3.3), Additive operators (5.7), Free store (12.5), and ISO C 7.1.6.

18.3 Implementation properties [support.limits]

18.3.1 In general [support.limits.general]

The headers `<limits>` (18.3.2), `<climits>`, and `<cfloat>` (18.3.3) supply characteristics of implementation-dependent arithmetic types (3.9.1).

18.3.2 Numeric limits [limits]

18.3.2.1 Class template `numeric_limits` [limits.numeric]

The `numeric_limits` class template provides a C++ program with information about various properties of the implementation’s representation of the arithmetic types.

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`.

For all members declared static const `constexpr` in the `numeric_limits` template, specializations shall define these values in such a way that they are usable as constant expressions.

Non-arithmetic standard types, such as `complex<T> (26.4.2)`, shall not have specializations.

18.3.2.2 Header `<limits>` synopsis [limits.syn]
namespace std {
    template<class T> class numeric_limits;
    enum float_round_style;
    enum float_denorm_style;

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>;
template<> class numeric_limits<float>;
template<> class numeric_limits<double>;
template<> class numeric_limits<long double>;
}

18.3.2.3 Class template numeric_limits

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 T epsilon() noexcept { return T(); }
    static constexpr T round_error() noexcept { return T(); }
    static constexpr int min_exponent = 0;
    static constexpr int min_exponent10 = 0;
    static constexpr int max_exponent = 0;
    static constexpr int max_exponent10 = 0;
    static constexpr bool has_infinity = false;
    static constexpr bool has_quiet_NaN = false;
    static constexpr bool has_signaling_NaN = false;
}
static constexpr float_denorm_style has_denorm = denorm_absent;
static constexpr bool has_denorm_loss = false;
static constexpr T infinity() noexcept { return T(); }
static constexpr T quiet_NaN() noexcept { return T(); }
static constexpr T signaling_NaN() noexcept { return T(); }
static constexpr T denorm_min() noexcept { return T(); }
static constexpr bool is_iec559 = false;
static constexpr bool is_bounded = false;
static constexpr bool is_modulo = false;
static constexpr bool traps = false;
static constexpr float_round_style round_style = round_toward_zero;
};

template<class T> class numeric_limits<const T>;

1 The default numeric_limits<T> template shall have all members, but with 0 or false values.
2 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.

18.3.2.4 numeric_limits members

static constexpr T min() noexcept;  
1 Minimum finite value.\(^{196}\)
2 For floating types with denormalization, returns the minimum positive normalized value.
3 Meaningful for all specializations in which is_bounded != false, or is_bounded == false && is_-signed == false.

static constexpr T max() noexcept;  
4 Maximum finite value.\(^{197}\)
5 Meaningful for all specializations in which is_bounded != false.

static constexpr T lowest() noexcept;  
6 A finite value x such that there is no other finite value y where y < x.\(^ {198}\)
7 Meaningful for all specializations in which is_bounded != false.

static constexpr int digits;  
8 Number of radix digits that can be represented without change.
9 For integer types, the number of non-sign bits in the representation.
10 For floating point types, the number of radix digits in the mantissa.\(^ {199}\)

\(^{196}\) Equivalent to CHAR_MIN, SHRT_MIN, FLT_MIN, DBL_MIN, etc.
\(^{197}\) Equivalent to CHAR_MAX, SHRT_MAX, FLT_MAX, DBL_MAX, etc.
\(^{198}\) 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.
\(^{199}\) Equivalent to FLT_MANT_DIG, DBL_MANT_DIG, LDBL_MANT_DIG.
static constexpr int digits10;
    Number of base 10 digits that can be represented without change.\(^\text{200}\)
    Meaningful for all specializations in which \textit{is\_bounded} \(!= \text{false}\).

static constexpr int max_digits10;
    Number of base 10 digits required to ensure that values which differ are always differentiated.
    Meaningful for all floating point types.

static constexpr bool is_signed;
    True if the type is signed.
    Meaningful for all specializations.

static constexpr bool is_integer;
    True if the type is integer.
    Meaningful for all specializations.

static constexpr bool is_exact;
    True if the type uses an exact representation. All integer types are exact, but not all exact types are integer. For example, rational and fixed-exponent representations are exact but not integer.
    Meaningful for all specializations.

static constexpr int radix;
    For floating types, specifies the base or radix of the exponent representation (often 2).\(^\text{201}\)
    For integer types, specifies the base of the representation.\(^\text{202}\)
    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.\(^\text{203}\)
    Meaningful for all floating point types.

static constexpr T round_error() noexcept;
    Measure of the maximum rounding error.\(^\text{204}\)

static constexpr int min_exponent;
    Minimum negative integer such that \textit{radix} raised to the power of one less than that integer is a normalized floating point number.\(^\text{205}\)
    Meaningful for all floating point types.

static constexpr int min_exponent10;
\(^\text{200}\) Equivalent to \texttt{FLT\_DIG}, \texttt{DBL\_DIG}, \texttt{LDBL\_DIG}.
\(^\text{201}\) Equivalent to \texttt{FLT\_RADIX}.
\(^\text{202}\) Distinguishes types with bases other than 2 (e.g. BCD).
\(^\text{203}\) Equivalent to \texttt{FLT\_EPSILON}, \texttt{DBL\_EPSILON}, \texttt{LDBL\_EPSILON}.
\(^\text{204}\) Rounding error is described in ISO/IEC 10967-1 Language independent arithmetic - Part 1 Section 5.2.8 and Annex A Rationale Section A.5.2.8 - Rounding constants.
\(^\text{205}\) Equivalent to \texttt{FLT\_MIN\_EXP}, \texttt{DBL\_MIN\_EXP}, \texttt{LDBL\_MIN\_EXP}.
Minimum negative integer such that 10 raised to that power is in the range of normalized floating point numbers.\textsuperscript{206}

Meaningful for all floating point types.

\begin{verbatim}
static constexpr int max_exponent;
\end{verbatim}

Maximum positive integer such that \texttt{radix} raised to the power one less than that integer is a representable finite floating point number.\textsuperscript{207}

Meaningful for all floating point types.

\begin{verbatim}
static constexpr int max_exponent10;
\end{verbatim}

Maximum positive integer such that 10 raised to that power is in the range of representable finite floating point numbers.\textsuperscript{208}

Meaningful for all floating point types.

\begin{verbatim}
static constexpr bool has_infinity;
\end{verbatim}

True if the type has a representation for positive infinity.

Meaningful for all floating point types.

Shall be \texttt{true} for all specializations in which \texttt{is_iec559} \emph{\texttt{!= false}}.

\begin{verbatim}
static constexpr bool has_quiet_NaN;
\end{verbatim}

True if the type has a representation for a quiet (non-signaling) “Not a Number.”\textsuperscript{209}

Meaningful for all floating point types.

Shall be \texttt{true} for all specializations in which \texttt{is_iec559} \emph{\texttt{!= false}}.

\begin{verbatim}
static constexpr bool has_signaling_NaN;
\end{verbatim}

True if the type has a representation for a signaling “Not a Number.”\textsuperscript{210}

Meaningful for all floating point types.

Shall be \texttt{true} for all specializations in which \texttt{is_iec559} \emph{\texttt{!= false}}.

\begin{verbatim}
static constexpr float_denorm_style has_denorm;
\end{verbatim}

denorm_present if the type allows denormalized values (variable number of exponent bits)\textsuperscript{211}, denorm_absent if the type does not allow denormalized values, and \texttt{denorm_indeterminate} if it is indeterminate at compile time whether the type allows denormalized values.

Meaningful for all floating point types.

\begin{verbatim}
static constexpr bool has_denorm_loss;
\end{verbatim}

True if loss of accuracy is detected as a denormalization loss, rather than as an inexact result.\textsuperscript{212}

\begin{verbatim}
static constexpr T infinity() noexcept;
\end{verbatim}

\texttt{Equivalent to FLT_MIN_10_EXP}, \texttt{DBL_MIN_10_EXP}, \texttt{LDBL_MIN_10_EXP}.

\texttt{Equivalent to FLT_MAX_10_EXP}, \texttt{DBL_MAX_10_EXP}, \texttt{LDBL_MAX_10_EXP}.

\texttt{Equivalent to FLT_MAX_10_EXP}, \texttt{DBL_MAX_10_EXP}, \texttt{LDBL_MAX_10_EXP}.

\texttt{Required by LIA-1.}

\texttt{Required by LIA-1.}

\texttt{Required by LIA-1.}

\texttt{See IEC 559.}

\texttt{\textsuperscript{\textcopyright ISO/IEC N4296 206}}

\texttt{\textsuperscript{\textcopyright ISO/IEC N4296 207}}

\texttt{\textsuperscript{\textcopyright ISO/IEC N4296 208}}

\texttt{\textsuperscript{\textcopyright ISO/IEC N4296 209}}

\texttt{\textsuperscript{\textcopyright ISO/IEC N4296 210}}

\texttt{\textsuperscript{\textcopyright ISO/IEC N4296 211}}

\texttt{\textsuperscript{\textcopyright ISO/IEC N4296 212}}

\texttt{\textsuperscript{\textcopyright ISO/IEC N4296 206}}

\texttt{\textsuperscript{\textcopyright ISO/IEC N4296 207}}

\texttt{\textsuperscript{\textcopyright ISO/IEC N4296 208}}

\texttt{\textsuperscript{\textcopyright ISO/IEC N4296 209}}

\texttt{\textsuperscript{\textcopyright ISO/IEC N4296 210}}

\texttt{\textsuperscript{\textcopyright ISO/IEC N4296 211}}

\texttt{\textsuperscript{\textcopyright ISO/IEC N4296 212}}

\texttt{§ 18.3.2.4}
47 Representation of positive infinity, if available.\textsuperscript{213}
48 Meaningful for all specializations for which has\_infinity != false. Required in specializations for which is\_iec559 != false.

static constexpr T quiet\_NaN() noexcept;
49 Representation of a quiet “Not a Number,” if available.\textsuperscript{214}
50 Meaningful for all specializations for which has\_quiet\_NaN != false. Required in specializations for which is\_iec559 != false.

static constexpr T signaling\_NaN() noexcept;
51 Representation of a signaling “Not a Number,” if available.\textsuperscript{215}
52 Meaningful for all specializations for which has\_signaling\_NaN != false. Required in specializations for which is\_iec559 != false.

static constexpr T denorm\_min() noexcept;
53 Minimum positive denormalized value.\textsuperscript{216}
54 Meaningful for all floating point types.
55 In specializations for which has\_denorm == false, returns the minimum positive normalized value.

static constexpr bool is\_iec559;
56 True if and only if the type adheres to IEC 559 standard.\textsuperscript{217}
57 Meaningful for all floating point types.

static constexpr bool is\_bounded;
58 True if the set of values representable by the type is finite.\textsuperscript{218} \[ Note: All fundamental types (3.9.1) are bounded. This member would be false for arbitrary precision types. --- end note \]
59 Meaningful for all specializations.

static constexpr bool is\_modulo;
60 True if the type is modulo.\textsuperscript{219} A type is modulo if, for any operation involving +, -, or \* on values of that type whose result would fall outside the range [min(),max()], the value returned differs from the true value by an integer multiple of max() - min() + 1.
61 On most machines, this is false for floating types, true for unsigned integers, and true for signed integers.
62 Meaningful for all specializations.

static constexpr bool traps;
63 \texttt{true} if, at program startup, there exists a value of the type that would cause an arithmetic operation using that value to trap.\textsuperscript{220}
64 Meaningful for all specializations.

\textsuperscript{213} Required by LIA-1.
\textsuperscript{214} Required by LIA-1.
\textsuperscript{215} Required by LIA-1.
\textsuperscript{216} Required by LIA-1.
\textsuperscript{217} International Electrotechnical Commission standard 559 is the same as IEEE 754.
\textsuperscript{218} Required by LIA-1.
\textsuperscript{219} Required by LIA-1.
\textsuperscript{220} Required by LIA-1.
static constexpr bool tinyness_before;

true if tinyness is detected before rounding.\textsuperscript{221}

Meaningful for all floating point types.

static constexpr float_round_style round_style;

The rounding style for the type.\textsuperscript{222}

Meaningful for all floating point types. Specializations for integer types shall return \texttt{round_toward_-zero}.

\section*{18.3.2.5 Type float_round_style [round.style]}

```
namespace std {
    enum float_round_style {
        round_indeterminate = -1,
        round_toward_zero = 0,
        round_to_nearest = 1,
        round_toward_infinity = 2,
        round_toward_neg_infinity = 3
    };
}
```

\begin{enumerate}
\item The rounding mode for floating point arithmetic is characterized by the values:
\begin{enumerate}
\item \texttt{round_indeterminate} if the rounding style is indeterminable
\item \texttt{round_toward_zero} if the rounding style is toward zero
\item \texttt{round_to_nearest} if the rounding style is to the nearest representable value
\item \texttt{round_toward_infinity} if the rounding style is toward infinity
\item \texttt{round_toward_neg_infinity} if the rounding style is toward negative infinity
\end{enumerate}
\end{enumerate}

\section*{18.3.2.6 Type float_denorm_style [denorm.style]}

```
namespace std {
    enum float_denorm_style {
        denorm_indeterminate = -1,
        denorm_absent = 0,
        denorm_present = 1
    };
}
```

\begin{enumerate}
\item The presence or absence of denormalization (variable number of exponent bits) is characterized by the values:
\begin{enumerate}
\item \texttt{denorm_indeterminate} if it cannot be determined whether or not the type allows denormalized values
\item \texttt{denorm_absent} if the type does not allow denormalized values
\item \texttt{denorm_present} if the type does allow denormalized values
\end{enumerate}
\end{enumerate}

\textsuperscript{221) Refer to IEC 559. Required by LIA-1.}
\textsuperscript{222) Equivalent to FLT_ROUNDS. Required by LIA-1.}
18.3.2.7 numeric_limits specializations

All members shall be provided for all specializations. However, many values are only required to be meaningful under certain conditions (for example, epsilon() is only meaningful if is_integer is false). Any value that is not “meaningful” shall be set to 0 or false.

Example:

```cpp
namespace std {
    template<> class numeric_limits<float> {
        public:
            static constexpr bool is_specialized = true;
            inline static constexpr float min() noexcept { return 1.17549435E-38F; }
            inline static constexpr float max() noexcept { return 3.40282347E+38F; }
            inline 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;
            inline static constexpr float epsilon() noexcept { return 1.19209290E-07F; }
            inline 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;

            inline static constexpr float infinity() noexcept { return value; }
            inline static constexpr float quiet_NaN() noexcept { return value; }
            inline static constexpr float signaling_NaN() noexcept { return value; }
            inline 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]
The specialization for \texttt{bool} shall be provided as follows:

```cpp
namespace std {
    template<> class numeric_limits<bool> {
        public:
            static constexpr bool is_specialized = true;
            static constexpr bool min() noexcept { return false; }
            static constexpr bool max() noexcept { return true; }
            static constexpr bool lowest() noexcept { return false; }
            static constexpr int digits = 1;
            static constexpr int digits10 = 0;
            static constexpr int max_digits10 = 0;
            static constexpr bool is_signed = false;
            static constexpr bool is_integer = true;
            static constexpr bool is_exact = true;
            static constexpr int radix = 2;
            static constexpr bool epsilon() noexcept { return 0; }
            static constexpr bool round_error() noexcept { return 0; }
            static constexpr int min_exponent = 0;
            static constexpr int min_exponent10 = 0;
            static constexpr int max_exponent = 0;
            static constexpr int max_exponent10 = 0;
            static constexpr bool has_infinity = false;
            static constexpr bool has_quiet_NaN = false;
            static constexpr bool has_signaling_NaN = false;
            static constexpr float_denorm_style has_denorm = denorm_absent;
            static constexpr bool has_denorm_loss = false;
            static constexpr bool infinity() noexcept { return 0; }
            static constexpr bool quiet_NaN() noexcept { return 0; }
            static constexpr bool signaling_NaN() noexcept { return 0; }
            static constexpr bool denorm_min() noexcept { return 0; }
            static constexpr bool is_iec559 = false;
            static constexpr bool is_bounded = true;
            static constexpr bool is_modulo = false;
            static constexpr bool traps = false;
            static constexpr bool tinyness_before = false;
            static constexpr float_round_style round_style = round_toward_zero;
    };
}
```

### 18.3.3 C library

Table 31 describes the header \texttt{climits}. The contents are the same as the Standard C library header \texttt{limits.h}. [Note: The types of the constants defined by macros in \texttt{climits} are not required to match the types to which the macros refer. — end note]

Table 32 describes the header \texttt{cfloat}. The contents are the same as the Standard C library header \texttt{float.h}.

See also: ISO C 7.1.5, 5.2.4.2.2, 5.2.4.2.1.
18.4 Integer types

namespace std {
    typedef signed integer type int8_t; // optional
    typedef signed integer type int16_t; // optional
    typedef signed integer type int32_t; // optional
    typedef signed integer type int64_t; // optional
    typedef signed integer type int_fast8_t;
    typedef signed integer type int_fast16_t;
    typedef signed integer type int_fast32_t;
    typedef signed integer type int_fast64_t;
    typedef signed integer type int_least8_t;
    typedef signed integer type int_least16_t;
    typedef signed integer type int_least32_t;
    typedef signed integer type int_least64_t;
    typedef signed integer type intmax_t;
    typedef signed integer type intptr_t; // optional
    typedef unsigned integer type uint8_t; // optional
    typedef unsigned integer type uint16_t; // optional
    typedef unsigned integer type uint32_t; // optional
    typedef unsigned integer type uint64_t; // optional
    typedef unsigned integer type uint_fast8_t;
    typedef unsigned integer type uint_fast16_t;
    typedef unsigned integer type uint_fast32_t;
    typedef unsigned integer type uint_fast64_t;
}
typedef unsigned integer type uint_least8_t;
typedef unsigned integer type uint_least16_t;
typedef unsigned integer type uint_least32_t;
typedef unsigned integer type uint_least64_t;

typedef unsigned integer type uintmax_t;
typedef unsigned integer type uintptr_t;  // optional
}
// namespace std

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 functions, types, and macros the same as 7.18 in the C standard. [Note: The macros defined by `<cstdint>` are provided unconditionally. In particular, the symbols `__STDC_LIMIT_MACROS` and `__STDC_CONSTANT_MACROS` (mentioned in footnotes 219, 220, and 222 in the C standard) play no role in C++. —end note]

18.5 Start and termination [support.start.term]

Table 33 describes some of the contents of the header `<cstdlib>`.

Table 33 — Header `<cstdlib>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td>EXIT_FAILURE</td>
</tr>
<tr>
<td>Functions:</td>
<td>_Exit</td>
</tr>
<tr>
<td></td>
<td>at_quick_exit</td>
</tr>
</tbody>
</table>

The contents are the same as the Standard C library header `<stdlib.h>`, with the following changes:

[[noreturn]] void _Exit(int status) noexcept;

The function `_Exit(int status)` has additional behavior in this International Standard:

(3.1) — The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling functions passed to `atexit()` (3.6.3).

[[noreturn]] void abort(void) noexcept;

The function `abort()` has additional behavior in this International Standard:

(4.1) — The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling functions passed to `atexit()` (3.6.3).

extern "C" int atexit(void (*)(void)) noexcept;
extern "C++" int atexit(void (*)(void)) noexcept;

§ 18.5
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 (1.10) a call to `exit()` will succeed. [Note: The `atexit()` functions do not introduce a data race (17.6.5.9). — end note]

Implementation limits: The implementation shall support the registration of at least 32 functions.

Returns: The `atexit()` function returns zero if the registration succeeds, non-zero if it fails.

[[noreturn]] void `exit`(int `status`)

The function `exit()` has additional behavior in this International Standard:

(8.1) — First, objects with thread storage duration and associated with the current thread are destroyed. Next, objects with static storage duration are destroyed and functions registered by calling `atexit` are called. See 3.6.3 for the order of destructions and calls. (Automatic objects are not destroyed as a result of calling `exit()`.)\(^{223}\) If control leaves a registered function called by `exit` because the function does not provide a handler for a thrown exception, `std::terminate()` shall be called (15.5.1).

(8.2) — Next, all open C streams (as mediated by the function signatures declared in `<cstdio>`) with unwritten buffered data are flushed, all open C streams are closed, and all files created by calling `tmpfile()` are removed.

(8.3) — Finally, control is returned to the host environment. If `status` is zero or `EXIT_SUCCESS`, an implementation-defined form of the status `successful termination` is returned. If `status` is `EXIT_FAILURE`, an implementation-defined form of the status `unsuccessful termination` is returned. Otherwise the status returned is implementation-defined.\(^{225}\)

```c
extern "C" int at_quick_exit(void (*f)(void)) noexcept;
extern "C++" int at_quick_exit(void (*f)(void)) noexcept;
```

Effects: The `at_quick_exit()` functions register the function pointed to by \( f \) to be called without arguments when `quick_exit` is called. It is unspecified whether a call to `at_quick_exit()` that does not happen before (1.10) all calls to `quick_exit` will succeed. [Note: The `at_quick_exit()` functions do not introduce a data race (17.6.5.9). — end note] [Note: The order of registration may be indeterminate if `at_quick_exit` was called from more than one thread. — end note] [Note: The `at_quick_exit` registrations are distinct from the `atexit` registrations, and applications may need to call both registration functions with the same argument. — end note]

Implementation limits: The implementation shall support the registration of at least 32 functions.

Returns: Zero if the registration succeeds, non-zero if it fails.

[[noreturn]] void `quick_exit`(int `status`) noexcept;

Effects: Functions registered by calls to `at_quick_exit` are called in the reverse order of their registration, except that a function shall be called after any previously registered functions that had already been called at the time it was registered. Objects shall not be destroyed as a result of calling `quick_exit`. If control leaves a registered function called by `quick_exit` because the function does not provide a handler for a thrown exception, `std::terminate()` shall be called. [Note: `at_quick_exit` may call a registered function from a different thread than the one that registered it, so registered functions should not rely on the identity of objects with thread storage duration. — end note] After calling registered functions, `quick_exit` shall call `_Exit(status)`.

— end note]

\(^{223}\) A function is called for every time it is registered.

\(^{224}\) Objects with automatic storage duration are all destroyed in a program whose function `main()` contains no automatic objects and executes the call to `exit()`. Control can be transferred directly to such a `main()` by throwing an exception that is caught in `main()`.

\(^{225}\) The macros `EXIT_FAILURE` and `EXIT_SUCCESS` are defined in `<cstdlib>`.

§ 18.5 480
18.6 Dynamic memory management

The header `<new>` defines several functions that manage the allocation of dynamic storage in a program. It also defines components for reporting storage management errors.

### Header `<new>` synopsis

```cpp
namespace std {
    class bad_alloc;
    class bad_array_new_length;
    struct nothrow_t {};
    extern const nothrow_t nothrow;
    typedef void (*new_handler)();
    new_handler get_new_handler() noexcept;
    new_handler set_new_handler(new_handler new_p) noexcept;
}
```

```cpp
void* operator new(std::size_t size);
void* operator new(std::size_t size, const std::nothrow_t&) noexcept;
void operator delete(void* ptr) noexcept;
void operator delete(void* ptr, const std::nothrow_t&) noexcept;
void operator delete(void* ptr, std::size_t size) noexcept;
void operator delete(void* ptr, std::size_t size, const std::nothrow_t&) noexcept;
void* operator new[](std::size_t size);
void* operator new[](std::size_t size, const std::nothrow_t&) noexcept;
void operator delete[](void* ptr) noexcept;
void operator delete[](void* ptr, const std::nothrow_t&) noexcept;
void operator delete[](void* ptr, std::size_t size) noexcept;
void operator delete[](void* ptr, std::size_t size, const std::nothrow_t&) noexcept;
void* operator new (std::size_t size, void* ptr) noexcept;
void* operator new[](std::size_t size, void* ptr) noexcept;
void operator delete (void* ptr, void*) noexcept;
void operator delete[](void* ptr, void*) noexcept;
```

### Storage allocation and deallocation

Except where otherwise specified, the provisions of (3.7.4) apply to the library versions of `operator new` and `operator delete`.

#### Single-object forms

```cpp
void* operator new(std::size_t size);
```

1. **Effects:** The allocation function (3.7.4.1) called by a `new-expression` (5.3.4) to allocate `size` bytes of storage suitably aligned to represent any object of that size.
2. **Replaceable:** A C++ program may define a function with this function signature that displaces the default version defined by the C++ standard library.
3. **Required behavior:** Return a non-null pointer to suitably aligned storage (3.7.4), or else throw a `bad_alloc` exception. This requirement is binding on a replacement version of this function.
4. **Default behavior:**

§ 18.6.1.1
— Executes a loop: Within the loop, the function first attempts to allocate the requested storage. Whether the attempt involves a call to the Standard C library function malloc is unspecified.

— Returns a pointer to the allocated storage if the attempt is successful. Otherwise, if the current
new_handler (18.6.2.5) is a null pointer value, throws bad_alloc.

— Otherwise, the function calls the current new_handler function (18.6.2.3). 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.

void* operator new(std::size_t size, const std::nothrow_t&) noexcept;

Effects: Same as above, except that it is 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 a function with this function signature that displaces the
default version defined by the C++ standard library.

Required behavior: Return a non-null pointer to suitably aligned storage (3.7.4), or else return a null
pointer. This nothrow version of operator new returns a pointer obtained as if acquired from the
(possibly replaced) ordinary version. This requirement is binding on a replacement version of this
function.

Default behavior: Calls operator new(size). If the call returns normally, returns the result of that
call. Otherwise, returns a null pointer.

[ Example:
    T* p1 = new T;           // throws bad_alloc if it fails
    T* p2 = new(nothrow) T; // returns nullptr if it fails

— end example ]

void operator delete(void* ptr) noexcept;
void operator delete(void* ptr, std::size_t size) noexcept;

Effects: The deallocation function (3.7.4.2) called by a delete-expression to render the value of ptr
invalid.

Replaceable: a C++ program may define a function with signature void operator delete(void* ptr) noexcept that displaces the default version defined by the C++ standard library. If this function (without size parameter) is defined, the program should also define void operator delete(void* ptr, std::size_t size) noexcept. If this function with size parameter is defined, the program shall also define the 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 be a value returned by an earlier call to the (possibly replaced) operator new(std::size_t) or operator new(std::size_t, const std::nothrow_t&) which has not been invalidated by an intervening call to operator delete(void*).

Requires: If an implementation has strict pointer safety (3.7.4.3) then ptr shall be a safely-derived
pointer.

Requires: If present, the std::size_t size argument shall equal the size argument passed to the
allocation function that returned ptr.

Required behavior: Calls to operator delete(void* ptr, std::size_t size) may be changed to calls to operator delete(void* ptr) without affecting memory allocation. [Note: A conforming

§ 18.6.1.1
implementation is for `operator delete(void* ptr, std::size_t size)` to simply call `operator delete(ptr). — end note]

Default behavior: the function `operator delete(void* ptr, std::size_t size)` calls `operator delete(ptr). [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 `calloc`, `malloc`, or `realloc`, declared in `<cstdlib>`.

```cpp
void operator delete(void* ptr, const std::nothrow_t&) noexcept;
void operator delete(void* ptr, std::size_t size, const std::nothrow_t&) noexcept;
```

Effects: The deallocation function (3.7.4.2) 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 a function with signature `void operator delete(void* ptr, const std::nothrow_t&) noexcept` that displaces the default version defined by the C++ standard library. If this function (without `size` parameter) is defined, the program should also define `void operator delete(void* ptr, std::size_t size, const std::nothrow_t&) noexcept`. If this function with `size` parameter is defined, the program shall also define the 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: If an implementation has strict pointer safety (3.7.4.3) then `ptr` shall be a safely-derived pointer.

Requires: If present, the `std::size_t size` argument must equal the size argument passed to the allocation function that returned `ptr`.

Required behavior: Calls to `operator delete(void* ptr, std::size_t size, const std::nothrow_t&)` may be changed to calls to `operator delete(void* ptr, const std::nothrow_t&)` without affecting memory allocation. [Note: A conforming implementation is for `operator delete(void* ptr, std::size_t size, const std::nothrow_t&)` to simply call `operator delete(void* ptr, const std::nothrow_t&)`. — end note]

Default behavior: `operator delete(void* ptr, std::size_t size, const std::nothrow_t&)` calls `operator delete(ptr, std::nothrow)`, and `operator delete(void* ptr, const std::nothrow_t&)` calls `operator delete(ptr)`. 

### 18.6.1.2 Array forms

```cpp
void* operator new[](std::size_t size);
```

Effects: The allocation function (3.7.4.1) called by the array form of a `new-expression` (5.3.4) to allocate `size` bytes of storage suitably aligned to represent any array object of that size or smaller.\(^2\)

Replaceable: a C++ program can define a function with this function signature that displaces the default version defined by the C++ standard library.

Required behavior: Same as for `operator new(std::size_t)`. This requirement is binding on a replacement version of this function.

Default behavior: Returns `operator new(size)`.

---

\(^2\) It is not the direct responsibility of `operator new[](std::size_t)` or `operator delete[](void*)` 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[](std::size_t)` to obtain space to store supplemental information.

§ 18.6.1.2
```c
void* operator new[](std::size_t size, const std::nothrow_t&) noexcept;
```

**Effects:** Same as above, except that it is 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 can define a function with this function signature that displaces the default version defined by the C++ standard library.

**Required behavior:** Return a non-null pointer to suitably aligned storage (3.7.4), or return a null pointer. This requirement is binding on a replacement version of this function.

**Default behavior:** Calls `operator new[](size)`. If the call returns normally, returns the result of that call. Otherwise, returns a null pointer.

```c
void operator delete[](void* ptr) noexcept;
void operator delete[](void* ptr, std::size_t size) noexcept;
```

**Effects:** The deallocation function (3.7.4.2) called by the array form of a delete-expression to render the value of ptr invalid.

**Replaceable:** a C++ program can define a function with signature `void operator delete[](void* ptr) noexcept` that displaces the default version defined by the C++ standard library. If this function (without size parameter) is defined, the program should also define `void operator delete[](void* ptr, std::size_t size) noexcept`. If this function with size parameter is defined, the program shall also define the 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 be the value returned by an earlier call to `operator new[](std::size_t) or operator new[](std::size_t, const std::nothrow_t&)` which has not been invalidated by an intervening call to `operator delete[](void*)`.

**Requires:** If present, the std::size_t size argument must equal the size argument passed to the allocation function that returned ptr.

**Required behavior:** Calls to `operator delete[](void* ptr, std::size_t size)` may be changed to calls to `operator delete[](void* ptr)` without affecting memory allocation. [Note: A conforming implementation is for `operator delete[](void* ptr, std::size_t size)` to simply call `operator delete[](void* ptr)` — end note]

**Requires:** If an implementation has strict pointer safety (3.7.4.3) then ptr shall be a safely-derived pointer.

**Default behavior:** `operator delete[](void* ptr, std::size_t size)` calls `operator delete[](ptr)`, and `operator delete[](void* ptr)` calls `operator delete(ptr)`.

```c
void operator delete[](void* ptr, const std::nothrow_t&) noexcept;
void operator delete[](void* ptr, std::size_t size, const std::nothrow_t&) noexcept;
```

**Effects:** The deallocation function (3.7.4.2) called by the implementation to render the value of ptr invalid when the constructor invoked from a nothrow placement version of the array new-expression throws an exception.

**Replaceable:** a C++ program may define a function with signature `void operator delete[](void* ptr, const std::nothrow_t&) noexcept` that displaces the default version defined by the C++ standard library. If this function (without size parameter) is defined, the program should also define `void operator delete[](void* ptr, std::size_t size, const std::nothrow_t&) noexcept`. If this function with size parameter is defined, the program shall also define the 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: If an implementation has strict pointer safety (3.7.4.3) then \( \text{ptr} \) shall be a safely-derived pointer.

Requires: If present, the `std::size_t size` argument must equal the size argument passed to the allocation function that returned \( \text{ptr} \).

Required behavior: Calls to `operator delete[]` may be changed to calls to `operator delete[]` without affecting memory allocation. [Note: A conforming implementation is for `operator delete[]` to simply call `operator delete[]` without affecting memory allocation. —end note]

Default behavior: `operator delete[]` calls `operator delete[]` , and `operator delete[]` calls `operator delete[]` .

### 18.6.1.3 Placement forms

These functions are reserved, a C++ program may not define functions that displace the versions in the Standard C++ library (17.6.4). The provisions of (3.7.4) do not apply to these reserved placement forms of `operator new` and `operator delete`.

```cpp
void* operator new(std::size_t size, void* ptr) noexcept;
```

Returns: \( \text{ptr} \).

Remarks: Intentionally performs no other action.

[Example: This can be useful for constructing an object at a known address:

```cpp
void* place = operator new(sizeof(Something));
Something* p = new (place) Something();
```

—end example]

```cpp
void* operator new[](std::size_t size, void* ptr) noexcept;
```

Returns: \( \text{ptr} \).

Remarks: Intentionally performs no other action.

```cpp
void operator delete(void* ptr, void*) noexcept;
```

Effects: Intentionally performs no action.

Requires: If an implementation has strict pointer safety (3.7.4.3) then \( \text{ptr} \) 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 (5.3.4).

```cpp
void operator delete[](void* ptr, void*) noexcept;
```

Effects: Intentionally performs no action.

Requires: If an implementation has strict pointer safety (3.7.4.3) then \( \text{ptr} \) 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 (5.3.4).
18.6.1.4 Data races

For purposes of determining the existence of data races, the library versions of \texttt{operator new}, user replacement versions of global \texttt{operator new}, the C standard library functions \texttt{calloc} and \texttt{malloc}, the library versions of \texttt{operator delete}, user replacement versions of \texttt{operator delete}, the C standard library function \texttt{free}, and the C standard library function \texttt{realloc} shall not introduce a data race (17.6.5.9). 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 (1.10) the next allocation (if any) in this order.

18.6.2 Storage allocation errors

18.6.2.1 Class \texttt{bad_alloc}

namespace \texttt{std} {
    class \texttt{bad_alloc} : public \texttt{exception} {
        public:
            \texttt{bad_alloc}() \texttt{noexcept};
            \texttt{bad_alloc}(const \texttt{bad_alloc}&) \texttt{noexcept};
            \texttt{bad_alloc}& \texttt{operator}=(const \texttt{bad_alloc}&) \texttt{noexcept};
            virtual const \texttt{char*} \texttt{what}() const \texttt{noexcept};
        }
    }

The class \texttt{bad_alloc} defines the type of objects thrown as exceptions by the implementation to report a failure to allocate storage.

\texttt{bad_alloc}() \texttt{noexcept};

- **Effects:** Constructs an object of class \texttt{bad_alloc}.
- **Remarks:** The result of calling \texttt{what()} on the newly constructed object is implementation-defined.

\texttt{bad_alloc}(const \texttt{bad_alloc}&) \texttt{noexcept};
\texttt{bad_alloc}& \texttt{operator}=(const \texttt{bad_alloc}&) \texttt{noexcept};

- **Effects:** Copies an object of class \texttt{bad_alloc}.
- **Returns:** An implementation-defined \texttt{ntbs}.

18.6.2.2 Class \texttt{bad_array_new_length}

namespace \texttt{std} {
    class \texttt{bad_array_new_length} : public \texttt{bad_alloc} {
        public:
            \texttt{bad_array_new_length}() \texttt{noexcept};
        }
    }

The class \texttt{bad_array_new_length} defines the type of objects thrown as exceptions by the implementation to report an attempt to allocate an array of size less than zero or greater than an implementation-defined limit (5.3.4).

\texttt{bad_array_new_length}() \texttt{noexcept};

- **Effects:** constructs an object of class \texttt{bad_array_new_length}.
- **Remarks:** the result of calling \texttt{what()} on the newly constructed object is implementation-defined.
18.6.2.3 Type new_handler

typedef void (*new_handler)();
1 The type of a handler function to be called by operator new() or operator new[]() (18.6.1) when they cannot satisfy a request for additional storage.

2 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;

18.6.2.4 set_new_handler

new_handler set_new_handler(new_handler new_p) noexcept;
1 Effects: Establishes the function designated by new_p as the current new_handler.

2 Returns: The previous new_handler.

3 Remarks: The initial new_handler is a null pointer.

18.6.2.5 get_new_handler

new_handler get_new_handler() noexcept;
1 Returns: The current new_handler. [Note: This may be a null pointer value. — end note]

18.7 Type identification [support.rtti]

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.

Header <typeinfo> synopsis

namespace std {
    class type_info;
    class bad_cast;
    class bad_typeid;
}

See also: 5.2.7, 5.2.8.

18.7.1 Class type_info [type.info]

namespace std {
    class type_info {
    public:
        virtual ~type_info();
        bool operator==(const type_info& rhs) const noexcept;
        bool operator!=(const type_info& rhs) const noexcept;
        bool before(const type_info& rhs) const noexcept;
        size_t hash_code() const noexcept;
        const char* name() const noexcept;
        type_info(const type_info& rhs) = delete; // cannot be copied
        type_info& operator=(const type_info& rhs) = delete; // cannot be copied
    };
}
The class \texttt{type\_info} describes type information generated by the implementation. 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.

\begin{verbatim}
bool operator==(const type\_info& rhs) const noexcept;
  Effects: Compares the current object with \texttt{rhs}.
  Returns: \texttt{true} if the two values describe the same type.

bool operator!=(const type\_info& rhs) const noexcept;
  Returns: \texttt{!(*this \(\neq\) rhs)}.

bool before(const type\_info& rhs) const noexcept;
  Effects: Compares the current object with \texttt{rhs}.
  Returns: \texttt{true} if \texttt{*this} precedes \texttt{rhs} in the implementation’s collation order.

size\_t hash\_code() const noexcept;
  Returns: An unspecified value, except that within a single execution of the program, it shall return
  the same value for any two \texttt{type\_info} objects which compare equal.
  Remark: an implementation should return different values for two \texttt{type\_info} objects which do not
  compare equal.

const char* name() const noexcept;
  Returns: An implementation-defined ntbs.
  Remarks: The message may be a null-terminated multibyte string (17.5.2.1.4.2), suitable for conversion
  and display as a wstring (21.3, 22.4.1.4)
\end{verbatim}

\textbf{18.7.2 Class bad\_cast}

\begin{verbatim}
namespace std {
  class bad\_cast : public exception {
public:
    bad\_cast() noexcept;
    bad\_cast(const bad\_cast&) noexcept;
    bad\_cast& operator=(const bad\_cast&) noexcept;
    virtual const char* what() const noexcept;
  };
}
\end{verbatim}

The class \texttt{bad\_cast} defines the type of objects thrown as exceptions by the implementation to report the execution of an invalid \textit{dynamic-cast} expression (5.2.7).

\begin{verbatim}
bad\_cast() noexcept;
  Effects: Constructs an object of class \texttt{bad\_cast}.
  Remarks: The result of calling \texttt{what()} on the newly constructed object is implementation-defined.

bad\_cast(const bad\_cast&) noexcept;
bad\_cast& operator=(const bad\_cast&) noexcept;
  Effects: Copies an object of class \texttt{bad\_cast}.
\end{verbatim}
virtual const char* what() const noexcept;
5    
6    Returns: An implementation-defined NTBS.
7    Remarks: The message may be a null-terminated multibyte string (17.5.2.1.4.2), suitable for conversion and display as a wstring (21.3, 22.4.1.4)

18.7.3 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;
            virtual const char* what() const noexcept;
        }
    }

1 The class bad_typeid defines the type of objects thrown as exceptions by the implementation to report a null pointer in a typeid expression (5.2.8).

bad_typeid() noexcept;
2    Effects: Constructs an object of class bad_typeid.
3    Remarks: The result of calling what() on the newly constructed object is implementation-defined.

bad_typeid(const bad_typeid&) noexcept;
bad_typeid& operator=(const bad_typeid&) noexcept;
4    Effects: Copies an object of class bad_typeid.

virtual const char* what() const noexcept;
5    Returns: An implementation-defined NTBS.
6    Remarks: The message may be a null-terminated multibyte string (17.5.2.1.4.2), suitable for conversion and display as a wstring (21.3, 22.4.1.4)

18.8 Exception handling

The header <exception> defines several types and functions related to the handling of exceptions in a C++ program.

Header <exception> synopsis

namespace std {
    class exception;
    class bad_exception;
    class nested_exception;

    typedef void (*unexpected_handler)();
    unexpected_handler get_unexpected() noexcept;
    unexpected_handler set_unexpected(unexpected_handler f) noexcept;
    [[noreturn]] void unexpected();

    typedef void (*terminate_handler)();
    terminate_handler get_terminate() noexcept;
    terminate_handler set_terminate(terminate_handler f) noexcept;
    [[noreturn]] void terminate();

§ 18.8
bool uncaught_exception() noexcept;
int uncaught_exceptions() noexcept;

typedef unspecified exception_ptr;

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);
}

See also: 15.5.

18.8.1 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;

Effects: Constructs an object of class exception.

exception(const exception& rhs) noexcept;
exception& operator=(const exception& rhs) noexcept;

Effects: Copies an exception object.

Postcondition: If *this and rhs both have dynamic type exception then strcmp(what(), rhs.what()) shall equal 0.

virtual ~exception();

Effects: Destroys an object of class exception.

virtual const char* what() const noexcept;

Returns: An implementation-defined NTBS.

Remarks: The message may be a null-terminated multibyte string (17.5.2.1.4.2), suitable for conversion and display as a wstring (21.3, 22.4.1.4). 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.
18.8.2 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;
        virtual const char* what() const noexcept;
    }
};

The class bad_exception defines the type of objects thrown as described in (15.5.2).

bad_exception() noexcept;

Effects: Constructs an object of class bad_exception.

Remarks: The result of calling what() on the newly constructed object is implementation-defined.

bad_exception(const bad_exception&) noexcept;
bad_exception& operator=(const bad_exception&) noexcept;

Effects: Copies an object of class bad_exception.

virtual const char* what() const noexcept;

Returns: An implementation-defined NTBS.

Remarks: The message may be a null-terminated multibyte string (17.5.2.1.4.2), suitable for conversion and display as a wstring (21.3, 22.4.1.4).

18.8.3 Abnormal termination

18.8.3.1 Type terminate_handler

typedef void (*terminate_handler)();

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().

18.8.3.2 set_terminate

terminate_handler set_terminate(terminate_handler f) noexcept;

Effects: Establishes the function designated by f as the current handler function for terminating exception processing.

Remarks: It is unspecified whether a null pointer value designates the default terminate_handler.

Returns: The previous terminate_handler.

18.8.3.3 get_terminate

terminate_handler get_terminate() noexcept;

Returns: The current terminate_handler. [Note: This may be a null pointer value. —end note]

§ 18.8.3.3
18.8.3.4 terminate
[[noreturn]] void terminate() noexcept;

Remarks: Called by the implementation when exception handling must be abandoned for any of several reasons (15.5.1), in effect immediately after throwing the exception. May also be called directly by the program.

Effects: Calls the current terminate_handler function. [Note: A default terminate_handler is always considered a callable handler in this context. — end note]

18.8.4 uncaught_exceptions
int uncaught_exceptions() noexcept;

Returns: The number of uncaught exceptions (15.5.3).

Remarks: When uncaught_exceptions() > 0, throwing an exception can result in a call of std::terminate() (15.5.1).

18.8.5 Exception propagation

typedef unspecified exception_ptr;

The type exception_ptr can be used to refer to an exception object.

exception_ptr shall satisfy the requirements of NullablePointer (17.6.3.3).

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]

exception_ptr current_exception() noexcept;

Returns: An exception_ptr object that refers to the currently handled exception (15.3) 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]

[[noreturn]] void rethrow_exception(exception_ptr p);
Requires: \( p \) shall not be a null pointer.

Throws: the exception object to which \( p \) refers.

\[
\text{template<class E> exception_ptr make_exception_ptr(E e) noexcept;}
\]

Effects: Creates an exception_ptr object that refers to a copy of \( e \), as if

\[
\begin{align*}
\text{try} & \{ \\
\text{throw } e; \\
\text{catch(...)} & \{ \\
\text{return current_exception();} \\
\}
\end{align*}
\]

[Note: This function is provided for convenience and efficiency reasons. —end note]

18.8.6 nested_exception

namespace std {
class nested_exception {
public:
  nested_exception() noexcept;
  nested_exception(const nested_exception&) noexcept = default;
  nested_exception& operator=(const nested_exception&) noexcept = default;
  virtual ~nested_exception() = default;

  // access functions
  [[noreturn]] void rethrow_nested() const;
  exception_ptr nested_ptr() const noexcept;
};

template<class T> [[noreturn]] void throw_with_nested(T&& t);
template <class E> void rethrow_if_nested(const E& e);
}

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]

nested_exception() noexcept;

3 Effects: The constructor calls current_exception() and stores the returned value.

[[noreturn]] void rethrow_nested() const;

4 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;

5 Returns: The stored exception captured by this nested_exception object.

template <class T> [[noreturn]] void throw_with_nested(T&& t);

6 Let \( U \) be remove_reference_t<T>.

7 Requires: \( U \) shall be CopyConstructible.

8 Throws: if \( U \) is a non-union class type not derived from nested_exception, 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 the dynamic type of e is publicly and unambiguously derived from nested_exception, calls
dynamic_cast<const nested_exception&>(e).rethrow_nested().

18.9 Initializer lists [support.initlist]

The header <initializer_list> defines a class template and several support functions related to list-
initialization (see 8.5.4).

Header <initializer_list> synopsis

namespace std {
  template<class E> class initializer_list {
    public:
      typedef E value_type;
      typedef const E& reference;
      typedef const E& const_reference;
      typedef size_t size_type;
      typedef const E* iterator;
      typedef const E* const_iterator;

      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
  };

  // 18.9.3 initializer list range access
  template<class E> constexpr const E* begin(initializer_list<E> il) noexcept;
  template<class E> constexpr const E* end(initializer_list<E> il) noexcept;
};

An object of type initializer_list<E> provides access to an array of objects of type const E. [Note: A pair of pointers or a pointer plus a length would be obvious representations for initializer_list. initializer_list is used to implement initializer lists as specified in 8.5.4. 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.

18.9.1 Initializer list constructors [support.initlist.cons]

constexpr initializer_list() noexcept;

Effects: constructs an empty initializer_list object.

Postcondition: size() == 0

18.9.2 Initializer list access [support.initlist.access]

constexpr const E* begin() const noexcept;

Returns: A pointer to the beginning of the array. If size() == 0 the values of begin() and end() are unspecified but they shall be identical.

constexpr const E* end() const noexcept;
\begin{verbatim}
2    Returns: begin() + size()

constexpr size_t size() const noexcept;

3    Returns: The number of elements in the array.

4    Complexity: Constant time.

18.9.3 Initializer list range access [support.initlist.range]

template<class E>constexpr const E* begin(initializer_list<E> il) noexcept;

1    Returns: il.begin().

template<class E>constexpr const E* end(initializer_list<E> il) noexcept;

2    Returns: il.end().

18.10 Other runtime support [support.runtime]

1    Headers <csetjmp> (nonlocal jumps), <csignal> (signal handling), <cstdalign> (alignment), <cstdarg>
    (variable arguments), <cstdbool> (__bool_true_false_are_defined), <cstdlib> (runtime environment
getenv(), system()), and <ctime> (system clock clock(), time()) provide further compatibility with C
    code.

2    The contents of these headers are the same as the Standard C library headers <setjmp.h>, <signal.h>,
    <stdalign.h>, <stdarg.h>, <stdbool.h>, <stdlib.h>, and <time.h>, respectively, with the following
    changes:

3    The restrictions that ISO C places on the second parameter to the va_start() macro in header <stdarg.h>
    are different in this International Standard. The parameter parmN is the identifier of the rightmost parameter
    in the variable parameter list of the function definition (the one just before the ...).227 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 4.8.1.1.

4    The function signature longjmp(jmp_buf jbuf, int val) has more restricted behavior in this International
    Standard. 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 7.10.4, 7.8, 7.6, 7.12.

5    Calls to the function getenv shall not introduce a data race (17.6.5.9) provided that nothing modifies the
    environment. [ Note: Calls to the POSIX functions setenv and putenv modify the environment. — end
    note ]

6    A call to the setlocale function may introduce a data race with other calls to the setlocale function or
    with calls to functions that are affected by the current C locale. The implementation shall behave as if no
    library function other than locale::global() calls the setlocale function.

7    The header <cstdalign> and the header <stdalign.h> shall not define a macro named alignas.

8    The header <cstdbool> and the header <stdbool.h> shall not define macros bool, true, or false.

9    A call to the function signal synchronizes with any resulting invocation of the signal handler so installed.

10   The common subset of the C and C++ languages consists of all declarations, definitions, and expressions
    that may appear in a well formed C++ program and also in a conforming C program. A POF (“plain old
    function”) is a function that uses only features from this common subset, and that does not directly or

227) Note that va_start is required to work as specified even if unary operator& is overloaded for the type of parmN.
\end{verbatim}
indirectly use any function that is not a POF, except that it may use plain lock-free atomic operations. A plain lock-free atomic operation is an invocation of a function \( f \) from Clause 29, such that \( f \) is not a member function, and either \( f \) is the function \text{atomic\_is\_lock\_free}, or for every atomic argument \( A \) passed to \( f \), \( \text{atomic\_is\_lock\_free}(A) \) yields true. All signal handlers shall have C linkage. The behavior of any function other than a POF used as a signal handler in a C++ program is implementation-defined.\(^{228}\)

Table 34 — Header \texttt{<csetjmp> synopsis}

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro</td>
<td>setjmp</td>
</tr>
<tr>
<td>Type</td>
<td>jmp_buf</td>
</tr>
<tr>
<td>Function</td>
<td>longjmp</td>
</tr>
</tbody>
</table>

Table 35 — Header \texttt{<csignal> synopsis}

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros</td>
<td>SIGABRT</td>
</tr>
<tr>
<td></td>
<td>SIGILL</td>
</tr>
<tr>
<td></td>
<td>SIGSEGV</td>
</tr>
<tr>
<td></td>
<td>SIG_DFL</td>
</tr>
<tr>
<td></td>
<td>SIG_IGN</td>
</tr>
<tr>
<td></td>
<td>SIGFPE</td>
</tr>
<tr>
<td></td>
<td>SIGINT</td>
</tr>
<tr>
<td></td>
<td>SIGTERM</td>
</tr>
<tr>
<td></td>
<td>SIG_ERR</td>
</tr>
<tr>
<td>Type</td>
<td>sig_atomic_t</td>
</tr>
<tr>
<td>Functions</td>
<td>raise</td>
</tr>
<tr>
<td></td>
<td>signal</td>
</tr>
</tbody>
</table>

Table 36 — Header \texttt{<cstdalign> synopsis}

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro</td>
<td>__alignas_is_defined</td>
</tr>
</tbody>
</table>

Table 37 — Header \texttt{<cstdarg> synopsis}

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros</td>
<td>va_arg</td>
</tr>
<tr>
<td></td>
<td>va_end</td>
</tr>
<tr>
<td></td>
<td>va_start</td>
</tr>
<tr>
<td></td>
<td>va_copy</td>
</tr>
<tr>
<td>Type</td>
<td>va_list</td>
</tr>
</tbody>
</table>

Table 38 — Header \texttt{<cstdbool> synopsis}

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro</td>
<td>__bool_true_false_are_defined</td>
</tr>
</tbody>
</table>

Table 39 — Header \texttt{<cstdlib> synopsis}

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
<td>getenv</td>
</tr>
<tr>
<td></td>
<td>system</td>
</tr>
</tbody>
</table>

\(^{228}\) In particular, a signal handler using exception handling is very likely to have problems. Also, invoking \texttt{std::exit} may cause destruction of objects, including those of the standard library implementation, which, in general, yields undefined behavior in a signal handler (see 1.9).
Table 40 — Header `<ctime>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro:</td>
<td>CLOCK_S_PER_SEC</td>
</tr>
<tr>
<td>Type:</td>
<td>clock_t</td>
</tr>
<tr>
<td>Function:</td>
<td>clock</td>
</tr>
</tbody>
</table>
19 Diagnostics library

19.1 General

This Clause describes components that C++ programs may use to detect and report error conditions.

The following subclauses describe components for reporting several kinds of exceptional conditions, documenting program assertions, and a global variable for error number codes, as summarized in Table 41.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.2 Exception classes</td>
<td>&lt;stdexcept&gt;</td>
</tr>
<tr>
<td>19.3 Assertions</td>
<td>&lt;cassert&gt;</td>
</tr>
<tr>
<td>19.4 Error numbers</td>
<td>&lt;cerrno&gt;</td>
</tr>
<tr>
<td>19.5 System error support</td>
<td>&lt;system_error&gt;</td>
</tr>
</tbody>
</table>

19.2 Exception classes

The Standard C++ library provides classes to be used to report certain errors (17.6.5.12) 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.

Header <stdexcept> synopsis

```cpp
namespace std {
    class logic_error;
    class domain_error;
    class invalid_argument;
    class length_error;
    class out_of_range;
    class runtime_error;
    class range_error;
    class overflow_error;
    class underflow_error;
}
```

19.2.1 Class logic_error

```cpp
namespace std {
    class logic_error : public exception {
        public:
            explicit logic_error(const string& what_arg);
            explicit logic_error(const char* what_arg);
    };
}
```

§ 19.2.1
The class `logic_error` defines the type of objects thrown as exceptions to report errors presumably detectable before the program executes, such as violations of logical preconditions or class invariants.

```cpp
logic_error(const string& what_arg);

Effects: Constructs an object of class `logic_error`.
Postcondition: strcmp(what(), what_arg.c_str()) == 0.
```

```cpp
logic_error(const char* what_arg);

Effects: Constructs an object of class `logic_error`.
Postcondition: strcmp(what(), what_arg) == 0.
```

### 19.2.2 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.

```cpp
domain_error(const string& what_arg);

Effects: Constructs an object of class `domain_error`.
Postcondition: strcmp(what(), what_arg.c_str()) == 0.
```

```cpp
domain_error(const char* what_arg);

Effects: Constructs an object of class `domain_error`.
Postcondition: strcmp(what(), what_arg) == 0.
```

### 19.2.3 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.

```cpp
invalid_argument(const string& what_arg);

Effects: Constructs an object of class `invalid_argument`.
Postcondition: strcmp(what(), what_arg.c_str()) == 0.
```

```cpp
invalid_argument(const char* what_arg);

Effects: Constructs an object of class `invalid_argument`.
Postcondition: strcmp(what(), what_arg) == 0.
```
19.2.4 Class \texttt{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);
    }
}
```
1 The class \texttt{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.

```cpp
length_error(const string& what_arg);
```
2 \textit{Effects:} Constructs an object of class \texttt{length\_error}.

```cpp
Postcondition: strcmp(what(), what_arg.c_str()) == 0.
```

```cpp
length_error(const char* what_arg);
```
4 \textit{Effects:} Constructs an object of class \texttt{length\_error}.

```cpp
Postcondition: strcmp(what(), what_arg) == 0.
```

19.2.5 Class \texttt{out\_of\_range}

```cpp
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 \texttt{out\_of\_range} defines the type of objects thrown as exceptions to report an argument value not in its expected range.

```cpp
out_of_range(const string& what_arg);
```
2 \textit{Effects:} Constructs an object of class \texttt{out\_of\_range}.

```cpp
Postcondition: strcmp(what(), what_arg.c_str()) == 0.
```

```cpp
out_of_range(const char* what_arg);
```
4 \textit{Effects:} Constructs an object of class \texttt{out\_of\_range}.

```cpp
Postcondition: strcmp(what(), what_arg) == 0.
```

19.2.6 Class \texttt{runtime\_error}

```cpp
namespace std {
    class runtime_error : public exception {
    public:
        explicit runtime_error(const string& what_arg);
        explicit runtime_error(const char* what_arg);
    }
}
```
1 The class \texttt{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);  
   Effects: Constructs an object of class runtime_error.
3   Postcondition: strcmp(what(), what_arg.c_str()) == 0.

runtime_error(const char* what_arg);  
   Effects: Constructs an object of class runtime_error.
5   Postcondition: strcmp(what(), what_arg) == 0.

19.2.7 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);  
   Effects: Constructs an object of class range_error.
3   Postcondition: strcmp(what(), what_arg.c_str()) == 0.

range_error(const char* what_arg);  
   Effects: Constructs an object of class range_error.
5   Postcondition: strcmp(what(), what_arg) == 0.

19.2.8 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);
   };
}

1   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.
3   Postcondition: strcmp(what(), what_arg.c_str()) == 0.

overflow_error(const char* what_arg);  
   Effects: Constructs an object of class overflow_error.
5   Postcondition: strcmp(what(), what_arg) == 0.
19.2.9 Class underflow_error

namespace std {
    class underflow_error : public runtime_error {
    public:
        explicit underflow_error(const string& what_arg);
        explicit underflow_error(const char* what_arg);
    };
};

1 The class underflow_error defines the type of objects thrown as exceptions to report an arithmetic underflow error.

underflow_error(const string& what_arg);

Effects: Constructs an object of class underflow_error.

Postcondition: strcmp(what(), what_arg.c_str()) == 0.

underflow_error(const char* what_arg);

Effects: Constructs an object of class underflow_error.

Postcondition: strcmp(what(), what_arg) == 0.

19.3 Assertions

The header <cassert>, described in (Table 42), provides a macro for documenting C++ program assertions and a mechanism for disabling the assertion checks.

Table 42 — Header <cassert> synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro</td>
<td>assert</td>
</tr>
</tbody>
</table>

2 The contents are the same as the Standard C library header <assert.h>.

See also: ISO C 7.2.

19.4 Error numbers

The header <cerrno> is described in Table 43. Its contents are the same as the POSIX header <errno.h>, except that errno shall be defined as a macro. [Note: The intent is to remain in close alignment with the POSIX standard. — end note] A separate errno value shall be provided for each thread.

19.5 System error support

This subclause describes components that the standard library and C++ programs may use to report error conditions originating from the operating system or other low-level application program interfaces.

Components described in this subclause shall not change the value of errno (19.4). Implementations should leave the error states provided by other libraries unchanged.

Header <system_error> synopsis

namespace std {
    class error_category;
    const error_category& generic_category() noexcept;
    const error_category& system_category() noexcept;

    class error_code;

§ 19.5
class error_condition;
class system_error;

template <class T>
struct is_error_code_enum : public false_type {};

template <class T>
struct is_error_condition_enum : public false_type {};

enum class errc {
    address_family_not_supported, // EAFNOSUPPORT
    address_in_use, // EADDRINUSE
    address_not_available, // EADDRNOTAVAIL
    already_connected, // EISCONN
    argument_list_too_long, // E2BIG
    argument_out_of_domain, // EDOM
    bad_address, //EFAULT
    bad_file_descriptor, // EBADF
    bad_message, // EBADMSG
    broken_pipe, // EPIPE
    connection_aborted, // ECONNABORTED
    connection_already_in_progress, // EALREADY
    connection_refused, // ECONNREFUSED
    connection_reset, // ECONNRESET
    cross_device_link, // EXDEV
    destination_address_required, // EDESTADDRREQ
    device_or_resource_busy, // EBUSY
    directory_not_empty, // ENOTEMPTY
    executable_format_error, // ENOSYS
    file_exists, // EXIST
    file_too_large, // EFBIG
    filename_too_long, // ENAMETOOLONG
    function_not_supported, // ENOSYS
    host_unreachable, // EHOSTUNREACH
    identifier_removed, // EIDRM

§ 19.5
illegal_byte_sequence,   // EILSEQ
inappropriate_io_control_operation, // ENOTTY
interrupted, // EINTR
invalid_argument, // EINVAL
invalid_seek, // ESPIPE
io_error, // EIO
is_a_directory, // EISDIR
message_size, // EMSGSIZE
network_down, // ENETDOWN
network_reset, // ENETRESET
network_unreachable, // ENETUNREACH
no_buffer_space, // ENOBUFS
no_child_process, // ECHILD
no_link, // ENOLINK
no_lock_available, // ENOLCK
no_message_available, // ENODATA
no_message, // ENOMSG
no_protocol_option, // ENOPROTOOPT
no_space_on_device, // ENOSPC
no_stream_resources, // ENOSR
no_such_device_or_address, // ENXIO
no_such_device, // ENODEV
no_such_file_or_directory, // ENOTDIR
no_such_process, // ENOTPD
not_a_directory, // ENOTDIR
not_a_socket, // ENOTSOCK
not_a_stream, // ENOSTR
not_connected, // ENOTCONN
not_enough_memory, // ENOMEM
not_supported, // ENOTSUP
operation_canceled, // ECANCELED
operation_in_progress, // EINPROGRESS
operation_not_permitted, // EPERM
operation_not_supported, // EOPNOTSUPP
operation_would_block, // EWOULDBLOCK
owner_dead, // EOWNERDEAD
permission_denied, // EACCES
protocol_error, // EPROTO
protocol_not_supported, // EPROTONOSUPPORT
read_only_file_system, // EROFS
resource_deadlock_would_occur, // EDEADLK
resource_unavailable_try_again, // EAGAIN
result_out_of_range, // ERANGE
state_not_recoverable, // ENOTRECOVERABLE
stream_timeout, // ETIME
text_file_busy, // ETXTBSY
timed_out, // ETIMEDOUT
too_many_files_open_in_system, // ENFILE
too_many_files, // 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 {}

error_code make_error_code(errc e) noexcept;
error_condition make_error_condition(errc e) noexcept;

// 19.5.4 Comparison operators:
bool operator==(const error_code& lhs, const error_code& rhs) noexcept;
bool operator==(const error_code& lhs, const error_condition& rhs) noexcept;
bool operator==(const error_condition& lhs, const error_code& rhs) noexcept;
bool operator==(const error_condition& lhs, const error_condition& rhs) noexcept;
bool operator!=(const error_code& lhs, const error_code& rhs) noexcept;
bool operator!=(const error_code& lhs, const error_condition& rhs) noexcept;
bool operator!=(const error_condition& lhs, const error_code& rhs) noexcept;
bool operator!=(const error_condition& lhs, const error_condition& rhs) noexcept;

// 19.5.5 Hash support
template <class T> struct hash;

template <> struct hash<error_code>;

} // namespace std

3 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.

4 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.

19.5.1 Class error_category [syserr.errcat]
19.5.1.1 Class error_category overview [syserr.errcat.overview]

The class error_category serves as a base class for types used to identify the source and encoding of a particular category of error code. Classes may be derived from error_category to support categories of errors in addition to those defined in this International Standard. Such classes shall behave as specified in this subclause. [Note: error_category objects are passed by reference, and two such objects are equal if they have the same address. This means that applications using custom error_category types should create a single object of each such type. — end note]

namespace std {

class error_category {

public:
constexpr error_category() noexcept;
virtual ~error_category();
error_category(const error_category&) = delete;
error_category& operator=(const error_category&) = delete;
virtual const char* name() const noexcept = 0;
v
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 noexcept;

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;
}
   // namespace std

19.5.1.2 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.

19.5.1.3 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 (20.9.6) provides a total ordering for pointers. — end note]

19.5.1.4 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;

§ 19.5.1.4
Returns: true if, for the category of error represented by *this, code is considered equivalent to condition; otherwise, false.

virtual bool equivalent(const error_code& code, int condition) const noexcept;

Returns: true if, for the category of error represented by *this, code is considered equivalent to condition; otherwise, false.

19.5.1.5 Error category objects [syserr.errcat.objects]

const error_category& generic_category() noexcept;

Returns: A reference to an object of a type derived from class error_category. All calls to this function shall return references to the same object.

Remarks: The object’s default_error_condition and equivalent virtual functions shall behave as specified for the class error_category. The object’s name virtual function shall return a pointer to the string "generic".

const error_category& system_category() noexcept;

Returns: A reference to an object of a type derived from class error_category. All calls to this function shall return references to the same object.

Remarks: The object’s equivalent virtual functions shall behave as specified for class error_category. The object’s name virtual function shall return a pointer to the string "system". The object’s default_error_condition virtual function shall behave as follows:

If the argument ev corresponds to a POSIX errno value posv, the function shall return error_condition(posv, generic_category()). Otherwise, the function shall return error_condition(ev, system_category()). What constitutes correspondence for any given operating system is unspecified. [Note: The number of potential system error codes is large and unbounded, and some may not correspond to any POSIX errno value. Thus implementations are given latitude in determining correspondence. — end note]

19.5.2 Class error_code [syserr.errcode]

19.5.2.1 Class error_code overview [syserr.errcode.overview]

The class error_code describes an object used to hold error code values, such as those originating from the operating system or other low-level application program interfaces. [Note: Class error_code is an adjunct to error reporting by exception. — end note]

namespace std {
    class error_code {
        public:
            // 19.5.2.2 constructors:
            error_code() noexcept;
            error_code(int val, const error_category& cat) noexcept;
            template <class ErrorCodeEnum>
                error_code(ErrorCodeEnum e) noexcept;

            // 19.5.2.3 modifiers:
            void assign(int val, const error_category& cat) noexcept;
            template <class ErrorCodeEnum>
                error_code& operator=(ErrorCodeEnum e) noexcept;
            void clear() noexcept;

            // 19.5.2.4 observers:
```cpp
int value() const noexcept;
const error_category& category() const noexcept;
error_condition default_error_condition() const noexcept;
string message() const;
explicit operator bool() const noexcept;

private:
    int val_;  // exposition only
    const error_category* cat_;  // exposition only
};
```

```cpp
// 19.5.2.5 non-member functions:
error_code make_error_code(errc e) noexcept;
bool operator<(const error_code& lhs, const error_code& rhs) noexcept;

template <class charT, class traits>
basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const error_code& ec);
}} // namespace std
```

### 19.5.2.2 Class error_code constructors

**error_code()** noexcept;

1. **Effects:** Constructs an object of type `error_code`.
2. **Postconditions:** `val_ == 0` and `cat_ == &system_category()`.

**error_code(int val, const error_category& cat) noexcept;**

3. **Effects:** Constructs an object of type `error_code`.
4. **Postconditions:** `val_ == val` and `cat_ == &cat`.

**template <class ErrorCodeEnum>**

```cpp
error_code(ErrorCodeEnum e) noexcept;
```

5. **Effects:** Constructs an object of type `error_code`.
6. **Postconditions:** `*this == make_error_code(e)`.
7. **Remarks:** This constructor shall not participate in overload resolution unless
   `is_error_code_enum<ErrorCodeEnum>::value` is `true`.

### 19.5.2.3 Class error_code modifiers

**void assign(int val, const error_category& cat) noexcept;**

1. **Postconditions:** `val_ == val` and `cat_ == &cat`.

**template <class ErrorCodeEnum>**

```cpp
error_code& operator=(ErrorCodeEnum e) noexcept;
```

2. **Postconditions:** `*this == make_error_code(e)`.
3. **Returns:** `*this`.
4. **Remarks:** This operator shall not participate in overload resolution unless
   `is_error_code_enum<ErrorCodeEnum>::value` is `true`.

**void clear() noexcept;**

5. **Postconditions:** `value() == 0` and `category() == system_category()`.

§ 19.5.2.3
19.5.2.4 Class \texttt{error\_code} observers

\begin{verbatim}
int value() const noexcept;
\end{verbatim}

\textit{Returns:} \texttt{val\_}.

const error_category& category() const noexcept;

\textit{Returns:} \texttt{cat\_}.

error_condition default_error_condition() const noexcept;

\textit{Returns:} \texttt{category().default\_error\_condition(value())}.

string message() const;

\textit{Returns:} \texttt{category().message(value())}.

explicit operator bool() const noexcept;

\textit{Returns:} \texttt{value() \(!= 0\).}

19.5.2.5 Class \texttt{error\_code} non-member functions

\begin{verbatim}
error_code make_error_code(errc e) noexcept;
\end{verbatim}

\textit{Returns:} \texttt{error\_code(static\_cast<int>(e), generic\_category())}.

\begin{verbatim}
bool operator<(const error_code& lhs, const error_code& rhs) noexcept;
\end{verbatim}

\textit{Returns:} \texttt{lhs.category() < rhs.category() || lhs.category() == rhs.category() \&\& lhs.value() < rhs.value()}.

\begin{verbatim}
template <class charT, class traits>
  basic_ostream<charT,traits>&
  operator<<(basic_ostream<charT,traits>& os, const error_code& ec);
\end{verbatim}

\textit{Effects:} \texttt{os \(<\ << ec\_category().name() \(<\ :\ :\ << ec\_value()}.\}

19.5.3 Class \texttt{error\_condition}

19.5.3.1 Class \texttt{error\_condition} overview

\begin{verbatim}
namespace std {
  class error_condition {
    public:
      // 19.5.3.2 constructors:
      error_condition() noexcept;
      error_condition(int val, const error_category& cat) noexcept;
      template <class ErrorConditionEnum>
        error_condition(ErrorConditionEnum e) noexcept;

      // 19.5.3.3 modifiers:
      void assign(int val, const error_category& cat) noexcept;
      template<class ErrorConditionEnum>
        error_condition& operator=(ErrorConditionEnum e) noexcept;
      void clear() noexcept;
  }
}
\end{verbatim}

§ 19.5.3.1
// 19.5.3.4 observers:
int value() const noexcept;
const error_category& category() const noexcept;
string message() const;
explicit operator bool() const noexcept;

private:
  int val_;          // exposition only
  const error_category* cat_; // exposition only
};

// 19.5.3.5 non-member functions:
bool operator<(const error_condition& lhs, const error_condition& rhs) noexcept;

} // namespace std

19.5.3.2 Class error_condition constructors [syserr.errcondition.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.
Postcondition: *this == make_error_condition(e).
Remarks: This constructor shall not participate in overload resolution unless
is_error_condition_enum<ErrorConditionEnum>::value is true.

19.5.3.3 Class error_condition modifiers [syserr.errcondition.modifiers]
void assign(int val, const error_category& cat) noexcept;
Postconditions: val_ == val and cat_ == &cat.

template <class ErrorConditionEnum>
error_condition& operator=(ErrorConditionEnum e) noexcept;
Postcondition: *this == make_error_condition(e).
Returns: *this.
Remarks: This operator shall not participate in overload resolution unless
is_error_condition_enum<ErrorConditionEnum>::value is true.

void clear() noexcept;
Postconditions: value() == 0 and category() == generic_category().

§ 19.5.3.3
19.5.3.4 Class error_condition observers

\texttt{int value() const noexcept;}

\textit{Returns:} \texttt{val_.}

\texttt{const error_category& category() const noexcept;}

\textit{Returns:} \texttt{*cat_}.

\texttt{string message() const;}

\textit{Returns:} \texttt{category().message(value())}.

\texttt{explicit operator bool() const noexcept;}

\textit{Returns:} \texttt{value() \textbar\textbar 0}.

19.5.3.5 Class error_condition non-member functions

\texttt{error_condition make_error_condition(errc e) noexcept;}

\textit{Returns:} \texttt{error_condition(static_cast<int>(e), generic_category())}.

\texttt{bool operator<(const error_condition& lhs, const error_condition& rhs) noexcept;}

\textit{Returns:} \texttt{lhs.category() \textbar\textbar rhs.category() || lhs.category() == rhs.category() \&\& \textbar\textbar
\texttt{lhs.value() < rhs.value()}}.

19.5.4 Comparison operators

\texttt{bool operator==(const error_code& lhs, const error_code& rhs) noexcept;}

\textit{Returns:} \texttt{lhs.category() == rhs.category() \&\& lhs.value() == rhs.value()}.  

\texttt{bool operator==(const error_code& lhs, const error_condition& rhs) noexcept;}

\textit{Returns:} \texttt{lhs.category().equivalent(lhs.value(), rhs) \&\& rhs.category().equivalent(rhs.value(),
\texttt{lhs.value()})}.  

\texttt{bool operator==(const error_condition& lhs, const error_code& rhs) noexcept;}

\textit{Returns:} \texttt{rhs.category().equivalent(rhs.value(), lhs) \&\& rhs.category().equivalent(lhs,
\texttt{rhs.value()})}.  

\texttt{bool operator==(const error_condition& lhs, const error_condition& rhs) noexcept;}

\textit{Returns:} \texttt{lhs.category() == rhs.category() \&\& lhs.value() == rhs.value()}.  

\texttt{bool operator!=(const error_code& lhs, const error_code& rhs) noexcept;}

\texttt{bool operator!=(const error_code& lhs, const error_condition& rhs) noexcept;}

\texttt{bool operator!=(const error_condition& lhs, const error_code& rhs) noexcept;}

\texttt{bool operator!=(const error_condition& lhs, const error_condition& rhs) noexcept;}

\textit{Returns:} \texttt{!(lhs == rhs)}.

19.5.5 System error hash support

\texttt{template <> struct hash<error_code>;}

\textit{The template specialization shall meet the requirements of class template hash (20.9.13)}.
19.5.6 Class system_error

19.5.6.1 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` 18.6.2.1 rather than `system_error`. — end note]

```cpp
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;
    };
}  // namespace std
```

19.5.6.2 Class system_error members

```cpp
system_error(error_code ec, const string& what_arg);

    Effects: Constructs an object of class `system_error`.
    Postconditions: code() == ec.
    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.
    string(what()).find(what_arg) != string::npos.

system_error(error_code ec);

    Effects: Constructs an object of class `system_error`.
    Postconditions: code() == ec.

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).
    string(what()).find(what_arg) != string::npos.
```

§ 19.5.6.2
Effects: Constructs an object of class `system_error`.

Postconditions: `code() == error_code(ev, ecat)`.  
`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;`  
Returns: An `ntbs` incorporating the arguments supplied in the constructor.

[Note: The returned `ntbs` might be the contents of `what_arg + ": " + code.message(). — end note]
20 General utilities library

20.1 General

This Clause describes utilities that are generally useful in C++ programs; some of these utilities are used by other elements of the C++ standard library. These utilities are summarized in Table 44.

Table 44 — General utilities library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.2</td>
<td>&lt;utility&gt;</td>
</tr>
<tr>
<td>20.3</td>
<td>&lt;utility&gt;</td>
</tr>
<tr>
<td>20.4</td>
<td>&lt;tuple&gt;</td>
</tr>
<tr>
<td>20.5</td>
<td>&lt;utility&gt;</td>
</tr>
<tr>
<td>20.6</td>
<td>&lt;bitset&gt;</td>
</tr>
<tr>
<td>20.7</td>
<td>&lt;memory&gt;</td>
</tr>
<tr>
<td>20.8</td>
<td>&lt;memory&gt;</td>
</tr>
<tr>
<td>20.9</td>
<td>&lt;functional&gt;</td>
</tr>
<tr>
<td>20.10</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>20.11</td>
<td>&lt;ratio&gt;</td>
</tr>
<tr>
<td>20.12</td>
<td>&lt;chrono&gt;</td>
</tr>
<tr>
<td>20.13</td>
<td>&lt;scoped_allocator&gt;</td>
</tr>
<tr>
<td>20.14</td>
<td>&lt;typeindex&gt;</td>
</tr>
</tbody>
</table>

20.2 Utility components

This subclause contains some basic function and class templates that are used throughout the rest of the library.

Header <utility> synopsis

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.

```cpp
#include <initializer_list>

namespace std {

   // 20.2.1, operators:
   namespace rel_ops {
      template<class T> bool operator!=(const T& a, const T& b); // `!=` operator
      template<class T> bool operator> (const T& a, const T& b);  // `>` operator
      template<class T> bool operator<=(const T& a, const T& b); // `<=` operator
      template<class T> bool operator>=(const T& a, const T& b); // `>=` operator
   }

   // 20.2.2, 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(noexcept(swap(*a, *b)));
```

§ 20.2 514
// 20.2.3, exchange:
template <class T, class U=⇒ T> T exchange(T& obj, U& new_val);

// 20.2.4, forward/move:
template <class T>
  constexpr T&& forward(remove_reference_t<T>& t) noexcept;
template <class T>
  constexpr T&& forward(remove_reference_t<T>&& t) noexcept;
template <class T>
  remove_reference_t<T>&& move(T&&) noexcept;
template <class T>
  conditional_t<
    !is_nothrow_move_constructible<T>::value && is_copy_constructible<T>::value,
    const T&, T&&> move_if_noexcept(T& x) noexcept;

// 20.2.5, declval:
template <class T>
  add_rvalue_reference_t<T> declval() noexcept; // as unevaluated operand

// 20.3, pairs:
template <class T1, class T2> struct pair;

// 20.3.3, 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)));  
template <class T1, class T2>
  constexpr see below make_pair(T1&&, T2&&);

// 20.3.4, tuple-like access to pair:
template <class T> class tuple_size;
template <size_t I, class T> class tuple_element;

template <class T1, class T2> struct tuple_size<pair<T1, T2>>(
  const pair<T1,T2>&, const pair<T1,T2>&);

template <class T1, class T2> struct tuple_element<0, pair<T1, T2>>(
  const pair<T1,T2>&, const pair<T1,T2>&);

template <class T1, class T2> struct tuple_element<1, pair<T1, T2>>(
  const pair<T1,T2>&, const pair<T1,T2>&);

template<size_t I, class T1, class T2>
  constexpr tuple_element_t<T1,T2> get(pair<T1, T2>&) noexcept;

template<size_t I, class T1, class T2>
  constexpr tuple_element_t<T1,T2> get(pair<T1, T2>&&)

template<
    size_t I, class T1, class T2>
    constexpr const tuple_element_t<I, pair<T1, T2>> &
    get(const pair<T1, T2>&) noexcept;

// 20.3.5, pair piecewise construction
struct piecewise_construct_t { };  // defined in <tuple>

// 20.5, Compile-time integer sequences
template<class T, T...> struct integer_sequence;

20.2.1 Operators

To avoid redundant definitions of operator!= out of operator== and operators >, <=, and >= out of operator<, the library provides the following:

```cpp
template <class T> bool operator!=(const T& x, const T& y);

Requires: Type T is EqualityComparable (Table 17).

Returns: !(x == y).
```

```cpp
template <class T> bool operator<(const T& x, const T& y);

Requires: Type T is LessThanComparable (Table 18).

Returns: y < x.
```

```cpp
template <class T> bool operator<=(const T& x, const T& y);

Requires: Type T is LessThanComparable (Table 18).

Returns: !(y < x).
```
Requires: Type T is LessThanComparable (Table 18).

Returns: !\(x < y\).

In this library, whenever a declaration is provided for an operator!\(!=\), operator\(>\), operator\(>\)=, or operator\(<\)=, and requirements and semantics are not explicitly provided, the requirements and semantics are as specified in this Clause.

20.2.2 swap

```
template<class T> void swap(T& a, T& b) noexcept(see below);
```

Remark: The expression inside noexcept is equivalent to:
```
is_nothrow_move_constructible<T>::value &&
is_nothrow_move_assignable<T>::value
```

Requires: Type T shall be MoveConstructible (Table 20) and MoveAssignable (Table 22).

Effects: Exchanges values stored in two locations.

```
template<class T, size_t N>
void swap(T (&a)[N], T (&b)[N]) noexcept(noexcept(swap(*a, *b)));
```

Requires: \(a[i]\) shall be swappable with (17.6.3.2) \(b[i]\) for all \(i\) in the range \([0,N)\).

Effects: swap_ranges(a, a + N, b)

20.2.3 exchange

```
template <class T, class U=T> T exchange(T& obj, U&& new_val);
```

Effects: Equivalent to:
```
T old_val = std::move(obj);
obj = std::forward<U>(new_val);
return old_val;
```

20.2.4 forward/move helpers

The library provides templated helper functions to simplify applying move semantics to an lvalue and to simplify the implementation of forwarding functions.

```
template <class T> constexpr T&& forward(remove_reference_t<T>& t) noexcept;
template <class T> constexpr T&& forward(remove_reference_t<T>&& t) noexcept;
```

Returns: \(\text{static
cast}\langle T\&\&\rangle(t)\).

Remark: If the second form is instantiated with an lvalue reference type, the program is ill-formed.

Example:
```
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)));
}
```

struct A {
    A(int&, const double&);
};

void g() {

\[ 20.2.4 \]
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: \texttt{static
cast<remove_reference_t<T>&&>(t)}.

Example:

template <class T, class A1>
shared_ptr<T> factory(A1&& a1) {
  return shared_ptr<T>(new T(std::forward<A1>(a1)));
}

struct A {
  A();
  A(const A&); // copies from lvalues
  A(A&&);     // moves from rvalues
};

void g() {
  A a;
  shared_ptr<A> sp1 = factory<A>(a); // “a” binds to A(const A&)
  shared_ptr<A> sp2 = 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<T>::value && is_copy_constructible<T>::value,
  const T&, T&&> move_if_noexcept(T& x) noexcept;

Returns: \texttt{std::move(x)}

\section*{20.2.5 \boldsymbol{Function template} declval}

The library provides the function template \texttt{declval} to simplify the definition of expressions which occur as unevaluated operands (Clause 5).

template <class T>
add_rvalue_reference_t<T> declval() noexcept; // as unevaluated operand

Remarks: If this function is odr-used (3.2), the program is ill-formed.

Remarks: The template parameter T of \texttt{declval} may be an incomplete type.

[Example:
template <class To, class From>
declaytype(static_cast<To>(::declval<From>()) convert(From&&);

declares a function template convert which only participates in overloading if the type From can be explicitly converted to type To. For another example see class template common_type (20.10.7.6).
—end example]

20.3 Pairs

20.3.1 In general

The library provides a template for heterogeneous pairs of values. The library also provides a matching function template to simplify their construction and several templates that provide access to pair objects as if they were tuple objects (see 20.4.2.5 and 20.4.2.6).

20.3.2 Class template pair

// defined in header <utility>

namespace std {
    template <class T1, class T2>
    struct pair {
        typedef T1 first_type;
        typedef T2 second_type;
        T1 first;
        T2 second;
        pair(const pair&) = default;
        pair(pair&&) = default;
        constexpr pair();
        constexpr pair(const T1& x, const T2& y);
        template<class U, class V> constexpr pair(U&& x, V&& y);
        template<class U, class V> constexpr pair(const pair<U, V>& p);
        template<class U, class V> constexpr pair(pair<U, V>&& 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 U, class V> pair& operator=(const pair<U, V>& p);
        pair& operator=(pair&& p) noexcept(see below);
        template <class U, class V> pair& operator=(pair<U, V>&& p);
    }
    void swap(pair& p) noexcept(see below);
};

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.

constexpr pair();

3 Requires: is_default_constructible<first_type>::value is true and is_default_constructible<second_type>::value is true.

§ 20.3.2 519
Effects: Value-initializes first and second.

```cpp
cconstexpr pair(const T1& x, const T2& y);
```

Requires: \( \text{is\_copy\_constructible}<\text{first\_type}>::\text{value}\) and \( \text{is\_copy\_constructible}<\text{second\_type}>::\text{value}\) are true.

Effects: The constructor initializes first with x and second with y.

```cpp
template<class U, class V> constexpr pair(U&& x, V&& y);
```

Requires: \( \text{is\_constructible}<\text{first\_type}, \text{U}\&>::\text{value}\) and \( \text{is\_constructible}<\text{second\_type}, \text{V}\&>::\text{value}\) are true.

Effects: The constructor initializes first with \( \text{std}::\text{forward}<\text{U}>(x)\) and second with \( \text{std}::\text{forward}<\text{V}>(y)\).

Remarks: If \( \text{U} \) is not implicitly convertible to \( \text{first\_type} \) or \( \text{V} \) is not implicitly convertible to \( \text{second\_type} \) this constructor shall not participate in overload resolution.

```cpp
template<class U, class V> constexpr pair(const pair<U, V>& p);
```

Requires: \( \text{is\_constructible}<\text{first\_type}, \text{const U}\&>::\text{value}\) and \( \text{is\_constructible}<\text{second\_type}, \text{const V}\&>::\text{value}\) are true.

Effects: Initializes members from the corresponding members of the argument.

Remark: This constructor shall not participate in overload resolution unless \( \text{const U}\& \) is implicitly convertible to \( \text{first\_type} \) and \( \text{const V}\& \) is implicitly convertible to \( \text{second\_type} \).

```cpp
template<class U, class V> constexpr pair(pair<U, V>&& p);
```

Requires: \( \text{is\_constructible}<\text{first\_type}, \text{U}\&&>::\text{value}\) and \( \text{is\_constructible}<\text{second\_type}, \text{V}\&&>::\text{value}\) are true.

Effects: The constructor initializes first with \( \text{std}::\text{forward}<\text{U}>(p.\text{first})\) and second with \( \text{std}::\text{forward}<\text{V}>(p.\text{second})\).

Remark: This constructor shall not participate in overload resolution unless \( \text{U} \) is implicitly convertible to \( \text{first\_type} \) and \( \text{V} \) is implicitly convertible to \( \text{second\_type} \).

```cpp
template<class... Args1, class... Args2>
pair(piecewise_construct_t,
    tuple<Args1...> first_args, tuple<Args2...> second_args);
```

Requires: \( \text{is\_constructible}<\text{first\_type}, \text{Args1}\&\&...>::\text{value}\) and \( \text{is\_constructible}<\text{second\_type}, \text{Args2}\&\&...>::\text{value}\) are true.

Effects: The constructor initializes first with arguments of types \( \text{Args1}\... \) obtained by forwarding the elements of \( \text{first\_args} \) and initializes second with arguments of types \( \text{Args2}\... \) obtained by forwarding the elements of \( \text{second\_args} \). (Here, forwarding an element \( x \) of type \( \text{U} \) within a \( \text{tuple} \) object means calling \( \text{std}::\text{forward}<\text{U}>(x)\).) This form of construction, whereby constructor arguments for first and second are each provided in a separate \( \text{tuple} \) object, is called \textit{piecewise construction}.

```cpp
pair& operator=(const pair& p);
```

Requires: \( \text{is\_copy\_assignable}<\text{first\_type}>::\text{value}\) and \( \text{is\_copy\_assignable}<\text{second\_type}>::\text{value}\) are true.

Effects: Assigns \( p.\text{first} \) to first and \( p.\text{second} \) to second.

Returns: \*this.
template<class U, class V> pair& operator=(const pair<U, V>& p);

   Requires: is_assignable<first_type&, const U&>::value is true and is_assignable<second_type&, const V&>::value is true.
   Effects: Assigns p.first to first and p.second to second.
   Returns: *this.

pair& operator=(pair&& p) noexcept(see below);

   Remarks: The expression inside noexcept is equivalent to:
   is_nothrow_move_assignable<T1>::value &&
   is_nothrow_move_assignable<T2>::value

   Requires: is_move_assignable<first_type>::value is true and is_move_assignable<second_type>::value is true.
   Effects: Assigns to first with std::forward<first_type>(p.first) and to second with std::forward<second_type>(p.second).
   Returns: *this.

template<class U, class V> pair& operator=(pair<U, V>&& p);

   Requires: is_assignable<first_type&, U&&>::value is true and is_assignable<second_type&,
   V&&>::value is true.
   Effects: Assigns to first with std::forward<U>(p.first) and to second with std::forward<V>(p.second).
   Returns: *this.

void swap(pair& p) noexcept(see below);

   Remarks: The expression inside noexcept is equivalent to:
   noexcept(swap(first, p.first)) &&
   noexcept(swap(second, p.second))

   Requires: first shall be swappable with (17.6.3.2) p.first and second shall be swappable with p.second.
   Effects: Swaps first with p.first and second with p.second.

20.3.3 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 == y)


```cpp
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: \( x \).swap\( y \)

```
template <class T1, class T2>
constexpr pair<V1, V2> make_pair(T1&& x, T2&& y);
```

Returns:

\[
\text{pair\<V1, V2\>(std::forward\<T1\>(x), std::forward\<T2\>(y))}\]

where \( V1 \) and \( V2 \) are determined as follows: Let \( U_i \) be \( \text{decay\_t<}T_i\text{> for each } T_i \). Then each \( V_i \) is \( X& \) if \( U_i \) equals \( \text{reference\_wrapper<}X\text{>} \), otherwise \( V_i \) is \( U_i \).

[Example: In place of:

```c++
return pair<int, double>(5, 3.1415926);
// explicit types
```

a C++ program may contain:

```c++
return make_pair(5, 3.1415926);
// types are deduced
```

—end example]

### 20.3.4 Tuple-like access to pair

```
template <class T1, class T2>
struct tuple_size<pair<T1, T2>>
  : integral_constant<size_t, 2> { };
```

```
tuple_element<0, pair<T1, T2>> :: type
```

Value: the type \( T_1 \).

```
tuple_element<1, pair<T1, T2>> :: type
```

Value: the type \( T_2 \).

```
template<size_t I, class T1, class T2>
constexpr tuple_element_t<I, pair<T1, T2>>&
  get(pair<T1, T2>& p) noexcept;
```

```
template<size_t I, class T1, class T2>
constexpr const tuple_element_t<I, pair<T1, T2>>&
  get(const pair<T1, T2>& p) noexcept;
```

Returns: If \( I == 0 \) returns \( p\).first; if \( I == 1 \) returns \( p\).second; otherwise the program is ill-formed.
template<
    size_t I, class T1, class T2>
constexpr tuple_element_t<I, pair<T1, T2>>&&
    get(pair<T1, T2>&& p) noexcept;

Returns: If I == 0 returns std::forward<T1&&>(p.first); if I == 1 returns std::forward<T2&&>(p.second); otherwise the program is ill-formed.

template <class T, class U>
constexpr T& get(pair<T, U>& p) noexcept;
template <class T, class U>
constexpr const T& get(const pair<T, U>& p) noexcept;

Requires: T and U are distinct types. Otherwise, the program is ill-formed.

Returns: get<0>(p);

template <class T, class U>
constexpr T& get(pair<T, U>&& p) noexcept;
template <class T, class U>
constexpr const T& get(const pair<T, U>&& p) noexcept;

Requires: T and U are distinct types. Otherwise, the program is ill-formed.

Returns: get<0>(std::move(p));

template <class T, class U>
constexpr T& get(pair<T, U>& p) noexcept;
template <class T, class U>
constexpr const T& get(const pair<T, U>& p) noexcept;

Requires: T and U are distinct types. Otherwise, the program is ill-formed.

Returns: get<1>(p);

template <class T, class U>
constexpr T& get(pair<U, T>& p) noexcept;
template <class T, class U>
constexpr const T& get(const pair<U, T>& p) noexcept;

Requires: T and U are distinct types. Otherwise, the program is ill-formed.

Returns: get<1>(std::move(p));

20.3.5 Piecewise construction

struct piecewise_construct_t { };constexpr piecewise_construct_t piecewise_construct{};

The struct piecewise_construct_t is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, pair has a constructor with piecewise_construct_t as the first argument, immediately followed by two tuple arguments used for piecewise construction of the elements of the pair object.

20.4 Tuples

20.4.1 In general

This subclause describes the tuple library that provides a tuple type as the class template tuple that can be instantiated with any number of arguments. Each template argument specifies the type of an element in the tuple. Consequently, tuples are heterogeneous, fixed-size collections of values. An instantiation of tuple with two arguments is similar to an instantiation of pair with the same two arguments. See 20.3.

Header <tuple> synopsis
namespace std {
  // 20.4.2, class template tuple:
  template <class... Types> class tuple;

  // 20.4.2.4, tuple creation functions:
  const unspecified ignore;

  template <class... Types>
  constexpr tuple<VTypes...> make_tuple(Types&&...);

  template <class... Types>
  constexpr tuple<Types&&...> forward_as_tuple(Types&&...) noexcept;

  template<class... Types>
  constexpr tuple<Types&...> tie(Types&...) noexcept;

  template<class... Tuples>
  constexpr tuple<Ctypes...> tuple_cat(Tuples&&...);

  // 20.4.2.5, tuple helper classes:
  template <class T> class tuple_size; // undefined
  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 tuple_element; // undefined
  template <size_t I, class T> class tuple_element<const T>;
  template <size_t I, class T> class tuple_element<volatile T>;
  template <size_t I, class T> class tuple_element<const volatile T>;
  template <size_t I, class... Types> class tuple_element<I, tuple<Types...> >;

  template <size_t I, class T>
  using tuple_element_t = typename tuple_element<I, T>::type;

  // 20.4.2.6, element access:
  template <size_t I, class... Types>
  constexpr tuple_element_t<I, tuple<Types...>>&
  get(tuple<Types&&...>&) noexcept;

  template <size_t I, class... Types>
  constexpr tuple_element_t<I, tuple<Types...>>&&
  get(tuple<Types&&...>&&) noexcept;

  template <size_t I, class... Types>
  constexpr const tuple_element_t<I, tuple<Types...>>&
  get(const tuple<Types&&...>&) noexcept;

  template <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;

  // 20.4.2.7, relational operators:
  template<class... TTypes, class... UTypes>
```cpp
constexpr bool operator==(const tuple<TTypes...>&, const tuple<UTypes...>&);
template<class... TTypes, class... UTypes>
constexpr bool operator<(const tuple<TTypes...>&, const tuple<UTypes...>&);
template<class... TTypes, class... UTypes>
constexpr bool operator!=(const tuple<TTypes...>&, const tuple<UTypes...>&);
template<class... TTypes, class... UTypes>
constexpr bool operator>(const tuple<TTypes...>&, const tuple<UTypes...>&);
template<class... TTypes, class... UTypes>
constexpr bool operator<=(const tuple<TTypes...>&, const tuple<UTypes...>&);
template<class... TTypes, class... UTypes>
constexpr bool operator>=(const tuple<TTypes...>&, const tuple<UTypes...>&);

// 20.4.2.8, allocator-related traits
template <class... Types, class Alloc>
struct uses_allocator<tuple<Types...>, Alloc>;

// 20.4.2.9, specialized algorithms:
template <class... Types>
void swap(tuple<Types...>& x, tuple<Types...>& y) noexcept((see below));
```

### 20.4.2 Class template `tuple`

```cpp	namespace std {
    template <class... Types>
    class tuple {
        public:

            // 20.4.2.1, tuple construction
            constexpr tuple();
            constexpr explicit tuple(const Types&...);
            template <class... UTypes>
            constexpr explicit tuple(UTypes&&...);
            tuple(const tuple&) = default;
            tuple(tuple&&) = default;
            template <class... UTypes>
            constexpr tuple(const tuple<UTypes...>&);
            template <class... UTypes>
            constexpr tuple(tuple<UTypes...>&&);

            template <class U1, class U2>
            constexpr tuple(const pair<U1, U2>&);   // only if sizeof...(Types) == 2
            template <class U1, class U2>
            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>
            tuple(allocator_arg_t, const Alloc& a, const Types&...);
            template <class Alloc, class... UTypes>
            tuple(allocator_arg_t, const Alloc& a, UTypes&&...);
            template <class Alloc>
```

§ 20.4.2
tuple(allocation_arg_t, const Alloc & a, const tuple &);
template <class Alloc>
tuple(allocation_arg_t, const Alloc & a, tuple&&);
template <class Alloc, class... UTypes>
tuple(allocation_arg_t, const Alloc & a, const tuple<UTypes...>&);
template <class Alloc, class... UTypes>
tuple(allocation_arg_t, const Alloc & a, tuple<UTypes...>&&);
template <class Alloc, class U1, class U2>
tuple(allocation_arg_t, const Alloc & a, const pair< U1, U2 >&);
template <class Alloc, class U1, class U2>
tuple(allocation_arg_t, const Alloc & a, pair< U1, U2>&&);

// 20.4.2.2, tuple assignment
tuple& operator=(const tuple&);
tuple& operator=(tuple&&) noexcept(see below);

template <class... UTypes>
tuple& operator=(const tuple< UTypes...>&);
tuple& operator=(tuple< UTypes...>&&);

template <class U1, class U2>
tuple& operator=(const pair< U1, U2 >&);
// only if sizeof...(Types) == 2
template <class U1, class U2>
tuple& operator=(pair< U1, U2>&&);
// only if sizeof...(Types) == 2

// 20.4.2.3, tuple swap
void swap(tuple&) noexcept(see below);
}

20.4.2.1 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 In the constructor descriptions that follow, let i be in the range \([0, \text{sizeof}...(\text{Types}))\) in order, \(T_i\) be the \(i^\text{th}\) type in Types, and \(U_i\) be the \(i^\text{th}\) type in a template parameter pack named UTypes, where indexing is zero-based.

constexpr tuple();

4 Requires: is_default_constructible<\(T_i\)>::value is true for all \(i\).

Effects: Value initializes each element.

castexpr explicit tuple(const Types&...);

5 Requires: is_copy_constructible<\(T_i\)>::value is true for all \(i\).

Effects: Initializes each element with the value of the corresponding parameter.

template <class... UTypes>
castexpr explicit tuple(UTypes&&... u);
Requires: sizeof...(Types) == sizeof...(UTypes). is_constructible<Ti, Ui&&>::value is true for all i.

Effects: Initializes the elements in the tuple with the corresponding value in std::forward<UTypes>(u).

Remark: This constructor shall not participate in overload resolution unless each type in UTypes is implicitly convertible to its corresponding type in Types.

tuple(const tuple& u) = default;

Requires: is_copy_constructible<Ti>::value is true for all i.

Effects: Initializes each element of *this with the corresponding element of u.

tuple(tuple&& u) = default;

Requires: is_move_constructible<Ti>::value is true for all i.

Effects: For all i, initializes the ith element of *this with std::forward<Ti>(get<i>(u)).

template <class... UTypes> constexpr tuple(const tuple<UTypes...>& u);

Requires: sizeof...(Types) == sizeof...(UTypes). is_constructible<Ti, const Ui&>::value is true for all i.

Effects: Constructs each element of *this with the corresponding element of u.

Remark: This constructor shall not participate in overload resolution unless const Ui& is implicitly convertible to Ti for all i.

template <class... UTypes> constexpr tuple(tuple<UTypes...>&& u);

Requires: sizeof...(Types) == sizeof...(UTypes). is_constructible<Ti, const Ui&>::value is true for all i.

Effects: For all i, initializes the ith element of *this with std::forward<Ui>(get<i>(u)).

Remark: This constructor shall not participate in overload resolution unless const Ui& is implicitly convertible to Ti for all i.

template <class U1, class U2> constexpr tuple(const pair<U1, U2>& u);

Requires: sizeof...(Types) == 2. is_constructible<T0, const U1&>::value is true for the first type T0 in Types and is_constructible<T1, const U2&>::value is true for the second type T1 in Types.

Effects: Constructs the first element with u.first and the second element with u.second.

Remark: This constructor shall not participate in overload resolution unless const U1& is implicitly convertible to T0 and const U2& is implicitly convertible to T1.

template <class U1, class U2> constexpr tuple(pair<U1, U2>&& u);

Requires: sizeof...(Types) == 2. is_constructible<T0, U1&&>::value is true for the first type T0 in Types and is_constructible<T1, U2&&>::value is true for the second type T1 in Types.

Effects: Initializes the first element with std::forward<U1>(u.first) and the second element with std::forward<U2>(u.second).

Remark: This constructor shall not participate in overload resolution unless U1 is implicitly convertible to T0 and U2 is implicitly convertible to T1.
template <class Alloc>
tuple(allocator_arg_t, const Alloc& a);

template <class Alloc>
tuple(allocator_arg_t, const Alloc& a, const Types&...);

template <class Alloc, class... UTypes>
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>
tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);

template <class Alloc, class... UTypes>
tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);

template <class Alloc, class U1, class U2>
tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);

template <class Alloc, class U1, class U2>
tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);

20.4.2.2 Assignment

For each tuple assignment operator, an exception is thrown only if the assignment of one of the types in Types throws an exception. In the function descriptions that follow, let \( i \) be in the range \([0,:0,sizeof...(Types))\) in order, \( T_i \) be the \( i \)th type in Types, and \( U_i \) be the \( i \)th type in a template parameter pack named UTypes, where indexing is zero-based. The expression inside noexcept is equivalent to the logical AND of the following expressions:

\[
\text{is_nothrow_move_assignable<}T_i\text{>::value}\text{ and }\text{is_assignable<}T_i\text{&, const }U_i\text{&>::value}
\]

where \( T_i \) is the \( i \)th type in Types.

1. Requires: is_nothrow_move_assignable<\( T_i \)>::value is true for all \( i \).
2. Effects: Assigns each element of \( u \) to the corresponding element of *this.
3. Returns: *this.
4. Remarks: The expression inside noexcept is equivalent to the logical AND of the following expressions:

\[
\text{is_nothrow_move_assignable<}T_i\text{>::value}\text{ and }\text{is_assignable<}T_i\text{&, const }U_i\text{&>::value}
\]

where \( T_i \) is the \( i \)th type in Types.

5. Requires: is_move_assignable<\( T_i \)>::value is true for all \( i \).
6. Effects: For all \( i \), assigns \( \text{std::forward<}T_i\text{>(get<}i\text{>(u))} \) to get<\( i \)>(*this).
7. Returns: *this.
8. Remarks: The expression inside noexcept is equivalent to the logical AND of the following expressions:

\[
\text{is_nothrow_move_assignable<}T_i\text{>::value}\text{ and }\text{is_assignable<}T_i\text{&, const }U_i\text{&>::value}
\]

where \( T_i \) is the \( i \)th type in Types.

9. Requires: sizeof...(Types) == sizeof...(UTypes) and is_assignable<\( T_i\)&, const \( U_i\) &>::value is true for all \( i \).
10. Effects: Assigns each element of \( u \) to the corresponding element of *this.
11. Returns: *this.
template <class... UTypes>
tuple& operator=(tuple<UTypes...>&& u);

Requires: is_assignable<Ti&, Ui&&>::value == true for all i.
sizeof...(Types) == sizeof...(UTypes).

Effects: For all i, assigns std::forward<Ui>(get<i>)(u)) to get<i>(*this).

Returns: *this.

template <class U1, class U2> tuple& operator=(const pair<U1, U2>& u);

Requires: sizeof...(Types) == 2.

Effects: Assigns u.first to the first element of *this and u.second to the second element of *this.

Returns: *this.

template <class U1, class U2> tuple& operator=(pair<U1, U2>&& u);

Requires: sizeof...(Types) == 2.

Effects: Assigns std::forward<U1>(u.first) to the first element of *this and 
std::forward<U2>(u.second) to the second element of *this.

Returns: *this.

20.4.2.3 swap

void swap(tuple& rhs) noexcept(see below);

Remark: The expression inside noexcept is equivalent to the logical AND of the following expressions:

noexcept(swap(declval<Ti&>(), declval<Ti&>()))

where Ti is the i-th type in Types.

Requires: Each element in *this shall be swappable with (17.6.3.2) the corresponding element in rhs.

Effects: Calls swap for each element in *this and its corresponding element in rhs.

Throws: Nothing unless one of the element-wise swap calls throws an exception.

20.4.2.4 Tuple creation functions

In the function descriptions that follow, let i be in the range [0, sizeof...(TTypes)) in order and let Ti be the i-th type in a template parameter pack named TTypes; let j be in the range [0, sizeof...(UTypes)) in order and Uj be the j-th type in a template parameter pack named UTypes, where indexing is zero-based.

template<class... Types>
constexpr tuple<VTypes...> make_tuple(Types&&... t);

Let Ui be decay_t<Ti> for each Ti in Types. Then each Vi in VTypes is X& if Ui equals reference_wrapper<X>, otherwise Vi is Ui.

Returns: tuple<VTypes...>(std::forward<Types>(t)...).

[Example:

int i; float j;
make_tuple(1, ref(i), cref(j))

creates a tuple of type

§ 20.4.2.4
template<class... Types>
constexpr tuple<Types&&...> forward_as_tuple(Types&&... 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<Types&&...>(std::forward<Types>(t)...)
template <size_t I, class... Types>
class tuple_element<I, tuple<Types...> > {
public:
    typedef TI type;
};

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. Then each of the three templates shall meet the UnaryTypeTrait requirements (20.10.1) with a BaseCharacteristic of integral_constant<size_t, TS::value>

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<I, T> of the cv-unqualified type T. Then each of the three templates shall meet the TransformationTrait requirements (20.10.1) with a member typedef that names the following type:

(5.1) for the first specialization, add_const_t<TE::type>,
(5.2) for the second specialization, add_volatile_t<TE::type>, and
(5.3) for the third specialization, add_cv_t<TE::type>.

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.

20.4.2.6 Element access [tuple.elem]

template <size_t I, class... Types>
constexpr tuple_element_t<I, tuple<Types...> >& get(tuple<Types...>& t) noexcept;

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.

template <size_t I, class... Types>
constexpr tuple_element_t<I, tuple<Types...>>&& get(tuple<Types...>&& t) noexcept;

Effects: Equivalent to return std::forward<typename tuple_element<I, tuple<Types...>> > ::type&&>(get<I>(t));

Note: 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&&.

template <size_t I, class... Types>
constexpr tuple_element_t<I, tuple<Types...>> const& get(const tuple<Types...>&& t) noexcept;

§ 20.4.2.6
Requires: \( I < \text{sizeof...(Types)} \). The program is ill-formed if \( I \) is out of bounds.

Returns: A const reference to the \( I \)th element of \( t \), where indexing is zero-based.

[Note: Constness is shallow. If a \( T \) in \( \text{Types} \) is some reference type \( X\& \), the return type is \( X\& \), not \( \text{const } X\& \). However, if the element type is non-reference type \( T \), the return type is \( \text{const } T\& \). This is consistent with how constness is defined to work for member variables of reference type. —end note]

```cpp
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;
```

Requires: The type \( T \) occurs exactly once in \( \text{Types} \). Otherwise, the program is ill-formed.

Returns: A reference to the element of \( t \) corresponding to the type \( T \) in \( \text{Types} \).

[Example:
```cpp
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 \( \text{get} \) is a nonmember function is that if this functionality had been provided as a member function, code where the type depended on a template parameter would have required using the \texttt{template} keyword. —end note]

20.4.2.7 Relational operators

```cpp
template<class... TTypes, class... UTypes>
constexpr bool operator==(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

Requires: For all \( i \), where \( 0 \leq i < \text{sizeof...(TTypes)} \), \( \text{get}<i>(t) == \text{get}<i>(u) \) is a valid expression returning a type that is convertible to \texttt{bool}. \( \text{sizeof...(TTypes)} == \text{sizeof...(UTypes)} \).

Returns: \texttt{true} if \( \text{get}<i>(t) == \text{get}<i>(u) \) for all \( i \), otherwise \texttt{false}. For any two zero-length tuples \( e \) and \( f \), \( e == f \) returns \texttt{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 \texttt{false}.

```cpp
template<class... TTypes, class... UTypes>
constexpr bool operator<(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

Requires: For all \( i \), where \( 0 \leq i < \text{sizeof...(TTypes)} \), \( \text{get}<i>(t) < \text{get}<i>(u) \) and \( \text{get}<i>(u) < \text{get}<i>(t) \) are valid expressions returning types that are convertible to \texttt{bool}. \( \text{sizeof...(TTypes)} == \text{sizeof...(UTypes)} \).

Returns: The result of a lexicographical comparison between \( t \) and \( u \). The result is defined as: \((\texttt{bool})(\text{get}<0>(t) < \text{get}<0>(u)) \) || (!\((\texttt{bool})(\text{get}<0>(u) < \text{get}<0>(t)) \)) \& \( t_{\text{tail}} < u_{\text{tail}} \), where \( r_{\text{tail}} \) for some tuple \( r \) is a tuple containing all but the first element of \( r \). For any two zero-length tuples \( e \) and \( f \), \( e < f \) returns \texttt{false}.

```cpp
template<class... TTypes, class... UTypes>
constexpr bool operator!=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

§ 20.4.2.7
template<class... TTypes, class... UTypes>
constexpr bool operator>(const tuple<TTypes...>& t, const tuple<UTypes...>& u);

Returns: !(t == u).

template<class... TTypes, class... UTypes>
constexpr bool operator<(const tuple<TTypes...>& t, const tuple<UTypes...>& u);

Returns: u < t.

template<class... TTypes, class... UTypes>
constexpr bool operator<=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);

Returns: !(u < t).

Returns: !(t < u).

[Note: The above definitions for comparison operators do not require t_tail (or u_tail) to be constructed. It may not even be possible, as t and u are not required to be copy constructible. Also, all comparison operators are short circuited; they do not perform element accesses beyond what is required to determine the result of the comparison. — end note]

20.4.2.8 Tuple traits

template <class... Types, class Alloc>
struct uses_allocator<tuple<Types...>, Alloc> : true_type { };  

Requires: Alloc shall be an Allocator (17.6.3.5).

[Note: Specialization of this trait informs other library components that tuple can be constructed with an allocator, even though it does not have a nested allocator_type. — end note]

20.4.2.9 Tuple specialized algorithms

template <class... Types>
void swap(tuple<Types...>& x, tuple<Types...>& y) noexcept(see below);

[Remark: The expression inside noexcept is equivalent to:
noexcept(x.swap(y))

Effects: x.swap(y)]

20.5 Compile-time integer sequences

20.5.1 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.

[Example:

```cpp
template<class F, class Tuple, std::size_t... I>
dectype(auto) apply_impl(F&& f, Tuple&& t, index_sequence<I...>) {
  return std::forward<F>(f)(std::get<0>(std::forward<Tuple>(t))...);
}
```

```cpp
template<class F, class Tuple>
dectype(auto) apply(F&& f, Tuple&& t) {
  using Indices = make_index_sequence<std::tuple_size<std::decay_t<Tuple>>::value>;
  return apply_impl(std::forward<F>(f), std::forward<Tuple>(t), Indices());
}
```

§ 20.5.1
20.5.2 Class template integer_sequence

```cpp
namespace std {
    template<class T, T... I>
    struct integer_sequence {
        typedef T value_type;
        static constexpr size_t size() noexcept { return sizeof...(I); }
    };
}
```

Note: The `index_sequence` alias template is provided for the common case of an integer sequence of type `size_t`. — end note

20.5.3 Alias template make_integer_sequence

```cpp
template<class T, T N>
using make_integer_sequence = integer_sequence<T,
```

Note: make_integer_sequence<int, 0> denotes the type `integer_sequence<int>`. — end note

20.6 Class template bitset

Header `<bitset>` synopsis

```cpp
#include <string>
#include <iosfwd>
// for istream, ostream
namespace std {
    template <size_t N> class bitset;

    // 20.6.4 bitset operators:
    template <size_t N>
    bitset<N> operator|(const bitset<N>&, const bitset<N>&) noexcept;
    template <size_t N>
    bitset<N> operator&(const bitset<N>&, const bitset<N>&) noexcept;
    template <size_t N>
    bitset<N> operator^(const bitset<N>&, const bitset<N>&) noexcept;
    template <class charT, class traits, size_t N>
    basic_istream<charT, traits>& operator>>(
        basic_istream<charT, traits>& is, bitset<N>& x);
    template <class charT, class traits, size_t N>
    basic_ostream<charT, traits>& operator<<(const bitset<N>& x, fls); 
}
```

The header `<bitset>` defines a class template and several related functions for representing and manipulating fixed-size sequences of bits.

```cpp
namespace std {
    template<size_t N> class bitset { 
        public:
        // bit reference:
        class reference {
            friend class bitset;
        }
    }
}
reference() noexcept;
public:
~reference() noexcept;
reference& operator=(bool x) noexcept;  // for b[i] = x;
reference& operator=(const reference&) noexcept;  // for b[i] = b[j];
bool operator¬() const noexcept;  // flips the bit
operator bool() const noexcept;  // for x = b[i];
reference& flip() noexcept;  // for b[i].flip();
};

// 20.6.1 constructors:
constexpr bitset() noexcept;
constexpr bitset(unsigned long long val) noexcept;
template<class charT, class traits, class Allocator>
explicit bitset(
    const basic_string<charT,traits,Allocator>& str,
    typename basic_string<charT,traits,Allocator>::size_type pos = 0,
    typename basic_string<charT,traits,Allocator>::size_type n =
    basic_string<charT,traits,Allocator>::npos,
    charT zero = charT('0'), charT one = charT('1'));

template <class charT>
explicit bitset(
    const charT* str,
    typename basic_string<charT>::size_type n = basic_string<charT>::npos,
    charT zero = charT('0'), charT one = charT('1'));

// 20.6.2 bitset operations:
bitset<N>& operator&=(const bitset<N>& rhs) noexcept;
bitset<N>& operator|=(const bitset<N>& rhs) noexcept;
bitset<N>& operator^=(const bitset<N>& rhs) noexcept;
bitset<N>& operator<<=(size_t pos) noexcept;
bitset<N>& operator>>=(size_t pos) noexcept;
bitset<N>& set() noexcept;
bitset<N>& set(size_t pos, bool val = true);
bitset<N>& reset() noexcept;
bitset<N>& reset(size_t pos);
bitset<N> operator¬() const noexcept;
bitset<N>& flip() noexcept;
bitset<N>& flip(size_t pos);

// element access:
constexpr bool operator[](size_t pos) const;  // for b[i];
reference operator[](size_t pos);  // for b[i];

unsigned long to_ulong() const;
unsigned long long to_ullong() const;
template <class charT = char,
    class traits = char_traits<charT>,
    class Allocator = allocator<charT> >
    basic_string<charT, traits, Allocator>
to_string(charT zero = charT('0'), charT one = charT('1')) const;
size_t count() const noexcept;
constexpr size_t size() const noexcept;
bool operator==(const bitset<N>& rhs) const noexcept;
bool operator!=(const bitset<N>& rhs) const noexcept;

§ 20.6 535
bool test(size_t pos) const;
bool all() const noexcept;
bool any() const noexcept;
bool none() const noexcept;
bitset<N> operator<<(size_t pos) const noexcept;
bitset<N> operator>>(size_t pos) const noexcept;

// 20.6.3 hash support
template <class T> struct hash;
template <size_t N> struct hash<bitset<N> >;

2 The class template bitset<N> describes an object that can store a sequence consisting of a fixed number of
bits, N.

3 Each bit represents either the value zero (reset) or one (set). To toggle a bit is to change the value zero to
one, or the value one to zero. Each bit has a non-negative position pos. When converting between an object
of class bitset<N> and a value of some integral type, bit position pos corresponds to the bit value 1 <<pos.
The integral value corresponding to two or more bits is the sum of their bit values.

4 The functions described in this subclause can report three kinds of errors, each associated with a distinct
exception:

   (4.1) — an invalid-argument error is associated with exceptions of type invalid_argument (19.2.3);
   (4.2) — an out-of-range error is associated with exceptions of type out_of_range (19.2.5);
   (4.3) — an overflow error is associated with exceptions of type overflow_error (19.2.8).

20.6.1 bitset constructors

constexpr bitset() noexcept;
Effects: Constructs an object of class bitset<N>, initializing all bits to zero.

cconstexpr bitset(unsigned long long val) noexcept;
Effects: Constructs an object of class bitset<N>, initializing the first M bit positions to the corresponding
bit values in val. M is the smaller of N and the number of bits in the value representation (3.9) of
unsigned long long. If M < N, the remaining bit positions are initialized to zero.

template <class charT, class traits, class Allocator>
explicit
bitset(const basic_string<charT, traits, Allocator>& str,
typename basic_string<charT, traits, Allocator>::size_type pos = 0,
typename basic_string<charT, traits, Allocator>::size_type n =
basic_string<charT, traits, Allocator>::npos,
charT zero = charT('0'), charT one = charT('1'));

Requires: pos <= str.size().

Throws: out_of_range if pos > str.size().

Effects: Determines the effective length rlen of the initializing string as the smaller of n and str.size() - pos.
The function then throws invalid_argument if any of the rlen characters in str beginning at position
pos is other than zero or one. The function uses traits::eq() to compare the character values.
Otherwise, the function constructs an object of class `bitset<N>`, initializing the first \( M \) bit positions to values determined from the corresponding characters in the string `str`. \( M \) is the smaller of \( N \) and `rlen`.

An element of the constructed object has value zero if the corresponding character in `str`, beginning at position `pos`, is `zero`. Otherwise, the element has the value one. Character position `pos + M - 1` corresponds to bit position zero. Subsequent decreasing character positions correspond to increasing bit positions.

If \( M < N \), remaining bit positions are initialized to zero.

```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)
```

### 20.6.2 `bitset` members

`bitset<N>& operator&=(const bitset<N>& rhs)` noexcept;

`Effects:` Clears each bit in `*this` for which the corresponding bit in `rhs` is clear, and leaves all other bits unchanged.

`Returns:` `*this`.

`bitset<N>& operator|=(const bitset<N>& rhs)` noexcept;

`Effects:` Sets each bit in `*this` for which the corresponding bit in `rhs` is set, and leaves all other bits unchanged.

`Returns:` `*this`.

`bitset<N>& operator^=(const bitset<N>& rhs)` noexcept;

`Effects:` Toggles each bit in `*this` for which the corresponding bit in `rhs` is set, and leaves all other bits unchanged.

`Returns:` `*this`.

`bitset<N>& operator<<=(size_t pos)` noexcept;

`Effects:` Replaces each bit at position \( I \) in `*this` with a value determined as follows:

\[\begin{align*}
    & (7.1) \quad \text{If} \ I < \text{pos}, \text{the new value is zero}; \\
    & (7.2) \quad \text{If} \ I \geq \text{pos}, \text{the new value is the previous value of the bit at position} \ I - \text{pos}.
\end{align*}\]

`Returns:` `*this`.

`bitset<N>& operator>>=(size_t pos)` noexcept;

`Effects:` Replaces each bit at position \( I \) in `*this` with a value determined as follows:

\[\begin{align*}
    & (9.1) \quad \text{If} \ \text{pos} \geq N - I, \text{the new value is zero}; \\
    & (9.2) \quad \text{If} \ \text{pos} < N - I, \text{the new value is the previous value of the bit at position} \ I + \text{pos}.
\end{align*}\]

§ 20.6.2
Returns: *this.

bitset<N>& set() noexcept;

    Effects: Sets all bits in *this.
    Returns: *this.

bitset<N>& set(size_t pos, bool val = true);

    Requires: pos is valid
    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 nonzero, the stored value is one, otherwise it is zero.
    Returns: *this.

bitset<N>& reset() noexcept;

    Effects: Resets all bits in *this.
    Returns: *this.

bitset<N>& reset(size_t pos);

    Requires: pos is valid
    Throws: out_of_range if pos does not correspond to a valid bit position.
    Effects: Resets the bit at position pos in *this.
    Returns: *this.

bitset<N> operator~() const noexcept;

    Effects: Constructs an object x of class bitset<N> and initializes it with *this.
    Returns: x.flip().

bitset<N>& flip() noexcept;

    Effects: Toggles all bits in *this.
    Returns: *this.

bitset<N>& flip(size_t pos);

    Requires: pos is valid
    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: \( x \).

```cpp
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.

```cpp
size_t count() const noexcept;
```

Returns: A count of the number of bits set in `*this`.

```cpp
constexpr size_t size() const noexcept;
```

Returns: \( N \).

```cpp
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 \( \text{rhs} \).

```cpp
bool operator!=(const bitset<N> & rhs) const noexcept;
```

Returns: `true` if \(!(*this == rhs)\).

```cpp
bool test(size_t pos) const;
```

Requires: `pos` is valid

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.

```cpp
bool all() const noexcept;
```

Returns: `count() == size()`

```cpp
bool any() const noexcept;
```

Returns: `count() != 0`

```cpp
bool none() const noexcept;
```

Returns: `count() == 0`

```cpp
bitset<N> operator<<(size_t pos) const noexcept;
```

Returns: `bitset<N>(*this) <<= pos`.

```cpp
bitset<N> operator>>(size_t pos) const noexcept;
```

Returns: `bitset<N>(*this) >>= pos`.

§ 20.6.2
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.

Remark: For the purpose of determining the presence of a data race (1.10), any access or update through the resulting reference potentially accesses or modifies, respectively, the entire underlying bitset.

20.6.3  bitset hash support

template <size_t N> struct hash<bitset<N> >;

The template specialization shall meet the requirements of class template hash (20.9.13).

20.6.4  bitset operators

bitset<N> operator&(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;

Returns: bitset<N>(lhs) &= rhs.

bitset<N> operator|(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;

Returns: bitset<N>(lhs) |= rhs.

bitset<N> operator^(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;

Returns: bitset<N>(lhs) ^= rhs.

template <class charT, class traits, size_t N>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& is, bitset<N>& x);

A formatted input function (27.7.2.2).

Effects: Extracts up to N characters from is. Stores these characters in a temporary object str of type basic_string<charT, traits>, then evaluates the expression x = bitset<N>(str). Characters are extracted and stored until any of the following occurs:

1. N characters have been extracted and stored;
2. End-of-file occurs on the input sequence;
3. The next input character is neither is.widen(‘0’) nor is.widen(‘1’) (in which case the input character is not extracted).

If no characters are stored in str, calls is.setstate(ios_base::failbit) (which may throw ios_base::failure (27.5.5.4)).

Returns: is.

template <class charT, class traits, size_t N>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);

§ 20.6.4
Returns:

```cpp
os << x.template to_string<charT,traits,allocator<charT>> (
    use_facet<ctype<charT>>(os.getloc()).widen('0'),
    use_facet<ctype<charT>>(os.getloc()).widen('1'))
```

(see 27.7.3.6).

## 20.7 Memory

### 20.7.1 In general

This subclause describes the contents of the header `<memory>` (20.7.2) and some of the contents of the C headers `<cstdlib>` and `<cstring>` (20.7.13).

### 20.7.2 Header `<memory>` synopsis

The header `<memory>` defines several types and function templates that describe properties of pointers and pointer-like types, manage memory for containers and other template types, and construct multiple objects in uninitialized memory buffers (20.7.3–20.7.12). The header also defines the templates `unique_ptr`, `shared_ptr`, `weak_ptr`, and various function templates that operate on objects of these types (20.8).

```cpp
namespace std {
    // 20.7.3, pointer traits
    template <class Ptr> struct pointer_traits;
    template <class T> struct pointer_traits<T*>;

    // 20.7.4, pointer safety
    enum class pointer_safety { relaxed, preferred, strict };  
    void declare_reachable(void* p);
    template <class T> T*undeclare_reachable(T* p);
    void declare_no_pointers(char* p, size_t n);
    void undeclare_no_pointers(char* p, size_t n);
    pointer_safety get_pointer_safety() noexcept;

    // 20.7.5, pointer alignment function
    void* align(std::size_t alignment, std::size_t size,
                void*& ptr, std::size_t& space);

    // 20.7.6, allocator argument tag
    struct allocator_arg_t { };  
    constexpr allocator_arg_t allocator_arg{};

    // 20.7.7, uses_allocator
    template <class T, class Alloc> struct uses_allocator;

    // 20.7.8, allocator traits
    template <class Alloc> struct allocator_traits;

    // 20.7.9, the default allocator:
    template <class T> class class_allocator;
    template <> class allocator<void>;
    template <class T, class U>
    bool operator==(const allocator<T>&, const allocator<U>&) noexcept;
    template <class T, class U>
    bool operator!=(const allocator<T>&, const allocator<U>&) noexcept;

    // 20.7.10, raw storage iterator:

    § 20.7.2
```
template <class OutputIterator, class T> class raw_storage_iterator;

// 20.7.11, temporary buffers:
template <class T>
    pair<T*,ptrdiff_t> get_temporary_buffer(ptrdiff_t n) noexcept;
template <class T>
    void return_temporary_buffer(T* p);

// 20.7.12, specialized algorithms:
template <class T> T* addressof(T& r) noexcept;
template <class InputIterator, class ForwardIterator>
    ForwardIterator uninitialized_copy(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 ForwardIterator, class T>
    void uninitialized_fill(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);

// 20.8.1 class template unique_ptr:
template <class T> struct default_delete;
template <class T> struct default_delete<T[]>;
template <class T, class D = default_delete<T>> class unique_ptr;
template <class T, class D> class unique_ptr<T[], D>;
template <class T, class... Args> unique_ptr<T> make_unique(Args&&... args);
template <class T> unique_ptr<T> make_unique(size_t n);
template <class T, class... Args> unspecified
    make_unique(Args&&...) = delete;
template <class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y) noexcept;
template <class T1, class D1, class T2, class D2>
    bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
    bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
    bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
    bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
    bool operator==(const unique_ptr<T1, D1>& x, nullptr_t) noexcept;
template <class T1, class D1, class T2, class D2>
    bool operator==(nullptr_t, const unique_ptr<T1, D1>& y) noexcept;
template <class T1, class D1, class T2, class D2>
    bool operator!=(const unique_ptr<T1, D1>& x, nullptr_t) noexcept;
template <class T1, class D1, class T2, class D2>
    bool operator!=(nullptr_t, const unique_ptr<T1, D1>& y) noexcept;
§ 20.7.2 542
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);

// 20.8.2.1, class bad_weak_ptr:
class bad_weak_ptr;

// 20.8.2.2, class template shared_ptr:
template<class T> class shared_ptr;

// 20.8.2.2.6, shared_ptr creation
template<class T, class... Args> shared_ptr<T> make_shared(Args&&... args);
template<class T, class A, class... Args>
  shared_ptr<T> allocate_shared(const A& a, Args&&... args);

// 20.8.2.2.7, shared_ptr comparisons:
template<class T, class U>
  bool operator==(shared_ptr<T> const& a, shared_ptr<U> const& b) noexcept;
template<class T, class U>
  bool operator!=(shared_ptr<T> const& a, shared_ptr<U> const& b) noexcept;
template<class T, class U>
  bool operator<(shared_ptr<T> const& a, shared_ptr<U> const& b) noexcept;
template<class T, class U>
  bool operator>(shared_ptr<T> const& a, shared_ptr<U> const& b) noexcept;
template<class T, class U>
  bool operator>=(shared_ptr<T> const& a, shared_ptr<U> const& b) noexcept;
template<class T, class U>
  bool operator<=(shared_ptr<T> const& a, shared_ptr<U> const& b) noexcept;

// 20.8.2.2.8, shared_ptr comparisons:
template<class T>
  bool operator==(const shared_ptr<T>& x, nullptr_t) noexcept;
template<class T>
  bool operator==(nullptr_t, const shared_ptr<T>& y) noexcept;
template<class T>
  bool operator!=(const shared_ptr<T>& x, nullptr_t) noexcept;
template<class T>
  bool operator!=(nullptr_t, const shared_ptr<T>& y) noexcept;
template<class T>
  bool operator<(const shared_ptr<T>& x, nullptr_t) noexcept;
template<class T>
  bool operator<(nullptr_t, const shared_ptr<T>& y) noexcept;

§ 20.7.2
bool operator<=(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator<=(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator>(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator>(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator>=(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator>=(nullptr_t, const shared_ptr<T>& y) noexcept;

// 20.8.2.2.8, shared_ptr specialized algorithms:
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;

// 20.8.2.2.9, shared_ptr casts:
template<class T, class U>
  shared_ptr<T> static_pointer_cast(shared_ptr<U> const& r) noexcept;
template<class T, class U>
  shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const& r) noexcept;
template<class T, class U>
  shared_ptr<T> const_pointer_cast(shared_ptr<U> const& r) noexcept;

// 20.8.2.2.10, shared_ptr get_deleter:
template<class D, class T> D* get_deleter(shared_ptr<T> const& p) noexcept;

// 20.8.2.2.11, shared_ptr I/O:
template<class E, class T, class Y>
  basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, shared_ptr<Y> const& p);

// 20.8.2.3, class template weak_ptr:
template<class T> class weak_ptr;

// 20.8.2.3.6, weak_ptr specialized algorithms:
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;

// 20.8.2.4, class template owner_less:
template<class T> class owner_less;

// 20.8.2.5, class template enable_shared_from_this:
template<class T> class enable_shared_from_this;

// 20.8.2.6, shared_ptr atomic access:
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);

// 20.8.2.7 hash support

} // 72x397

20.7.3 Pointer traits [pointer.traits] 1

The class template pointer_traits supplies a uniform interface to certain attributes of pointer-like types.

namespace std {

    template <class Ptr> struct pointer_traits {
        typedef Ptr pointer;
        typedef see below element_type;
        typedef see below difference_type;

        template <class U> using rebind = see below;

        static pointer pointer_to(see below r);
    };

    template <class T> struct pointer_traits<T*> {
        typedef T* pointer;
        typedef T element_type;
        typedef ptrdiff_t difference_type;

        template <class U> using rebind = U*;

        static pointer pointer_to(see below r) noexcept;
    };

} // std

20.7.3.1 Pointer traits member types [pointer.traits.types]
typedef see below element_type;

Type: 

1. Type: $\text{Ptr}$::$\text{element_type}$ if the qualified-id $\text{Ptr}$::$\text{element_type}$ is valid and denotes a type (14.8.2); otherwise, $T$ if $\text{Ptr}$ is a class template instantiation of the form $\text{SomePointer}<T, \text{Args}>$, where $\text{Args}$ is zero or more type arguments; otherwise, the specialization is ill-formed.

typedef see below difference_type;

2. Type: $\text{Ptr}$::$\text{difference_type}$ if the qualified-id $\text{Ptr}$::$\text{difference_type}$ is valid and denotes a type (14.8.2); otherwise, $\text{std}::\text{ptrdiff_t}$.

template <class U> using rebind = see below;

3. Alias template: $\text{Ptr}$::$\text{rebind}<U>$ if the qualified-id $\text{Ptr}$::$\text{rebind}<U>$ is valid and denotes a type (14.8.2); otherwise, $\text{SomePointer}<U, \text{Args}>$ if $\text{Ptr}$ is a class template instantiation of the form $\text{SomePointer}<T, \text{Args}>$, where $\text{Args}$ is zero or more type arguments; otherwise, the instantiation of $\text{rebind}$ is ill-formed.

20.7.3.2 Pointer traits member functions

static pointer $\text{pointer_traits}::\text{pointer_to}(\text{see below } r)$;
static pointer $\text{pointer_traits}<\text{T}*>::\text{pointer_to}(\text{see below } r)$ noexcept;

1. Remark: If $\text{element_type}$ is (possibly cv-qualified) $\text{void}$, the type of $r$ is unspecified; otherwise, it is $\text{element_type}$. 

2. 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 $\text{std}::\text{addressof}(r)$.

20.7.4 Pointer safety

A complete object is 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.

void $\text{declare_reachable}(\text{void}^* \, p)$;

1. Requires: $p$ shall be a safely-derived pointer (3.7.4.3) or a null pointer value.

2. Effects: If $p$ is not null, the complete object referenced by $p$ is subsequently declared reachable (3.7.4.3).

3. Throws: May throw $\text{std}::\text{bad_alloc}$ if the system cannot allocate additional memory that may be required to track objects declared reachable.

template <class T> T* $\text{undeclare_reachable}(T^* \, p)$;

1. Requires: If $p$ is not null, the complete object referenced by $p$ shall have been previously declared reachable, and shall be live (3.8) from the time of the call until the last $\text{undeclare_reachable}(p)$ call on the object.

2. Returns: A safely derived copy of $p$ which shall compare equal to $p$.


4. \[ Note: It is expected that calls to $\text{declare_reachable}(p)$ will consume a small amount of memory in addition to that occupied by the referenced object until the matching call to $\text{undeclare_reachable}(p)$ is encountered. Long running programs should arrange that calls are matched. — end note \]

void $\text{declare_no_pointers}(\text{char}^* \, p, \text{size_t} \, n)$;
9. **Requires:** No bytes in the specified range are currently registered with `declare_no_pointers()`. If the specified range is in an allocated object, then it must be entirely within a single allocated object. The object must be live until the corresponding `undeclare_no_pointers()` call. [Note: In a garbage-collecting implementation, the fact that a region in an object is registered with `declare_no_pointers()` should not prevent the object from being collected. — end note]

10. **Effects:** The n bytes starting at p no longer contain traceable pointer locations, independent of their type. Hence indirection through a pointer located there is undefined if the object it points to was created by global `operator new` and not previously declared reachable. [Note: This may be used to inform a garbage collector or leak detector that this region of memory need not be traced. — end note]

11. **Throws:** Nothing.

12. [Note: Under some conditions implementations may need to allocate memory. However, the request can be ignored if memory allocation fails. — end note]

   ```
   void undeclare_no_pointers(char* p, size_t n);
   ```

13. **Requires:** The same range must previously have been passed to `declare_no_pointers()`.

14. **Effects:** Unregisters a range registered with `declare_no_pointers()` for destruction. It must be called before the lifetime of the object ends.

15. **Throws:** Nothing.

16. `pointer_safety get_pointer_safety() noexcept;`

   **Returns:** `pointer_safety::strict` if the implementation has strict pointer safety (3.7.4.3). It is implementation defined whether `get_pointer_safety` returns `pointer_safety::relaxed` or `pointer_safety::preferred` if the implementation has relaxed pointer safety.\(^\text{229}\)

---

20.7.5 Align

```
void* align(std::size_t alignment, std::size_t size, void*& ptr, std::size_t& space);
```

1. **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 point to the first possible address of such storage and decreases `space` by the number of bytes used for alignment. Otherwise, the function does nothing.

2. **Requires:**
   - `alignment` shall be a power of two
   - `ptr` shall point to contiguous storage of at least `space` bytes

3. **Returns:** A null pointer if the requested aligned buffer would not fit into the available space, otherwise the adjusted value of `ptr`.

4. [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]

---

20.7.6 Allocator argument tag

```
namespace std {
    struct allocator_arg_t { }
    constexpr allocator_arg_t allocator_arg{};
}
```

\(^\text{229}\) `pointer_safety::preferred` might be returned to indicate that a leak detector is running so that the program can avoid spurious leak reports.
The allocator_arg_t struct is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, several types (see tuple 20.4) have constructors with allocator_arg_t as the first argument, immediately followed by an argument of a type that satisfies the Allocator requirements (17.6.3.5).

20.7.7 uses_allocator

20.7.7.1 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 (20.10.1). 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 (14.8.2) and is_convertible<Alloc, T::allocator_type>::value != 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.

20.7.7.2 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<T, Alloc>::value is false and is_constructible<T, V1, V2, ..., VN>::value is true, then obj is initialized as obj(v1, v2, ..., vN);
2. otherwise, if uses_allocator<T, Alloc>::value is true and is_constructible<T, allocator_arg_t, Alloc, V1, V2, ..., VN>::value is true, then obj is initialized as obj(allocator_arg, alloc, v1, v2, ..., vN);
3. otherwise, if uses_allocator<T, Alloc>::value is true and is_constructible<T, V1, V2, ..., VN, Alloc>::value 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<T, Alloc>::value 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]

20.7.8 Allocator traits

The class template allocator_traits supplies a uniform interface to all allocator types. An allocator cannot be a non-class type, however, even if allocator_traits supplies the entire required interface. [Note: Thus, it is always possible to create a derived class from an allocator. —end note]

namespace std {
    template <class Alloc> struct allocator_traits {
        typedef Alloc allocator_type;
        typedef typename Alloc::value_type value_type;
        typedef see below pointer;
        typedef see below const_pointer;
        typedef see below void_pointer;
    }
}

§ 20.7.8
typedef see below const_void_pointer;

typedef see below difference_type;
typedef see below size_type;

typedef see below propagate_on_container_copy_assignment;
typedef see below propagate_on_container_move_assignment;
typedef see below propagate_on_container_swap;
typedef see below is_always_equal;

template <class T> using rebind_alloc = see below;
template <class T> using rebind_traits = allocator_traits<rebind_alloc<T> >;

static pointer allocate(Alloc& a, size_type n);
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);

}
typedef see below size_type;

Type: Alloc::size_type if the qualified-id Alloc::size_type is valid and denotes a type (14.8.2); otherwise, make_unsigned_t<difference_type>.

typedef see below propagate_on_container_copy_assignment;

Type: Alloc::propagate_on_container_copy_assignment if the qualified-id Alloc::propagate_on_container_copy_assignment is valid and denotes a type (14.8.2); otherwise false_type.

typedef see below propagate_on_container_move_assignment;

Type: Alloc::propagate_on_container_move_assignment if the qualified-id Alloc::propagate_on_container_move_assignment is valid and denotes a type (14.8.2); otherwise false_type.

typedef see below propagate_on_container_swap;

Type: Alloc::propagate_on_container_swap if the qualified-id Alloc::propagate_on_container_swap is valid and denotes a type (14.8.2); otherwise false_type.

typedef see below is_always_equal;

Type: Alloc::is_always_equal if the qualified-id Alloc::is_always_equal is valid and denotes a type (14.8.2); 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 (14.8.2); otherwise, Alloc<T, Args> if Alloc is a class template instantiation of the form Alloc<U, Args>, where Args is zero or more type arguments; otherwise, the instantiation of rebind_alloc is ill-formed.

20.7.8.2 Allocator traits static member functions

static pointer allocate(Alloc& a, size_type n);

Returns: a.allocate(n).

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().
Returns: \texttt{a.max\_size()} if that expression is well-formed; otherwise, \texttt{numeric\_limits<\textit{size\_type}>::max()}.

static Alloc select\_on\_container\_copy\_construction(const Alloc& rhs);

Returns: \texttt{rhs.select\_on\_container\_copy\_construction()} if that expression is well-formed; otherwise, \texttt{rhs}.

\section*{20.7.9 The default allocator \hfill [default.allocator]}

namespace std {
    template <class T> class allocator;

    // specialize for \texttt{void}:
    template <> class allocator<void> {
        public:
            typedef void* pointer;
            typedef const void* const_pointer;
            // reference-to-\texttt{void} members are impossible.
            typedef void value_type;
            template <class U> struct rebind { typedef allocator<U> other; };
    };

    template <class T> class allocator {
        public:
            typedef size_t size_type;
            typedef ptrdiff_t difference_type;
            typedef T* pointer;
            typedef const T* const_pointer;
            typedef T& reference;
            typedef const T& const_reference;
            typedef T value_type;
            template <class U> struct rebind { typedef allocator<U> other; };
            typedef true_type propagate_on_container_move_assignment;
            typedef true_type is_always_equal;
            allocator() noexcept;
            allocator(const allocator&) noexcept;
            template <class U> allocator(const allocator<U>&) noexcept;
            ~allocator();

            pointer address(reference x) const noexcept;
            const_pointer address(const_reference x) const noexcept;

            pointer allocate(
                size_type, allocator<void>::const_pointer hint = 0);
            void deallocate(pointer p, size_type n);
            size_type max_size() const noexcept;

            template<class U, class... Args>
                void construct(U* p, Args&&... args);
            template <class U>
                void destroy(U* p);
    };

§ 20.7.9
20.7.9.1 allocator members

Except for the destructor, member functions of the default allocator shall not introduce data races (1.10) 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.

pointer address(reference x) const noexcept;

Returns: The actual address of the object referenced by x, even in the presence of an overloaded operator&.

const_pointer address(const_reference x) const noexcept;

Returns: The actual address of the object referenced by x, even in the presence of an overloaded operator&.

pointer allocate(size_type n, allocator<void>::const_pointer hint = 0);

[ Note: In a container member function, the address of an adjacent element is often a good choice to pass for the hint argument. — end note ]

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 (3.11).

Remark: the storage is obtained by calling ::operator new(std::size_t) (18.6.1), but it is unspecified when or how often this function is called. The use of hint is unspecified, but intended as an aid to locality if an implementation so desires.

Throws: bad_alloc if the storage cannot be obtained.

void deallocate(pointer p, size_type 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(void*, std::size_t) (18.6.1), but it is unspecified when this function is called.

size_type max_size() const noexcept;

Returns: The largest value N for which the call allocate(N,0) might succeed.

template <class U, class... Args>
void construct(U* p, Args&&... args);

Effects: ::new((void *)p) U(std::forward<Args>(args)...)

template <class U>
void destroy(U* p);

Effects: p->~U()

20.7.9.2 allocator globals

template <class T1, class T2>
bool operator==(const allocator<T1>&, const allocator<T2>&) noexcept;

Returns: true.

template <class T1, class T2>
bool operator!=(const allocator<T1>&, const allocator<T2>&) noexcept;

Returns: false.
20.7.10 Raw storage iterator

`raw_storage_iterator` is provided to enable algorithms to store their results into uninitialized memory. The template parameter `OutputIterator` is required to have its `operator*` return an object for which `operator&` is defined and returns a pointer to `T`, and is also required to satisfy the requirements of an output iterator (24.2.4).

```cpp
namespace std {
  template <class OutputIterator, class T>
  class raw_storage_iterator
  : public iterator<output_iterator_tag,void,void,void,void> {
    public:
      explicit raw_storage_iterator(OutputIterator x);

      raw_storage_iterator& operator*();
      raw_storage_iterator& operator=(const T& element);
      raw_storage_iterator& operator++();
      raw_storage_iterator operator++(int);
  };
}
```

1. `explicit raw_storage_iterator(OutputIterator x);`
   Effects: Initializes the iterator to point to the same value to which `x` points.

2. `raw_storage_iterator& operator*();`
   Returns: `*this`

3. `raw_storage_iterator& operator=(const T& element);`
   Effects: Constructs a value from `element` at the location to which the iterator points.
   Returns: A reference to the iterator.

4. `raw_storage_iterator& operator++();`
   Effects: Pre-increment: advances the iterator and returns a reference to the updated iterator.

5. `raw_storage_iterator operator++(int);`
   Effects: Post-increment: advances the iterator and returns the old value of the iterator.

20.7.11 Temporary buffers

```cpp
template <class T>
pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n) noexcept;
```

1. `Effects: Obtains a pointer to storage sufficient to store up to `n` adjacent `T` objects. It is implementation-defined whether over-aligned types are supported (3.11).

2. `Returns: A pair containing the buffer’s address and capacity (in the units of sizeof(T)), or a pair of 0 values if no storage can be obtained or if `n <= 0`.

```cpp
template <class T> void return_temporary_buffer(T* p);
```

3. `Effects: Deallocates the buffer to which `p` points.

4. `Requires: The buffer shall have been previously allocated by `get_temporary_buffer`.`
20.7.12 Specialized algorithms

All the iterators that are used as template parameters in the following algorithms are required to have their
operator* return an object for which operator& is defined and returns a pointer to T. In the algorithm uninitialized_copy, the template parameter InputIterator is required to satisfy the requirements of an input iterator (24.2.3). In all of the following algorithms, the template parameter ForwardIterator is required to satisfy the requirements of a forward iterator (24.2.5), and is required to have the property that no exceptions are thrown from increment, assignment, comparison, or indirection through valid iterators. In the following algorithms, if an exception is thrown there are no effects.

20.7.12.1 addressof

template <class T> 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&.

20.7.12.2 uninitialized_copy

template <class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_copy(InputIterator first, InputIterator last, ForwardIterator result);

Effects:
for (; first != last; ++result, ++first)
  ::new (static_cast<void*>(&*result))
  typename iterator_traits<ForwardIterator>::value_type(*first);

Returns: result

20.7.12.3 uninitialized_fill

template <class ForwardIterator, class T>
void uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x);

Effects:
for (; first != last; ++first)
  ::new (static_cast<void*>(&*first))
  typename iterator_traits<ForwardIterator>::value_type(x);
20.7.12.4 uninitialized_fill_n

```cpp
template <class ForwardIterator, class Size, class T>
ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
```

Effects:
```
for (; n--; ++first)
    ::new (static_cast<void*>(&*first))
        typename iterator_traits<ForwardIterator>::value_type(x);
return first;
```

20.7.13 C library

1 Table 45 describes the header `<cstdlib>`.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
<td>calloc</td>
</tr>
<tr>
<td></td>
<td>free</td>
</tr>
</tbody>
</table>

2 The contents are the same as the Standard C library header `<stdlib.h>`, with the following changes:

3 The functions `calloc()`, `malloc()`, and `realloc()` do not attempt to allocate storage by calling `::operator new()` (18.6).

4 The function `free()` does not attempt to deallocate storage by calling `::operator delete()`.

See also: ISO C Clause 7.11.2.

5 Storage allocated directly with `malloc()`, `calloc()`, or `realloc()` is implicitly declared reachable (see 3.7.4.3) 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]

6 Table 46 describes the header `<cstring>`.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro:</td>
<td>NULL</td>
</tr>
<tr>
<td>Type:</td>
<td>size_t</td>
</tr>
<tr>
<td>Functions</td>
<td>memchr</td>
</tr>
<tr>
<td></td>
<td>memmove</td>
</tr>
</tbody>
</table>

7 The contents are the same as the Standard C library header `<string.h>`, with the change to `memchr()` specified in 21.8.

See also: ISO C Clause 7.11.2.
20.8 Smart pointers

20.8.1 Class template unique_ptr

A unique pointer is an object that owns another object and manages that other object through a pointer. More precisely, a unique pointer is an object $u$ that stores a pointer to a second object $p$ and will dispose of $p$ when $u$ is itself destroyed (e.g., when leaving block scope (6.7)). In this context, $u$ is said to own $p$.

The mechanism by which $u$ disposes of $p$ is known as $p$’s associated deleter, a function object whose correct invocation results in $p$’s appropriate disposition (typically its deletion).

Let the notation $u.p$ denote the pointer stored by $u$, and let $u.d$ denote the associated deleter. Upon request, $u$ can reset (replace) $u.p$ and $u.d$ with another pointer and deleter, but must properly dispose 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:

- $u2.p$ is equal to the pre-transfer $u.p$,
- $u.p$ is equal to nullptr, and
- if the pre-transfer $u.d$ maintained state, such state has been transferred to $u2.d$.

As in the case of a reset, $u2$ must properly dispose 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 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 MoveConstructible and MoveAssignable, but is not CopyConstructible nor CopyAssignable. The template parameter $T$ of unique_ptr may be an incomplete type.

[Note: The uses of unique_ptr include providing exception safety for dynamically allocated memory, passing ownership of dynamically allocated memory to a function, and returning dynamically allocated memory from a function. —end note]

```cpp
namespace std {
  template<class T> struct default_delete;
  template<class T> struct default_delete<T[]>;

  template<class T, class D = default_delete<T>> class unique_ptr;
  template<class T, class D> class unique_ptr<T[], D>;

  template<class T, class... Args> unique_ptr<T> make_unique(Args&&... args);
  template<class T> unique_ptr<T> make_unique(size_t n);
  template<class T, class... Args> unspecified make_unique(Args&&...) = delete;

  template<class T1, class D1, class T2, class D2> void swap(unique_ptr<T1, D1>& x, unique_ptr<T2, D2>& y) noexcept;
  template<class T1, class D1, class T2, class D2> bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
  template<class T1, class D1, class T2, class D2> bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
  template<class T1, class D1, class T2, class D2> bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
  template<class T1, class D1, class T2, class D2> bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
  template<class T1, class D1, class T2, class D2> bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
  template<class T1, class D1, class T2, class D2> bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
}
```

§ 20.8.1 556
bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

template<class T1, class D1, class T2, class D2>
bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

template <class 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) 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);

}  

20.8.1.1 Default deleters [unique.ptr.dltr]
20.8.1.1.1 In general [unique.ptr.dltr.general]

1 The class template default_delete serves as the default deleter (destruction policy) for the class template unique_ptr.

2 The template parameter T of default_delete may be an incomplete type.

20.8.1.1.2 default_delete [unique.ptr.dltr.dflt]

namespace std {

    template <class T> struct default_delete {
        constexpr default_delete() noexcept = default;
        template <class U> default_delete(const default_delete<U>&) noexcept = default;
        void operator() (T* ptr) const;
    };
}

template <class U> default_delete(const default_delete<U>& other) noexcept;

1 Effects: Constructs a default_delete object from another default_delete<U> object.

2 Remarks: This constructor shall not participate in overload resolution unless U* is implicitly convertible to T*.

void operator() (T* ptr) const;

§ 20.8.1.1.2
**Effects:** calls `delete` on `ptr`.

**Remarks:** If `T` is an incomplete type, the program is ill-formed.

### 20.8.1.1.3 `default_delete<T[]>`

```cpp
namespace std {
    template <class T> struct default_delete<T[]> {
        constexpr default_delete() noexcept = default;
        template <class U> default_delete(const default_delete<U[]>&) noexcept;
        template <class U> void operator()(U* ptr) const;
    };
}
```

**Effects:** constructs a `default_delete` object from another `default_delete<U[]>` object.

**Remarks:** This constructor shall not participate in overload resolution unless `U(*)[]` is convertible to `T(*)[]`.

```cpp
template <class U> default_delete(const default_delete<U[]>& other) noexcept;
```

**Effects:** calls `delete[]` on `ptr`.

**Remarks:** If `U` is an incomplete type, the program is ill-formed. This function shall not participate in overload resolution unless `U(*)[]` is convertible to `T(*)[]`.

### 20.8.1.2 `unique_ptr` for single objects

```cpp
namespace std {
    template <class T, class D = default_delete<T>> class unique_ptr {
        public:
            typedef see below pointer;
            typedef T element_type;
            typedef D deleter_type;

            // 20.8.1.2.1, constructors
            constexpr unique_ptr() noexcept;
            explicit unique_ptr(pointer p) noexcept;
            unique_ptr(pointer p, see below d1) noexcept;
            unique_ptr(pointer p, see below d2) noexcept;
            unique_ptr(unique_ptr&amp; u) noexcept;
            constexpr unique_ptr(nullptr_t) noexcept:
                unique_ptr() {
            } // 20.8.1.2.1, constructors

            // 20.8.1.2.2, destructor
            ~unique_ptr();

            // 20.8.1.2.3, assignment
            unique_ptr&amp; operator=(unique_ptr&amp; u) noexcept;
            template <class U, class E> unique_ptr&amp; operator=(unique_ptr<U, E>&& u) noexcept;
            unique_ptr&amp; operator=(nullptr_t) noexcept;

            // 20.8.1.2.4, observers
            add_lvalue_reference_t<T> operator*() const;
            pointer operator->() const noexcept;
    }
}
```

§ 20.8.1.2 558
pointer get() const noexcept;

deleter_type& get_deleter() noexcept;

const deleter_type& get_deleter() const noexcept;

explicit operator bool() const noexcept;

// 20.8.1.2.5 modifiers

pointer release() noexcept;

void reset(pointer p = pointer()) noexcept;

void swap(unique_ptr& u) noexcept;

// disable copy from lvalue

unique_ptr(const unique_ptr&) = delete;

unique_ptr& operator=(const unique_ptr&) = delete;

1 The default type for the template parameter D is default_delete. A client-supplied template argument D shall be a function object type (20.9), 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 24).

3 If the qualified-id remove_reference_t<D>::pointer is valid and denotes a type (14.8.2), then unique_ptr<T, D>::pointer shall be a synonym for remove_reference_t<D>::pointer. Otherwise unique_ptr<T, D>::pointer shall be a synonym for element_type*. The type unique_ptr<T, D>::pointer shall satisfy the requirements of NullablePointer (17.6.3.3).

[Example: Given an allocator type X (17.6.3.5) and letting A be a synonym for allocator_traits<X>, the types A::pointer, A::const_pointer, A::void_pointer, and A::const_void_pointer may be used as unique_ptr<T, D>::pointer. —end example]

20.8.1.2.1 unique_ptr constructors [unique.ptr.single.ctor]

cconstexpr unique_ptr() noexcept;

1 Requires: D shall satisfy the requirements of DefaultConstructible (Table 19), and that construction shall not throw an exception.

2 Effects: Constructs a unique_ptr object that owns nothing, value-initializing the stored pointer and the stored deleter.

3 Postconditions: get() == nullptr. get_deleter() returns a reference to the stored deleter.

4 Remarks: If this constructor is instantiated with a pointer type or reference type for the template argument D, the program is ill-formed.

explicit unique_ptr(pointer p) noexcept;

5 Requires: D shall satisfy the requirements of DefaultConstructible (Table 19), and that construction shall not throw an exception.

6 Effects: Constructs a unique_ptr which owns p, initializing the stored pointer with p and value-initializing the stored deleter.

7 Postconditions: get() == p. get_deleter() returns a reference to the stored deleter.

8 Remarks: If this constructor is instantiated with a pointer type or reference type for the template argument D, the program is ill-formed.
The signature of these constructors depends upon whether \( D \) is a reference type. If \( D \) is non-reference type \( A \), then the signatures are:

\[
\begin{align*}
\text{unique} \_\text{ptr} & (\text{pointer} \ p, \ \text{const} \ A & d); \\
\text{unique} \_\text{ptr} & (\text{pointer} \ p, \ & A & d);
\end{align*}
\]

If \( D \) is an lvalue-reference type \( A & \), then the signatures are:

\[
\begin{align*}
\text{unique} \_\text{ptr} & (\text{pointer} \ p, \ & A & d); \\
\text{unique} \_\text{ptr} & (\text{pointer} \ p, \ & A & d);
\end{align*}
\]

If \( D \) is an lvalue-reference type \( \text{const} A & \), then the signatures are:

\[
\begin{align*}
\text{unique} \_\text{ptr} & (\text{pointer} \ p, \ \text{const} \ A & d); \\
\text{unique} \_\text{ptr} & (\text{pointer} \ p, \ \text{const} \ A & d);
\end{align*}
\]

\begin{enumerate}
\item \textbf{Requires:}
\begin{enumerate}
\item If \( D \) is not an lvalue-reference type then
\begin{enumerate}
\item If \( d \) is an lvalue or \text{const} rvalue then the first constructor of this pair will be selected. \( D \) shall satisfy the requirements of \text{CopyConstructible} (Table 21), and the copy constructor of \( D \) shall not throw an exception. This \text{unique} \_\text{ptr} will hold a copy of \( d \).
\item Otherwise, \( d \) is a non-const rvalue and the second constructor of this pair will be selected. \( D \) shall satisfy the requirements of \text{MoveConstructible} (Table 20), and the move constructor of \( D \) shall not throw an exception. This \text{unique} \_\text{ptr} will hold a value move constructed from \( d \).
\end{enumerate}
\item Otherwise \( D \) is an lvalue-reference type. \( d \) shall be reference-compatible with one of the constructors. If \( d \) is an rvalue, it will bind to the second constructor of this pair and the program is ill-formed. \[ \text{Note: The diagnostic could be implemented using a } \text{static} \_\text{assert} \text{ which assures that } D \text{ is not a reference type.} \quad \text{end note} \] Else \( d \) is an lvalue and will bind to the first constructor of this pair. The type which \( D \) references need not be \text{CopyConstructible} nor \text{MoveConstructible}. This \text{unique} \_\text{ptr} will hold a \( D \) which refers to the lvalue \( d \). \[ \text{Note: } D \text{ may not be an rvalue-reference type.} \quad \text{end note} \]
\end{enumerate}
\end{enumerate}

\begin{enumerate}
\item \textbf{Effects:} Constructs a \text{unique} \_\text{ptr} object which owns \( p \), initializing the stored pointer with \( p \) and initializing the deleter as described above.
\item \textbf{Postconditions:} \text{get()} == \( p \), \text{get} \_\text{deleter}() returns a reference to the stored deleter. If \( D \) is a reference type then \text{get} \_\text{deleter}() returns a reference to the lvalue \( d \).
\end{enumerate}

\[ \text{Example:} \]

\[
\begin{align*}
D \ d; \\
\text{unique} \_\text{ptr} & <\text{int}, \ D> \ p1(new \text{ int}, \ & D()); \quad \text{// } D \text{ must be } \text{MoveConstructible} \\
\text{unique} \_\text{ptr} & <\text{int}, \ & D> \ p2(new \text{ int}, \ d); \quad \text{// } D \text{ must be } \text{CopyConstructible} \\
\text{unique} \_\text{ptr} & <\text{int}, \ D &> \ p3(new \text{ int}, \ d); \quad \text{// } p3 \text{ holds a reference to } d \\
\text{unique} \_\text{ptr} & <\text{int}, \ \text{const} \ D &> \ p4(new \text{ int}, \ D()); \quad \text{// } \text{error: } \text{rvalue deleter object combined} \\
& \quad \text{// with reference deleter type}
\end{align*}
\]

\[ \text{— } \text{end example} \]
requires: If \( \text{D} \) is not a reference type, \( \text{D} \) shall satisfy the requirements of \text{MoveConstructible} (Table 20). Construction of the deleter from an rvalue of type \( \text{D} \) shall not throw an exception.

effects: Constructs a \text{unique_ptr} by transferring ownership from \( \text{u} \) to \*\this. If \( \text{D} \) is a reference type, this deleter is copy constructed from \( \text{u} \)'s deleter; otherwise, this deleter is move constructed from \( \text{u} \)'s deleter. [ Note: The deleter constructor can be implemented with std::forward<\text{D}>. — end note ]

postconditions: \text{get()} yields the value \( \text{u.get()} \) yielded before the construction. \text{get_deleter()} returns a reference to the stored deleter that was constructed from \( \text{u.get_deleter()} \). If \( \text{D} \) is a reference type then \text{get_deleter()} and \( \text{u.get_deleter()} \) both reference the same lvalue deleter.

template <class U, class E> unique_ptr(unique_ptr<U, E>&& u) noexcept;

requires: If \( \text{E} \) is not a reference type, construction of the deleter from an rvalue of type \( \text{E} \) shall be well formed and shall not throw an exception. Otherwise, \( \text{E} \) is a reference type and construction of the deleter from an lvalue of type \( \text{E} \) shall be well formed and shall not throw an exception.

remarks: This constructor shall not participate in overload resolution unless:

(19.1) unique_ptr<\text{U}, \text{E}>::pointer is implicitly convertible to pointer,

(19.2) \( \text{U} \) is not an array type, and

(19.3) either \( \text{D} \) is a reference type and \( \text{E} \) is the same type as \( \text{D} \), or \( \text{D} \) is not a reference type and \( \text{E} \) is implicitly convertible to \( \text{D} \).

effects: Constructs a \text{unique_ptr} by transferring ownership from \( \text{u} \) to \*\this. If \( \text{E} \) is a reference type, this deleter is copy constructed from \( \text{u} \)'s deleter; otherwise, this deleter is move constructed from \( \text{u} \)'s deleter. [ Note: The deleter constructor can be implemented with std::forward<\text{E}>. — end note ]

postconditions: \text{get()} yields the value \( \text{u.get()} \) yielded before the construction. \text{get_deleter()} returns a reference to the stored deleter that was constructed from \( \text{u.get_deleter()} \).

20.8.1.2.2 unique_ptr destructor

\text{~unique_ptr();}

requirements: The expression \text{get_deleter()}(\text{get()}) shall be well formed, shall have well-defined behavior, and shall not throw exceptions. [ Note: The use of \text{default_delete} requires \( \text{T} \) to be a complete type. — end note ]

effects: If \( \text{get() == nullptr} \) there are no effects. Otherwise \text{get_deleter()}(\text{get()}).

20.8.1.2.3 unique_ptr assignment

unique_ptr& operator=(unique_ptr&& u) noexcept;

requirements: If \( \text{E} \) is not a reference type, assignment of the deleter from an rvalue of type \( \text{E} \) shall be well-formed and shall not throw an exception. Otherwise, \( \text{E} \) is a reference type and assignment of the deleter from an lvalue of type \( \text{E} \) 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.

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: reset().

Postcondition: get() == nullptr

Returns: *this.

20.8.1.2.4 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: use typically requires that T be a complete type.

pointer get() const noexcept;

Returns: The stored pointer.

deleter_type& get_deleter() noexcept;

Returns: A reference to the stored deleter.

explicit operator bool() const noexcept;

Returns: get() != nullptr.

20.8.1.2.5 unique_ptr modifiers

pointer release() noexcept;

Postcondition: 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 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 (17.6.3.2) and shall not throw an exception under swap.

Effects: Invokes swap on the stored pointers and on the stored deleters of *this and u.
20.8.1.3 unique_ptr for array objects with a runtime length

namespace std {
    template <class T, class D> class unique_ptr<T[], D> {
public:
    typedef see below pointer;
    typedef T element_type;
    typedef D deleter_type;

    // 20.8.1.3.1, 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<T[]>& u) noexcept;
    template <class U, class E>
        unique_ptr(unique_ptr<U, E>&& u) noexcept;
    constexpr unique_ptr(nullptr_t) noexcept : unique_ptr() { }

    // destructor
    ~unique_ptr();

    // assignment
    unique_ptr& operator=(unique_ptr&& u) noexcept;
    template <class U, class E>
        unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
    unique_ptr& operator=(nullptr_t) noexcept;

    // 20.8.1.3.3, observers
    T& operator[](size_t i) const;
    pointer get() const noexcept;
    deleter_type& get_deleter() noexcept;
    const deleter_type& get_deleter() const noexcept;
    explicit operator bool() const noexcept;

    // 20.8.1.3.4 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.

§ 20.8.1.3
The indexing observer `operator[]` is provided.

— The default deleter will call `delete[]`.

Descriptions are provided below only for members that differ from the primary template.

The template argument `T` shall be a complete type.

### 20.8.1.3.1 unique_ptr constructors

```cpp
template <class U> explicit unique_ptr(U p) noexcept;
template <class U> unique_ptr(U p, see below d) noexcept;
```

These constructors behave the same as the constructors that take a `pointer` parameter in the primary template except that they shall not participate in overload resolution unless either

1. `U` is the same type as `pointer`, or
2. `pointer` is the same type as `element_type*`, `U` is a pointer type `V*`, and `V(*[])` is convertible to `element_type(*)[]`.

```cpp
template <class U, class E>
unique_ptr(unique_ptr<U, E>&& u) noexcept;
```

This constructor behaves the same as in the primary template, except that it shall not participate in overload resolution unless all of the following conditions hold, where `UP` is `unique_ptr<U, E>`:

1. `U` is an array type, and
2. `pointer` is the same type as `element_type*`, and
3. `UP::pointer` is the same type as `UP::element_type*`, and
4. `UP::element_type(*)[]` is convertible to `element_type(*)[]`, and
5. either `D` is a reference type and `E` is the same type as `D`, or `D` is not a reference type and `E` is implicitly convertible to `D`.

[Note: this replaces the overload-resolution specification of the primary template — end note]

### 20.8.1.3.2 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. either `D` is a reference type and `E` is the same type as `D`, or `D` is not a reference type and `E` is implicitly convertible to `D`.

[Note: this replaces the overload-resolution specification of the primary template — end note]

### 20.8.1.3.3 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.
20.8.1.3.4 unique_ptr modifiers

```cpp
void reset(nullptr_t p = nullptr) noexcept;
```

1. 
   Effects: Equivalent to `reset(pointer())`.

```cpp
template <class U> void reset(U p) noexcept;
```

2. 
   This function behaves the same as the `reset` member of the primary template, except that it shall not participate in overload resolution unless either
   - U is the same type as `pointer`, or
   - `pointer` is the same type as `element_type*`, U is a pointer type `V*`, and `V(*)[]` is convertible to `element_type(*)[]`.

20.8.1.4 unique_ptr creation

```cpp
template <class T, class... Args> unique_ptr<T> make_unique(Args&&... args);
```

1. 
   Remarks: This function shall not participate in overload resolution unless `T` is not an array.

2. 
   Returns: `unique_ptr<T>(new T(std::forward<Args>(args)...))`.

```cpp
template <class T> unique_ptr<T> make_unique(size_t n);
```

3. 
   Remarks: This function shall not participate in overload resolution unless `T` is an array of unknown bound.

4. 
   Returns: `unique_ptr<T>(new remove_extent_t<T>[n]())`.

```cpp
template <class T, class... Args> unspecified make_unique(Args&&...) = delete;
```

5. 
   Remarks: This function shall not participate in overload resolution unless `T` is an array of known bound.

20.8.1.5 unique_ptr specialized algorithms

```cpp
template <class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y) noexcept;
```

1. 
   Effects: Calls `x.swap(y)`.

```cpp
template <class T1, class D1, class T2, class D2>
  bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

2. 
   Returns: `x.get() == y.get()`.

```cpp
template <class T1, class D1, class T2, class D2>
  bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

3. 
   Returns: `x.get() != y.get()`.

```cpp
template <class T1, class D1, class T2, class D2>
  bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

4. 
   Requires: Let `CT` be `common_type<unique_ptr<T1, D1>::pointer, unique_ptr<T2, D2>::pointer>::type`. Then the specialization `less<CT>` shall be a function object type (20.9) that induces a strict weak ordering (25.4) on the pointer values.

5. 
   Returns: `less<CT>()(x.get(), y.get())`.

6. 
   Remarks: If `unique_ptr<T1, D1>::pointer` is not implicitly convertible to `CT` or `unique_ptr<T2, D2>::pointer` is not implicitly convertible to `CT`, the program is ill-formed.
template <class T1, class D1, class T2, class D2>
bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

    Returns: !(y < x).

template <class T1, class D1, class T2, class D2>
bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

    Returns: y < x.

template <class T1, class D1, class T2, class D2>
bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

    Returns: !(x < y).

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;

    Returns: (bool)x.

template <class T, class D>
bool operator<(const unique_ptr<T, D>& x, nullptr_t);

    Requires: The specialization less<unique_ptr<T, D>::pointer> shall be a function object type (20.9) that induces a strict weak ordering (25.4) on the pointer values.

    Returns: The first function template returns less<unique_ptr<T, D>::pointer>()(x.get(), nullptr). The second function template returns less<unique_ptr<T, D>::pointer>()(nullptr, x.get()).

template <class T, class D>
bool operator>(const unique_ptr<T, D>& x, nullptr_t);

    Returns: The first function template returns nullptr < x. The second function template returns x < nullptr.

template <class T, class D>
bool operator>=(const unique_ptr<T, D>& x, nullptr_t);

    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).

20.8.2 Shared-ownership pointers

20.8.2.1 Class bad_weak_ptr

namespace std {
    class bad_weak_ptr: public std::exception {
public:
    bad_weak_ptr() noexcept;
    } // namespace std
}

1 An exception of type bad_weak_ptr is thrown by the shared_ptr constructor taking a weak_ptr.

bad_weak_ptr() noexcept;

2 Postconditions: what() returns an implementation-defined NTBS.

20.8.2.2 Class template shared_ptr

1 The shared_ptr class template stores a pointer, usually obtained via new. shared_ptr implements semantics of shared ownership: the last remaining owner of the pointer is responsible for destroying the object, or otherwise releasing the resources associated with the stored pointer. A shared_ptr object is empty if it does not own a pointer.

namespace std {
    template<class T> class shared_ptr {
public:
    typedef T element_type;

    // 20.8.2.2.1, constructors:
    constexpr shared_ptr() noexcept;
    template<class Y> explicit shared_ptr(Y* p);
    template<class Y, class D> shared_ptr(Y* p, D d);
    template<class Y, class D, class A> shared_ptr(Y* p, D d, A a);
    template <class D> shared_ptr(nullptr_t p, D d);
    template <class D, class A> shared_ptr(nullptr_t p, D d, A a);
    template<class Y> shared_ptr(const shared_ptr<Y>& r, T* 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);
    template <class Y, class D> shared_ptr& operator=(const shared_ptr<Y>&& r);
    template <class D> shared_ptr& operator=(const shared_ptr<Y>&& r) noexcept;
    template <class D, class A> shared_ptr& operator=(const shared_ptr<Y>&& r) noexcept;
    template <class D> shared_ptr& operator=(shared_ptr<Y>&& r) noexcept;
    template <class D, class A> shared_ptr& operator=(shared_ptr<Y>&& r) noexcept;
    } // template<class T> class shared_ptr

    // 20.8.2.2.2, destructor:
    ~shared_ptr();

    // 20.8.2.2.3, 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);
    } // template<class T> class shared_ptr

§ 20.8.2.2
// 20.8.2.2.4, modifiers:
void swap(shared_ptr& r) noexcept;
void reset() noexcept;
template<class Y> void reset(Y* p);
template<class Y, class D> void reset(Y* p, D d);
template<class Y, class D, class A> void reset(Y* p, D d, A a);

// 20.8.2.2.5, observers:
T* get() const noexcept;
T& operator*() const noexcept;
T* operator->() const noexcept;
long use_count() const noexcept;
bool unique() const noexcept;
explicit operator bool() const noexcept;
template<class U> bool owner_before(shared_ptr<U> const& b) const;
template<class U> bool owner_before(weak_ptr<U> const& b) const;
};

// 20.8.2.2.6, shared_ptr creation
template<class T, class... Args> shared_ptr<T> make_shared(Args&&... args);
template<class T, class A, class... Args>
    shared_ptr<T> allocate_shared(const A& a, Args&&... args);

// 20.8.2.2.7, shared_ptr comparisons:
template<class T, class U>
    bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
    bool operator!=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
    bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
    bool operator>(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
    bool operator<=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
    bool operator>=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template <class T>
    bool operator==(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
    bool operator==(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
    bool operator!=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
    bool operator!=(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
    bool operator<(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
    bool operator<(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
    bool operator<=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
    bool operator<=(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
    bool operator>(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
    bool operator>(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
    bool operator>=(const shared_ptr<T>& a, nullptr_t) noexcept;

§ 20.8.2.2
template <class T>
    bool operator>(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
    bool operator>=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
    bool operator>=(nullptr_t, const shared_ptr<T>& b) noexcept;

    // 20.8.2.2.8, shared_ptr specialized algorithms:
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;

    // 20.8.2.2.9, 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;

    // 20.8.2.2.10, shared_ptr get_deleter:
template<class D, class T> D* get_deleter(const shared_ptr<T>& p) noexcept;

    // 20.8.2.2.11, 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);
} // namespace std

Specializations of shared_ptr shall be CopyConstructible, CopyAssignable, and LessThanComparable, allowing their use in standard containers. Specializations of shared_ptr shall be 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.

[Example:
    if(shared_ptr<X> px = dynamic_pointer_cast<X>(py)) {
        // do something with px
    }
  ]
  —end example]

For purposes of determining the presence of a data race, member functions shall access and modify only the shared_ptr and weak_ptr objects themselves and not objects they refer to. Changes in use_count() do not reflect modifications that can introduce data races.

20.8.2.2.1 shared_ptr constructors

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: p shall be convertible to T*. Y shall be a complete type. The expression delete p shall be well formed, shall have well defined behavior, and shall not throw exceptions.

    Effects: Constructs a shared_ptr object that owns the pointer p.

    Postconditions: use_count() == 1 && get() == p.
6  
Throws: bad_alloc, or an implementation-defined exception when a resource other than memory

could not be obtained.

7  
Exception safety: If an exception is thrown, delete p is called.

8

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);

8  
Requires: p shall be convertible to T*. D shall be CopyConstructible. The copy constructor and

destructor of D shall not throw exceptions. The expression d(p) shall be well formed, shall have

well defined behavior, and shall not throw exceptions. A shall be an allocator (17.6.3.5). The copy

constructor and destructor of A shall not throw exceptions.

9  
Effects: Constructs a shared_ptr object that owns the object p and the deleter d. The second and

fourth constructors shall use a copy of a to allocate memory for internal use.

10  
Postconditions: use_count() == 1 && get() == p.

11  
Throws: bad_alloc, or an implementation-defined exception when a resource other than memory

could not be obtained.

12  
Exception safety: If an exception is thrown, d(p) is called.

13

template<class Y> shared_ptr(const shared_ptr<Y>& r, T* p) noexcept;

13  
Effects: Constructs a shared_ptr instance that stores p and shares ownership with r.

14  
Postconditions: get() == p && use_count() == r.use_count()

[ Note: To avoid the possibility of a dangling pointer, the user of this constructor must ensure that p

remains valid at least until the ownership group of r is destroyed. — end note ]

16  
[ Note: This constructor allows creation of an empty shared_ptr instance with a non-null stored

pointer. — end note ]

17

shared_ptr(const shared_ptr& r) noexcept;

17  
 Remark: The second constructor shall not participate in overload resolution unless Y* is implicitly

convertible to T*.

18  
Effects: If r is empty, constructs an empty shared_ptr object; otherwise, constructs a shared_ptr

object that shares ownership with r.

19  
Postconditions: get() == r.get() && use_count() == r.use_count().

shared_ptr(shared_ptr&& r) noexcept;

19  
 Remark: The second constructor shall not participate in overload resolution unless Y* is convertible
to T*.

19  
Effects: Move-constructs a shared_ptr instance from r.

20  
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);

§ 20.8.2.2.1
Requires: \( Y \) shall be convertible to \( T \).

**Effects:** Constructs a `shared_ptr` object that shares ownership with \( r \) and stores a copy of the pointer stored in \( r \).

**Postconditions:** `use_count() == r.use_count()`.

**Throws:** `bad_weak_ptr` when \( r \).expired().

**Exception safety:** If an exception is thrown, the constructor has no effect.

```cpp
template <class Y, class D> shared_ptr(unique_ptr<Y, D>&& r);
```

**Remark:** This constructor shall not participate in overload resolution unless `unique_ptr<Y, D>::pointer` is convertible to \( T \).

**Effects:** Equivalent to `shared_ptr(r.release(), r.get_deleter())` when \( D \) is not a reference type, otherwise `shared_ptr(r.release(), ref(r.get_deleter()))`.

**Exception safety:** If an exception is thrown, the constructor has no effect.

### 20.8.2.2.2 `shared_ptr` destructor

~`shared_ptr();`

**Effects:**

1. **(1.1)** If `*this` is empty or shares ownership with another `shared_ptr` instance (`use_count() > 1`), there are no side effects.
2. **(1.2)** Otherwise, if `*this` owns an object \( p \) and a deleter \( d \), \( d(p) \) is called.
3. **(1.3)** Otherwise, `*this` owns a pointer \( p \), and `delete p` is called.

**Note:** Since the destruction of `*this` decreases the number of instances that share ownership with `*this` by one, after `*this` has been destroyed all `shared_ptr` instances that shared ownership with `*this` will report a `use_count()` that is one less than its previous value. —end note

### 20.8.2.2.3 `shared_ptr` assignment

```cpp
shared_ptr& operator=(const shared_ptr& r) noexcept;
template<class Y> shared_ptr& operator=(const shared_ptr<Y>& r) noexcept;
```

**Effects:** Equivalent to `shared_ptr(r).swap(*this)`.

**Returns:** `*this`.

**Note:** The use count updates caused by the temporary object construction and destruction are not observable side effects, so the implementation may meet the effects (and the implied guarantees) via different means, without creating a temporary. In particular, in the example:

```cpp
shared_ptr<int> p(new int);
shared_ptr<void> q(p);
p = p;
q = p;
```

both assignments may be no-ops. —end note

```cpp
shared_ptr& operator=(shared_ptr&& r) noexcept;
template<class Y> shared_ptr& operator=(shared_ptr<Y>&& r) noexcept;
```

**Effects:** Equivalent to `shared_ptr(std::move(r)).swap(*this)`.

**Returns:** `*this`.

§ 20.8.2.2.3
template <class Y, class D> shared_ptr& operator=(unique_ptr<Y, D>&& r);

Effects: Equivalent to shared_ptr(std::move(r)).swap(*this).

Returns: *this

20.8.2.2.4 shared_ptr modifiers

void swap(shared_ptr& r) noexcept;

Effects: Exchanges the contents of *this and r.

void reset() noexcept;

Effects: Equivalent to shared_ptr().swap(*this).

template<class Y> void reset(Y* p);

Effects: Equivalent to shared_ptr(p).swap(*this).

template<class Y, class D> void reset(Y* p, D d);

Effects: Equivalent to shared_ptr(p, d).swap(*this).

template<class Y, class D, class A> void reset(Y* p, D d, A a);

Effects: Equivalent to shared_ptr(p, d, a).swap(*this).

20.8.2.2.5 shared_ptr observers

T* get() const noexcept;

Returns: the stored pointer.

T& operator*() const noexcept;

Requires: get() != 0.

Returns: *get().

Remarks: When T is 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().

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.

[Note: use_count() is not necessarily efficient. — end note]

bool unique() const noexcept;

Returns: use_count() == 1.

[Note: unique() may be faster than use_count(). If you are using unique() to implement copy on write, do not rely on a specific value when get() == nullptr. — end note]
explicit operator bool() const noexcept;

Returns: get() != 0.

template<class U> bool owner_before(shared_ptr<U> const& b) const;
template<class U> bool owner_before(weak_ptr<U> const& b) const;

Returns: An unspecified value such that

(12.1) — x.owner_before(y) defines a strict weak ordering as defined in 25.4;

(12.2) — under the equivalence relation defined by owner_before, !a.owner_before(b) & & !b.owner_before(a), two shared_ptr or weak_ptr instances are equivalent if and only if they share ownership or are both empty.

20.8.2.2.6 shared_ptr creation

template<class T, class... Args> shared_ptr<T> make_shared(Args&&... args);
template<class T, class A, class... Args>
shared_ptr<T> allocate_shared(const A& a, Args&&... args);

Requires: The expression ::new (pv) T(std::forward<Args>(args)...) where pv has type void* and points to storage suitable to hold an object of type T, shall be well formed. A shall be an allocator (17.6.3.5). The copy constructor and destructor of A shall not throw exceptions.

Effects: Allocates memory suitable for an object of type T and constructs an object in that memory via the placement new-expression ::new (pv) T(std::forward<Args>(args)...). The template allocate_shared uses a copy of a 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 of type T.

Postconditions: get() != 0 & & use_count() == 1

Throws: bad_alloc, or an exception thrown from A::allocate or from the constructor of T.

Remarks: Implementations should perform no more than one memory allocation. [ Note: This provides efficiency equivalent to an intrusive smart pointer. —end note ]

[ Note: These functions will typically allocate more memory than sizeof(T) to allow for internal bookkeeping structures such as the reference counts. —end note ]

20.8.2.2.7 shared_ptr comparison

template<class T, class U> bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;

Returns: a.get() == b.get().

template<class T, class U> bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;

Returns: less<V>()(a.get(), b.get()), where V is the composite pointer type (Clause 5) of T* and U*.

[ Note: Defining a comparison operator allows shared_ptr objects to be used as keys in associative containers. —end note ]

template <class T>
bool operator==(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
bool operator==(nullptr_t, const shared_ptr<T>& a) noexcept;

Returns: !a.
template <class T>
bool operator!=(const shared_ptr<T>& a, nullptr_t) noexcept;

Returns: (bool)a.

template <class T>
bool operator<(const shared_ptr<T>& a, nullptr_t) noexcept;

Returns: The first function template returns less<T*>(a.get(), nullptr). The second function template returns less<T*>(nullptr, a.get()).

template <class T>
bool operator>(const shared_ptr<T>& a, nullptr_t) 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;

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;

Returns: The first function template returns !(a < nullptr). The second function template returns !(nullptr < a).

20.8.2.2.8 shared_ptr specialized algorithms

template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;

Effects: Equivalent to a.swap(b).

20.8.2.2.9 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*>(r.get()) shall be well formed.

Returns: If r is empty, an empty shared_ptr<T>; otherwise, a shared_ptr<T> object that stores static_cast<T*>(r.get()) and shares ownership with r.

Postconditions: w.get() == static_cast<T*>(r.get()) and w.use_count() == r.use_count(), where w is the return value.

[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*>(r.get())` shall be well formed and shall have well defined behavior.

Returns:

(6.1) — When `dynamic_cast<T*>(r.get())` returns a nonzero value, a `shared_ptr<T>` object that stores a copy of it and shares ownership with `r`;

(6.2) — Otherwise, an empty `shared_ptr<T>` object.

Postcondition: `w.get() == dynamic_cast<T*>(r.get())`, where `w` is the return value.

[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*>(r.get())` shall be well formed.

Returns: If `r` is empty, an empty `shared_ptr<T>`; otherwise, a `shared_ptr<T>` object that stores `const_cast<T*>(r.get())` and shares ownership with `r`.

Postconditions: `w.get() == const_cast<T*>(r.get())` and `w.use_count() == r.use_count()`, where `w` is the return value.

[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]

```
20.8.2.2.10 get_deleter [util.smartptr.getdeleter]
```

```
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 `std::addressof(d)`; otherwise returns `nullptr`. The returned pointer remains valid as long as there exists a `shared_ptr` instance that owns `d`. [Note: It is unspecified whether the pointer remains valid longer than that. This can happen if the implementation doesn’t destroy the deleter until all `weak_ptr` instances that share ownership with `p` have been destroyed. — end note]

```
20.8.2.2.11 shared_ptr I/O [util.smartptr.shared.io]
```

```
template<class E, class T, class Y>
    basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os, shared_ptr<Y> const& p);
```

Effects: `os << p.get();`

Returns: `os`.

```
20.8.2.3 Class template weak_ptr [util.smartptr.weak]
```

The `weak_ptr` class template stores a weak reference to an object that is already managed by a `shared_ptr`. To access the object, a `weak_ptr` can be converted to a `shared_ptr` using the member function `lock`.

```
namespace std {
    template<class T> class weak_ptr {

        public:
            typedef T element_type;

            // 20.8.2.3.1, constructors
            constexpr weak_ptr() noexcept;
            template<class Y> weak_ptr(shared_ptr<Y> const& r) noexcept;
            weak_ptr(weak_ptr const& r) noexcept;
            template<class Y> weak_ptr(weak_ptr<Y> const& r) noexcept;

            // 20.8.2.3.2, use
```
weak_ptr(weak_ptr&& r) noexcept;
template<class Y> weak_ptr(weak_ptr<Y>&& r) noexcept;

// 20.8.2.3.2, destructor
-weak_ptr();

// 20.8.2.3.3, assignment
weak_ptr& operator=(weak_ptr const& r) noexcept;
template<class Y> weak_ptr& operator=(weak_ptr<Y> const& r) noexcept;
template<class Y> weak_ptr& operator=(shared_ptr<Y> const& r) noexcept;
weak_ptr& operator=(weak_ptr&& r) noexcept;
template<class Y> weak_ptr& operator=(weak_ptr<Y>&& r) noexcept;

// 20.8.2.3.4, modifiers
void swap(weak_ptr& r) noexcept;
void reset() noexcept;

// 20.8.2.3.5, observers
long use_count() const noexcept;
bool expired() const noexcept;
shared_ptr<T> lock() const noexcept;
template<class U> bool owner_before(shared_ptr<U> const& b) const;
template<class U> bool owner_before(weak_ptr<U> const& b) const;
};

// 20.8.2.3.6, specialized algorithms
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;
} // namespace std

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.

20.8.2.3.1 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;
template<class Y> weak_ptr(const weak_ptr<Y>& r) noexcept;
template<class Y> weak_ptr(const shared_ptr<Y>& r) noexcept;

Remark: The second and third constructors shall not participate in overload resolution unless Y* is implicitly convertible to 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;
template<class Y> weak_ptr(weak_ptr<Y>&& r) noexcept;

Remark: The second constructor shall not participate in overload resolution unless Y* is implicitly convertible to 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.
20.8.2.3.2 weak_ptr destructor

`weak_ptr();`

**Effects:** Destroys this `weak_ptr` object but has no effect on the object its stored pointer points to.

20.8.2.3.3 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.

`weak_ptr& operator=(weak_ptr&& r) noexcept;`

**Effects:** Equivalent to `weak_ptr(std::move(r)).swap(*this).`

20.8.2.3.4 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).`

20.8.2.3.5 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`.

**Note:** `use_count()` is not necessarily efficient. — end note

`bool expired() const noexcept;`

**Returns:** `use_count()` == 0.

**Note:** `expired()` may be faster than `use_count()`. — end note

`shared_ptr<T> lock() const noexcept;`

**Returns:** `expired()` ? `shared_ptr<T>()` : `shared_ptr<T>(*this)`, executed atomically.

**Template:***

- `template<class U> bool owner_before(shared_ptr<U>& a) const;`
- `template<class U> bool owner_before(weak_ptr<U>& b) const;`

**Returns:** An unspecified value such that

(6.1) `x.owner_before(y)` defines a strict weak ordering as defined in 25.4;

(6.2) under the equivalence relation defined by `owner_before, !a.owner_before(b) && !b.owner_before(a)`, two `shared_ptr` or `weak_ptr` instances are equivalent if and only if they `share ownership` or are both empty.
20.8.2.3.6  weak_ptr specialized algorithms

[util.smartptr.weak.spec]

template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;

1  Effects: Equivalent to a.swap(b).

20.8.2.4  Class template owner_less

[util.smartptr.ownerless]

1  The class template owner_less allows ownership-based mixed comparisons of shared and weak pointers.

namespace std {
    template<class T> struct owner_less;

    template<class T> struct owner_less<shared_ptr<T> > {
        typedef bool result_type;
        typedef shared_ptr<T> first_argument_type;
        typedef shared_ptr<T> second_argument_type;
        bool operator()(shared_ptr<T> const&, shared_ptr<T> const&) const;
        bool operator()(shared_ptr<T> const&, weak_ptr<T> const&) const;
        bool operator()(weak_ptr<T> const&, shared_ptr<T> const&) const;
    };

    template<class T> struct owner_less<weak_ptr<T> > {
        typedef bool result_type;
        typedef weak_ptr<T> first_argument_type;
        typedef weak_ptr<T> second_argument_type;
        bool operator()(weak_ptr<T> const&, weak_ptr<T> const&) const;
        bool operator()(shared_ptr<T> const&, weak_ptr<T> const&) const;
        bool operator()(weak_ptr<T> const&, shared_ptr<T> const&) const;
    };
}

2  operator()(x,y) shall return x.owner_before(y).  [ Note: Note that

(2.1) — operator() defines a strict weak ordering as defined in 25.4;

(2.2) — under the equivalence relation defined by operator(), !operator()(a, b) &amp; !operator()(b, a),
          two shared_ptr or weak_ptr instances are equivalent if and only if they share ownership or are both
          empty.

— end note ]

20.8.2.5  Class template enable_shared_from_this

[util.smartptr.enab]

1  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.

2  [ 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 < q ) &amp; !(q < p));  // p and q share ownership
            }

        — end example ]

§ 20.8.2.5
namespace std {
    template<class T> class enable_shared_from_this {
        protected:
            constexpr enable_shared_from_this() noexcept;
            enable_shared_from_this(enable_shared_from_this const&) noexcept;
            enable_shared_from_this& operator=(enable_shared_from_this const&) noexcept;
            ~enable_shared_from_this();
        public:
            shared_ptr<T> shared_from_this();
            shared_ptr<T const> shared_from_this() const;
    };
} // namespace std

The template parameter T of enable_shared_from_this may be an incomplete type.

constexpr enable_shared_from_this() noexcept;
enable_shared_from_this(const enable_shared_from_this<T>&) noexcept;

Effects: Constructs an enable_shared_from_this<T> object.

Returns: *this.

~enable_shared_from_this();

Effects: Destroys *this.

shared_ptr<T> shared_from_this();
shared_ptr<T const> shared_from_this() const;

Requires: enable_shared_from_this<T> shall be an accessible base class of T. *this shall be a subobject of an object t of type T. There shall be at least one shared_ptr instance p that owns &t.

Returns: A shared_ptr<T> object r that shares ownership with p.

Postconditions: r.get() == this.

[Note: A possible implementation is shown below:

    template<class T> class enable_shared_from_this {
        private:
            weak_ptr<T> __weak_this;
        protected:
            constexpr enable_shared_from_this() : __weak_this() { }
            enable_shared_from_this(enable_shared_from_this const &){ } 
            enable_shared_from_this& operator=(enable_shared_from_this const &){ return *this; }
            ~enable_shared_from_this() { }
        public:
            shared_ptr<T> shared_from_this() { return shared_ptr<T>(__weak_this); }
            shared_ptr<T const> shared_from_this() const { return shared_ptr<T const>(__weak_this); }
    };

The shared_ptr constructors that create unique pointers can detect the presence of an enable_shared_from_this base and assign the newly created shared_ptr to its __weak_this member. —end note]

§ 20.8.2.5
20.8.2.6 shared_ptr atomic access [util.smartptr.shared.atomic]

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 section and the instance is passed as their first argument.

The meaning of the arguments of type memory_order is explained in 29.3.

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: 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: 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: p\rightarrowwap(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);

Requires: p shall not be null and v shall not be null.

Requires: failure shall not be memory_order_release, memory_order_acq_rel, or stronger than success.

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.

Remarks: the weak forms may fail spuriously. See 29.6.

20.8.2.7  Smart pointer hash support  

template <class T, class D> struct hash<unique_ptr<T, D>>;

The template specialization shall meet the requirements of class template hash (20.9.13). For an object \*p of type UP, where UP is unique_ptr<T, D>, hash<UP>()(p) shall evaluate to the same value as hash<typename UP::pointer>()(p.get()).

Requires: The specialization hash<typename UP::pointer> shall be well-formed and well-defined, and shall meet the requirements of class template hash (20.9.13).
template <class T> struct hash<shared_ptr<T> >;

The template specialization shall meet the requirements of class template hash (20.9.13). For an object \( p \) of type \( \text{shared\_ptr}\langle T \rangle \), \( \text{hash}\langle \text{shared\_ptr}\langle T \rangle \rangle ()(p) \) shall evaluate to the same value as \( \text{hash}\langle T\ast\rangle ()(p\ast().get()) \).

### 20.9 Function objects

A function object type is an object type (3.9) that can be the type of the postfix-expression in a function call (5.2.2, 13.3.1.1). A function object is an object of a function object type. In the places where one would expect to pass a pointer to a function to an algorithmic template (Clause 25), the interface is specified to accept a function object. This not only makes algorithmic templates work with pointers to functions, but also enables them to work with arbitrary function objects.

#### Header \texttt{<functional>} synopsis

```cpp
namespace std {
    // 20.9.3, invoke:
    template <class F, class... Args> result_of_t<F&&(Args&&...)> invoke(F&& f, Args&&... args);

    // 20.9.4, 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;

    // 20.9.5, arithmetic operations:
    template <class T = void> struct plus;
    template <class T = void> struct minus;
    template <class T = void> struct multiplies;
    template <class T = void> struct divides;
    template <class T = void> struct modulus;
    template <class T = void> struct negate;
    template <> struct plus<void>;
    template <> struct minus<void>;
    template <> struct multiplies<void>;
    template <> struct divides<void>;
    template <> struct modulus<void>;
    template <> struct negate<void>;

    // 20.9.6, 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>;
```

230) Such a type is a function pointer or a class type which has a member \texttt{operator()} or a class type which has a conversion to a pointer to function.
template <> struct greater<void>;
template <> struct less<void>;
template <> struct greater_equal<void>;
template <> struct less_equal<void>;

// 20.9.7, logical operations:
template <class T = void> struct logical_and;
template <class T = void> struct logical_or;
template <class T = void> struct logical_not;
template <> struct logical_and<void>;
template <> struct logical_or<void>;
template <> struct logical_not<void>;

// 20.9.8, bitwise operations:
template <class T = void> struct bit_and;
template <class T = void> struct bit_or;
template <class T = void> struct bit_xor;
template <class T = void> struct bit_not;
template <> struct bit_and<void>;
template <> struct bit_or<void>;
template <> struct bit_xor<void>;
template <> struct bit_not<void>;

// 20.9.9, negators:
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&);

// 20.9.10, 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
    extern unspecified _1;
    extern unspecified _2;
    
    extern unspecified _M;
}

// 20.9.11, member function adaptors:
template<class R, class T> unspecified mem_fn(R T::*);

// 20.9.12 polymorphic function wrappers:
class bad_function_call;
template<
class> class function; // undefined
template<
class R, class... ArgTypes> class function<R(ArgTypes...)>; 

template<
class R, class... ArgTypes>
 void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&);

template<
class R, class... ArgTypes>
 bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;
template<
class R, class... ArgTypes>
 bool operator==(nullptr_t, const function<R(ArgTypes...)>&) noexcept;
template<
class R, class... ArgTypes>
 bool operator!=(const function<R(ArgTypes...)>&, nullptr_t) noexcept;
template<
class R, class... ArgTypes>
 bool operator!=(nullptr_t, const function<R(ArgTypes...)>&) noexcept;

// 20.9.13, hash function primary template:
template <class T> struct hash;

// Hash function specializations
 template <> struct hash<bool>;
 template <> struct hash<char>;
 template <> struct hash<signed char>;
 template <> struct hash<unsigned char>;
 template <> struct hash<short>;
 template <> struct hash<unsigned short>;
 template <> struct hash<int>;
 template <> struct hash<unsigned int>;
 template <> struct hash<long>;
 template <> struct hash<unsigned long>;

 template <> struct hash<float>;
 template <> struct hash<double>;
 template <> struct hash<long double>;

 template<
class T> struct hash<T*>;

3 [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]

4 [Example: To negate every element of a:
   transform(a.begin(), a.end(), a.begin(), negate<double>());
   — end example]

5 [Note: To enable adaptors and other components to manipulate function objects that take one or two arguments many of the function objects in this clause correspondingly provide typedefs argument_type and

§ 20.9
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. — end note]

20.9.1 Definitions

The following definitions apply to this Clause:

1. A call signature is the name of a return type followed by a parenthesized comma-separated list of zero or more argument types.

2. A callable type is a function object type (20.9) or a pointer to member.

3. A callable object is an object of a callable type.

4. A call wrapper type is a type that holds a callable object and supports a call operation that forwards to that object.

5. A call wrapper is an object of a call wrapper type.

6. A target object is the callable object held by a call wrapper.

20.9.2 Requirements

1. Define INVOKE(f, t1, t2, ..., tN) as follows:

   (1.1) — (t1.*f)(t2, ..., tN) when f is a pointer to a member function of a class T and t1 is an object of type T or a reference to an object of type T or a reference to an object of a type derived from T;

   (1.2) — (f(t1).*f)(t2, ..., tN) when f is a pointer to a member function of a class T and t1 is not one of the types described in the previous item;

   (1.3) — t1.*f when N == 1 and f is a pointer to member data of a class T and t1 is an object of type T or a reference to an object of type T or a reference to an object of a type derived from T;

   (1.4) — (*t1).*f when N == 1 and f is a pointer to member data of a class T and t1 is not one of the types described in the previous item;

   (1.5) — f(t1, t2, ..., tN) in all other cases.

2. Define INVOKE(f, t1, t2, ..., tN, R) as INVOKE(f, t1, t2, ..., tN) implicitly converted to R.

3. If a call wrapper (20.9.1) has a weak result type the type of its member type result_type is based on the type T of the wrapper’s target object (20.9.1):

   (3.1) — if T is a pointer to function type, result_type shall be a synonym for the return type of T;

   (3.2) — if T is a pointer to member function, result_type shall be a synonym for the return type of T;

   (3.3) — if T is a class type and the qualified-id T::result_type is valid and denotes a type (14.8.2), then result_type shall be a synonym for T::result_type;

   (3.4) — otherwise result_type shall not be defined.

4. Every call wrapper (20.9.1) shall be MoveConstructible. A simple call wrapper is a call wrapper that is CopyConstructible and CopyAssignable and whose copy constructor, move constructor, and assignment operator do not throw exceptions. 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. [ Note: In a typical implementation forwarding call wrappers have an overloaded function call operator of the form

§ 20.9.2
template<class... UnBoundArgs>
R operator()(UnBoundArgs&&... unbound_args) cv-qual;

— end note]

20.9.3 Function template invoke

template <class F, class... Args>
result_of_t<F&&(Args&&...)>> invoke(F&& f, Args&&... args);

1 Returns: INVOCET (std::forward<F>f), std::forward<Args> (args)... (20.9.2).

20.9.4 Class template reference_wrapper

namespace std {
 template <class T> class reference_wrapper {
 public:
  // types
  typedef T type;
  typedef see below result_type;     // not always defined
  typedef see below argument_type;   // not always defined
  typedef see below first_argument_type;  // not always defined
  typedef see below second_argument_type;  // not always defined

  // construct/copy/destroy
  reference_wrapper(T&) noexcept;
  reference_wrapper(T&&) = delete; // do not bind to temporary objects
  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>
  result_of_t<T&(ArgTypes&&...)>
  operator() (ArgTypes&&...) const;
};
}

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 (3.9).

3 reference_wrapper<T> has a weak result type (20.9.2). If T is a function type, result_type shall be a
synonym for the return type of T.

4 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:

(4.1) — a function type or a pointer to function type taking one argument of type T1

(4.2) — 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 (14.8.2); 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 (14.8.2); the type `T1` is `T::first_argument_type` and the type `T2` is `T::second_argument_type`.

### 20.9.4.1 `reference_wrapper` construct/copy/destroy

`reference_wrapper(T& t) noexcept;`

**Effects:** Constructs a `reference_wrapper` object that stores a reference to `t`.

`reference_wrapper(const reference_wrapper& x) noexcept;`

**Effects:** Constructs a `reference_wrapper` object that stores a reference to `x.get()`.

### 20.9.4.2 `reference_wrapper` assignment

`reference_wrapper& operator=(const reference_wrapper& x) noexcept;`

**Postconditions:** `*this` stores a reference to `x.get()`.

### 20.9.4.3 `reference_wrapper` access

`operator T& () const noexcept;`

**Returns:** The stored reference.

`T& get() const noexcept;`

**Returns:** The stored reference.

### 20.9.4.4 `reference_wrapper` invocation

```
template <class... ArgTypes>
operator() (ArgTypes&&... args) const;
```

**Returns:** `INVOKE(get(), std::forward<ArgTypes>(args)...). (20.9.2)`

**Remark:** `operator()` is described for exposition only. Implementations are not required to provide an actual `reference_wrapper::operator()`. Implementations are permitted to support `reference_wrapper` function invocation through multiple overloaded operators or through other means.

### 20.9.4.5 `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())`

§ 20.9.5
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());

20.9.5 Arithmetic operations

The library provides basic function object classes for all of the arithmetic operators in the language (5.6, 5.7).

template <class T = void> struct plus {
    constexpr T operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef T result_type;
};

operator() returns \(x + y\).

template <class T = void> struct minus {
    constexpr T operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef T result_type;
};

operator() returns \(x - y\).

template <class T = void> struct multiplies {
    constexpr T operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef T result_type;
};

operator() returns \(x \times y\).

template <class T = void> struct divides {
    constexpr T operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef T result_type;
};

operator() returns \(x / y\).

template <class T = void> struct modulus {
    constexpr T operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef T result_type;
};

operator() returns \(x \% y\).
template <class T = void> struct negate {
  constexpr T operator()(const T& x) const;
  typedef T argument_type;
  typedef T result_type;
};

operator() returns -x.

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));
  typedef unspecified is_transparent;
};

operator() returns std::forward<T>(t) + std::forward<U>(u).

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));
  typedef unspecified is_transparent;
};

operator() returns std::forward<T>(t) - std::forward<U>(u).

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));
  typedef unspecified is_transparent;
};

operator() returns std::forward<T>(t) * std::forward<U>(u).

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));
  typedef unspecified is_transparent;
};

operator() returns std::forward<T>(t) / std::forward<U>(u).

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));
  typedef unspecified is_transparent;
};

operator() returns std::forward<T>(t) % std::forward<U>(u).

template <> struct negate<void> {
  template <class T> constexpr auto operator()(T&& t) const
    -> decltype(-std::forward<T>(t));
  typedef unspecified is_transparent;
};

operator() returns -std::forward<T>(t).
The library provides basic function object classes for all of the comparison operators in the language (5.9, 5.10).

```cpp
1 template <class T = void> struct equal_to {
    constexpr bool operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef bool result_type;
};

2     operator() returns x == y.

```n

```cpp
3 template <class T = void> struct not_equal_to {
    constexpr bool operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef bool result_type;
};

4     operator() returns x != y.

```n

```cpp
5 template <class T = void> struct greater {
    constexpr bool operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef bool result_type;
};

6     operator() returns x > y.

```n

```cpp
7 template <class T = void> struct less {
    constexpr bool operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef bool result_type;
};

8     operator() returns x < y.

```n

```cpp
9 template <class T = void> struct greater_equal {
    constexpr bool operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef bool result_type;
};

10    operator() returns x >= y.

```n

```cpp
11 template <class T = void> struct less_equal {
    constexpr bool operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef bool result_type;
};

12   operator() returns x <= y.

```n
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));

typedef unspecified is_transparent;
};

operator() returns std::forward<T>(t) == std::forward<U>(u).

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));

typedef unspecified is_transparent;
};

operator() returns std::forward<T>(t) != std::forward<U>(u).

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));

typedef unspecified is_transparent;
};

operator() returns std::forward<T>(t) > std::forward<U>(u).

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));

typedef unspecified is_transparent;
};

operator() returns std::forward<T>(t) < std::forward<U>(u).

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));

typedef unspecified is_transparent;
};

operator() returns std::forward<T>(t) >= std::forward<U>(u).

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));

typedef unspecified is_transparent;
};

operator() returns std::forward<T>(t) <= std::forward<U>(u).

For templates greater, less, greater_equal, and less_equal, the specializations for any pointer type
yield a total order, even if the built-in operators <, >, <=, => do not.

§ 20.9.6
20.9.7 Logical operations

The library provides basic function object classes for all of the logical operators in the language (5.14, 5.15, 5.3.1).

```cpp
template <class T = void> struct logical_and {
    constexpr bool operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef bool result_type;
};

operator() returns x && y.
```

```cpp
template <class T = void> struct logical_or {
    constexpr bool operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef bool result_type;
};

operator() returns x || y.
```

```cpp
template <class T = void> struct logical_not {
    constexpr bool operator()(const T& x) const;
    typedef T argument_type;
    typedef bool result_type;
};

operator() returns !x.
```

```cpp
template <> struct logical_and<void> {
    template <class T, class U> constexpr auto operator()(T&& t, U&& u) const -> decltype(std::forward<T>(t) && std::forward<U>(u));
    typedef unspecified is_transparent;
};

operator() returns std::forward<T>(t) && std::forward<U>(u).
```

```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));
    typedef unspecified is_transparent;
};

operator() returns std::forward<T>(t) || std::forward<U>(u).
```

```cpp
template <> struct logical_not<void> {
    template <class T> constexpr auto operator()(T&& t) const -> decltype(!std::forward<T>(t));
    typedef unspecified is_transparent;
};

operator() returns !std::forward<T>(t).
```
20.9.8 Bitwise operations

The library provides basic function object classes for all of the bitwise operators in the language (5.11, 5.13, 5.12, 5.3.1).

```cpp
template <class T = void> struct bit_and {
    constexpr T operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef T result_type;
};

operator() returns x & y.

template <class T = void> struct bit_or {
    constexpr T operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef T result_type;
};

operator() returns x | y.

template <class T = void> struct bit_xor {
    constexpr T operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef T result_type;
};

operator() returns x ^ y.

template <class T = void> struct bit_not {
    constexpr T operator()(const T& x) const;
    typedef T argument_type;
    typedef T result_type;
};

operator() returns ~x.
```

§ 20.9.8 593
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));
    typedef unspecified is_transparent;
};

operator() returns std::forward<T>(t) ^ std::forward<U>(u).

template <> struct bit_not<void> {
    template <class T> constexpr auto operator()(T&& t) const
                    -> decltype(~std::forward<T>(t));
    typedef unspecified is_transparent;
};

operator() returns ~std::forward<T>(t).

20.9.9 Negators

Negators not1 and not2 take a unary and a binary predicate, respectively, and return their complements (5.3.1).

template <class Predicate>
    class unary_negate {
    public:
        constexpr explicit unary_negate(const Predicate& pred);
        constexpr bool operator()(const typename Predicate::argument_type& x) const;
        typedef typename Predicate::argument_type argument_type;
        typedef bool result_type;
    };  
    operator() returns !pred(x).

template <class Predicate>
    constexpr unary_negate<Predicate> not1(const Predicate& pred);

Returns: unary_negate<Predicate>(pred).

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;
        typedef typename Predicate::first_argument_type first_argument_type;
        typedef typename Predicate::second_argument_type second_argument_type;
        typedef bool result_type;
    };  
    operator() returns !pred(x,y).

template <class Predicate>
    constexpr binary_negate<Predicate> not2(const Predicate& pred);

Returns: binary_negate<Predicate>(pred).

20.9.10 Function object binders

This subclause describes a uniform mechanism for binding arguments of callable objects.
20.9.10.1 Class template is_bind_expression

namespace std {
    template<class T> struct is_bind_expression; // see below
}

1 is_bind_expression can be used to detect function objects generated by bind. bind uses is_bind_expression to detect subexpressions.

2 Instantiations of the is_bind_expression template shall meet the UnaryTypeTrait requirements (20.10.1). The implementation shall provide a definition that has a BaseCharacteristic of true_type if T is a type returned from bind, otherwise it shall have a BaseCharacteristic of false_type. A program may specialize this template for a user-defined type T to have a BaseCharacteristic of true_type to indicate that T should be treated as a subexpression in a bind call.

20.9.10.2 Class template is_placeholder

namespace std {
    template<class T> struct is_placeholder; // see below
}

1 is_placeholder can be used to detect the standard placeholders _1, _2, and so on. bind uses is_placeholder to detect placeholders.

2 Instantiations of the is_placeholder template shall meet the UnaryTypeTrait requirements (20.10.1). The implementation shall provide a definition that has the BaseCharacteristic of integral_constant<int, J> if T is the type of std::placeholders::_J, otherwise it shall have a BaseCharacteristic of integral_constant<int, 0>. A program may specialize this template for a user-defined type T to have a BaseCharacteristic of integral_constant<int, N> with N > 0 to indicate that T should be treated as a placeholder type.

20.9.10.3 Function template bind

In the text that follows, the following names have the following meanings:

(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) — Ti is the i'th type in the template parameter pack BoundArgs,
(1.4) — TiD is the type decay_t<Ti>,
(1.5) — ti is the i'th argument in the function parameter pack bound_args,
(1.6) — tid is an lvalue of type TiD constructed from std::forward<Ti>(ti),
(1.7) — Uj is the j'th deduced type of the UnBoundArgs&&... parameter of the forwarding call wrapper, and
(1.8) — uj is the j'th argument associated with Uj.

template<class F, class... BoundArgs>
    unspecified bind(F&& f, BoundArgs&&... bound_args);

2 Requires: is_constructible<FD, F>::value shall be true. For each Ti in BoundArgs, is_constructible<TiD, Ti>::value shall be true. INVOKE (fd, w1, w2, ..., wN) (20.9.2) shall be a valid expression for some values w1, w2, ..., wN, where N == sizeof...(bound_args).

3 Returns: A forwarding call wrapper g with a weak result type (20.9.2). The effect of g(u1, u2, ..., uM) shall be INVOKE(fd, std::forward<V1>(v1), std::forward<V2>(v2), ..., std::forward<VN>(vN),
The values of the `bound arguments` $v_1$, $v_2$, ..., $v_N$ and their corresponding types $V_1$, $V_2$, ..., $V_N$ depend on the types $T_iD$ derived from the call to `bind` and the `cv`-qualifiers $cv$ of the call wrapper $g$ as follows:

1. If $T_iD$ is `reference_wrapper<T>`, the argument is `tid.get()` and its type $V_i$ is $T$&;
2. If the value of `is_bind_expression<TiD>::value` is true, the argument is `tid(std::forward<Uj>(uj)...)` and its type $V_i$ is `result_of_t<TiD cv & (Uj&&...)>&&`;
3. If the value $j$ of `is_placeholder<TiD>::value` is not zero, the argument is `std::forward<Uj>(uj)` and its type $V_i$ is $Uj&&$;
4. Otherwise, the value is `tid` and its type $V_i$ is $T_iD$ cv &.

### 20.9.10.4 Placeholders

```cpp
namespace std::placeholders {
    // M is the implementation-defined number of placeholders
    extern unspecified _1;
    extern unspecified _2;
    ...
    extern unspecified _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.

### 20.9.11 Function template `mem_fn`

```
template<class R, class T> unspecified mem_fn(R T::* pm);
```

1. **Returns:** A simple call wrapper (20.9.1) `fn` such that the expression `fn(t, a2, ..., aN)` is equivalent to `INVOKE` `(pm, t, a2, ..., aN)` (20.9.2). `fn` 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.

2. The simple call wrapper 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.

3. The simple call wrapper 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.

4. **Throws:** Nothing.

### 20.9.12 Polymorphic function wrappers

This subclause describes a polymorphic wrapper class that encapsulates arbitrary callable objects.

#### 20.9.12.1 Class `bad_function_call`

```
namespace std {
    class bad_function_call : public std::exception {
        public:
            // 20.9.12.1.1, constructor:
            bad_function_call() noexcept;
    }
}
```

1. **bad_function_call constructor**

```
bad_function_call() noexcept;
```

1. **Effects:** constructs a `bad_function_call` object.

2. **Postconditions:** `what()` returns an implementation-defined NTBS.

#### 20.9.12.2 Class template `function`

```
namespace std {
    template<class> class function; // undefined
    template<class R, class... ArgTypes>
    class function<R(ArgTypes...)>
    {
        public:
            typedef R result_type;
            typedef T1 argument_type; // only if sizeof...(ArgTypes) == 1 and
                                      // the type in ArgTypes is T1
    }
}
```

§ 20.9.12.2
typedef T1 first_argument_type;  // only if sizeof...(ArgTypes) == 2 and // ArgTypes contains T1 and T2
typedef T2 second_argument_type;  // only if sizeof...(ArgTypes) == 2 and // ArgTypes contains T1 and T2

// 20.9.12.2.1, construct/copy/destroy:
function() noexcept;
function(nullptr_t) noexcept;
function(const function&);
function(function&&);
template<class F> function(F);
template<class A> function(allocator_arg_t, const A&) noexcept;
template<class A> function(allocator_arg_t, const A&, nullptr_t) noexcept;
template<class A> function(allocator_arg_t, const A&, const function&);
template<class A> function(allocator_arg_t, const A&, function&&);
template<class F, class A> function(allocator_arg_t, const A&, const F);

function& operator=(const function&);
function& operator=(function&&);
function& operator=(nullptr_t) noexcept;
template<class F> function& operator=(F&&);
template<class F> function& operator=(reference_wrapper<F>) noexcept;

~function();

// 20.9.12.2.2, function modifiers:
void swap(function&) noexcept;
template<class F, class A> void assign(F&, const A&);

// 20.9.12.2.3, function capacity:
explicit operator bool() const noexcept;

// 20.9.12.2.4, function invocation:
R operator()(ArgTypes...) const;

// 20.9.12.2.5, function target access:
const std::type_info& target_type() const noexcept;
template<class T> T* target() noexcept;
template<class T> const T* target() const noexcept;

};

// 20.9.12.2.6, Null pointer comparisons:
template <class R, class... ArgTypes>
bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;

template <class R, class... ArgTypes>
bool operator==(nullptr_t, const function<R(ArgTypes...)>&) noexcept;

template <class R, class... ArgTypes>
bool operator!=(const function<R(ArgTypes...)>&, nullptr_t) noexcept;

§ 20.9.12.2
template <class R, class... ArgTypes>
bool operator!=(nullptr_t, const function<R(ArgTypes...)>&) noexcept;

// 20.9.12.2.7, specialized algorithms:
template <class R, class... ArgTypes>
void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&);

template<class R, class... ArgTypes, class Alloc>
struct uses_allocator<function<R(ArgTypes...)>, Alloc>
  : true_type { };
Requires: F shall be CopyConstructible.

Remarks: These constructors shall not participate in overload resolution unless f is Callable (20.9.12.2) for argument types ArgTypes... and return type R.

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 function class template, and !f.

Otherwise, *this targets a copy of f initialized with std::move(f). [Note: Implementations are encouraged to 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]

Throws: shall not throw exceptions when f is a function pointer or a reference_wrapper<T> for some T. Otherwise, may throw bad_alloc or any exception thrown by F’s copy or move constructor.

function& operator=(const function& f);

Effects: function(f).swap(*this);

Returns: *this

function& operator=(function&& f);

Effects: Replaces the target of *this with the target of f.

Returns: *this

function& operator=(nullptr_t noexcept);

Effects: If *this != nullptr, destroys the target of this.

Postconditions: !(*this).

Returns: *this

template<class F> function& operator=(F&& f);

Effects: function(std::forward<F>(f)).swap(*this);

Returns: *this

Remarks: This assignment operator shall not participate in overload resolution unless declval.typename decay<F>::type() is Callable (20.9.12.2) for argument types ArgTypes... and return type R.

template<class F> function& operator=(reference_wrapper<F> f) noexcept;

Effects: function(f).swap(*this);

Returns: *this

~function();

Effects: If *this != nullptr, destroys the target of this.
20.9.12.2 function modifiers

void swap(function& other) noexcept;

Effects: interchanges the targets of *this and other.

template<class F, class A>
void assign(F&& f, const A& a);

Effects: function(allocation_arg, a, std::forward<F>(f)).swap(*this)

20.9.12.3 function capacity

explicit operator bool() const noexcept;

Returns: true if *this has a target, otherwise false.

20.9.12.4 function invocation

R operator()(ArgTypes... args) const

Effects: INVOKE(f, std::forward<ArgTypes>(args)...) (20.9.2), where f is the target object (20.9.1) of *this.

Returns: Nothing if R is void, otherwise the return value of INVOKE(f, std::forward<ArgTypes>(args)...) R.

Throws: bad_function_call if !*this; otherwise, any exception thrown by the wrapped callable object.

20.9.12.5 function target access

const std::type_info& target_type() const noexcept;

Returns: If *this has a target of type T, typeid(T); otherwise, typeid(void).

template<class T>
T* target() noexcept;
template<class T> const T* target() const noexcept;

Requires: T shall be a type that is Callable (20.9.12.2) for parameter types ArgTypes and return type R.

Returns: If target_type() == typeid(T) a pointer to the stored function target; otherwise a null pointer.

20.9.12.6 null pointer comparison operators

template <class R, class... ArgTypes>
bool operator==(const function<R(ArgTypes...)>& f, nullptr_t) noexcept;
template <class R, class... ArgTypes>
bool operator==(nullptr_t, const function<R(ArgTypes...)>& f) noexcept;

Returns: !f.

template <class R, class... ArgTypes>
bool operator!=(const function<R(ArgTypes...)>& f, nullptr_t) noexcept;
template <class R, class... ArgTypes>
bool operator!=(nullptr_t, const function<R(ArgTypes...)>& f) noexcept;

Returns: (!f).
20.9.12.2.7 specialized algorithms

```cpp
template<class R, class... ArgTypes>
void swap(function<R(ArgTypes...)>& f1, function<R(ArgTypes...)>& f2);
```

**Effects:** `f1.swap(f2);`

20.9.13 Class template hash

The unordered associative containers defined in 23.5 use specializations of the class template `hash` as the default hash function. For all object types `Key` for which there exists a specialization `hash<Key>`, and for all enumeration types (7.2) `Key`, the instantiation `hash<Key>` shall:

1. satisfy the Hash requirements (17.6.3.4), with `Key` as the function call argument type, the Default-Constructible requirements (Table 19), the CopyAssignable requirements (Table 23),
2. be swappable (17.6.3.2) for lvalues,
3. provide two nested types `result_type` and `argument_type` which shall be synonyms for `size_t` and `Key`, respectively,
4. satisfy the requirement that if `k1 == k2` is true, `h(k1) == h(k2)` is also true, where `h` is an object of type `hash<Key>` and `k1` and `k2` are objects of type `Key`;
5. satisfy the requirement that the expression `h(k)`, where `h` is an object of type `hash<Key>` and `k` is an object of type `Key`, shall not throw an exception unless `hash<Key>` is a user-defined specialization that depends on at least one user-defined type.

```cpp
template <> struct hash<bool>;
template <> struct hash<char>;
template <> struct hash<signed char>;
template <> struct hash<unsigned char>;
template <> struct hash<char16_t>;
template <> struct hash<char32_t>;
template <> struct hash<wchar_t>;
template <> struct hash<short>;
template <> struct hash<unsigned short>;
template <> struct hash<int>;
template <> struct hash<unsigned int>;
template <> struct hash<long>;
template <> struct hash<unsigned long>;
template <> struct hash<long long>;
template <> struct hash<unsigned long long>;
template <> struct hash<float>;
template <> struct hash<double>;
template <> struct hash<long double>;
template <> struct hash<T*>;
```

The template specializations shall meet the requirements of class template `hash` (20.9.13).

20.10 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.
20.10.1 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 *BaseCharacteristic*, which is a specialization of the template `integral_constant` (20.10.3), with the arguments to the template `integral_constant` determined by the requirements for the particular property being described. The member names of the *BaseCharacteristic* 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 *BaseCharacteristic*, which is a specialization of the template `integral_constant` (20.10.3), with the arguments to the template `integral_constant` determined by the requirements for the particular relationship being described. The member names of the *BaseCharacteristic* 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.

20.10.2 Header `<type_traits>` synopsis

```cpp
namespace std {

    // 20.10.3, helper class:
    template <class T, T v> struct integral_constant;
    typedef integral_constant<bool, true> true_type;
    typedef integral_constant<bool, false> false_type;

    // 20.10.4.1, primary type categories:
    template <class T> struct is_void;
    template <class T> struct is_null_pointer;
    template <class T> struct is_integral;
    template <class T> struct is_floating_point;
    template <class T> struct is_array;
    template <class T> struct is_pointer;
    template <class T> struct is_lvalue_reference;
    template <class T> struct is_rvalue_reference;
    template <class T> struct is_member_object_pointer;
    template <class T> struct is_member_function_pointer;
    template <class T> struct is_enum;
    template <class T> struct is_union;
    template <class T> struct is_class;
    template <class T> struct is_function;

    // 20.10.4.2, composite type categories:
    template <class T> struct is_reference;
    template <class T> struct is_arithmetic;
    template <class T> struct is_fundamental;
    template <class T> struct is_object;
    template <class T> struct is_scalar;
    template <class T> struct is_compound;
    template <class T> struct is_member_pointer;

    // 20.10.4.3, 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_pod;
template <class T> struct is_literal_type;
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_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_copy_assignable;
template <class T> struct is_move_assignable;

template <class T> struct is_destructible;

template <class T, class... Args> struct is_trivially_constructible;
template <class T> struct is_trivially_default_constructible;
template <class T> struct is_trivially_copy_constructible;
template <class T> struct is_trivially_move_constructible;

template <class T, class U> struct is_trivially_assignable;
template <class T> struct is_trivially_copy_assignable;
template <class T> struct is_trivially_move_assignable;
template <class T> struct is_trivially_destructible;

template <class T, class... Args> struct is_nothrow_constructible;
template <class T> struct is_nothrow_default_constructible;
template <class T> struct is_nothrow_copy_constructible;
template <class T> struct is_nothrow_move_constructible;

template <class T, class U> struct is_nothrow_assignable;
template <class T> struct is_nothrow_copy_assignable;
template <class T> struct is_nothrow_move_assignable;

template <class T> struct is_nothrow_destructible;
template <class T> struct has_virtualDestructor;

// 20.10.5, type property queries:
template <class T> struct alignment_of;
template <class T> struct rank;
template <class T, unsigned I = 0> struct extent;

// 20.10.6, 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;

// 20.10.7.1, const-volatile modifications:
template <class T> struct remove_const;
template <class T> struct remove_volatile;
template <class T> struct remove_cv;
template <class T> struct add_const;
template <class T> struct add_volatile;
template <class T> struct add_cv;

template <class T>
  using remove_const_t = typename remove_const<T>::type;
template <class T>
  using remove_volatile_t = typename remove_volatile<T>::type;
template <class T>
  using remove_cv_t = typename remove_cv<T>::type;
template <class T>
  using add_const_t = typename add_const<T>::type;
template <class T>
  using add_volatile_t = typename add_volatile<T>::type;
template <class T>
  using add_cv_t = typename add_cv<T>::type;

// 20.10.7.2, 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;

// 20.10.7.3, sign modifications:
template <class T> struct make_signed;
template <class T> struct make_unsigned;

template <class T>
  using make_signed_t = typename make_signed<T>::type;
template <class T>
  using make_unsigned_t = typename make_unsigned<T>::type;

// 20.10.7.4, array modifications:
template <class T> struct remove_extent;
template <class T> struct remove_all_extents;

template <class T>
  using remove_extent_t = typename remove_extent<T>::type;
template <class T>
  using remove_all_extents_t = typename remove_all_extents<T>::type;

// 20.10.7.5, 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;

// 20.10.7.6, other transformations:

// see 20.10.7.6

// namespace std

1 The behavior of a program that adds specializations for any of the class templates defined in this subclause is undefined unless otherwise specified.

20.10.3 Helper classes [meta.help]

namespace std {

    template <class T, T v>
    struct integral_constant {
        static constexpr T value = v;
        typedef T value_type;
        typedef integral_constant<T,v> type;
        constexpr operator value_type() const noexcept { return value; }
        constexpr value_type operator()() const noexcept { return value; }
    };
}

§ 20.10.3
typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;
}

1 The class template integral_constant and its associated typedefs true_type and false_type are used as base classes to define the interface for various type traits.

20.10.4 Unary type traits

1 This sub-clause contains templates that may be used to query the properties of a type at compile time.

2 Each of these templates shall be a UnaryTypeTrait (20.10.1) with a BaseCharacteristic of true_type if the corresponding condition is true, otherwise false_type.

20.10.4.1 Primary type categories

1 The primary type categories correspond to the descriptions given in section 3.9 of the C++ standard.

2 For any given type T, the result of applying one of these templates to T and to cv-qualified T shall yield the same result.

3 [Note: For any given type T, exactly one of the primary type categories has a value member that evaluates to true. — end note]

Table 47 — Primary type category predicates

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt; struct is_void;</td>
<td>T is void</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_null_pointer;</td>
<td>T is std::nullptr_t (3.9.1)</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_integral;</td>
<td>T is an integral type (3.9.1)</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_floating_point;</td>
<td>T is a floating point type (3.9.1)</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_array;</td>
<td>T is an array type (3.9.2) of known or unknown extent</td>
<td>Class template array (23.3.2) is not an array type.</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_pointer;</td>
<td>T is a pointer type (3.9.2)</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 (8.3.2)</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_rvalue_reference;</td>
<td>T is an rvalue reference type (8.3.2)</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_member_object_pointer;</td>
<td>T is a pointer to non-static 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 non-static member function</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_enum;</td>
<td>T is an enumeration type (3.9.2)</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_union;</td>
<td>T is a union type (3.9.2)</td>
<td></td>
</tr>
</tbody>
</table>
Table 47 — Primary type category predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt;</td>
<td>T is a class type but not a union type (3.9.2)</td>
<td></td>
</tr>
<tr>
<td>struct is_class;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>T is a function type (3.9.2)</td>
<td></td>
</tr>
<tr>
<td>struct is_function;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

20.10.4.2 Composite type traits [meta.unary.comp]

1 These templates provide convenient compositions of the primary type categories, corresponding to the descriptions given in section 3.9.

2 For any given type T, the result of applying one of these templates to T, and to cv-qualified T shall yield the same result.

Table 48 — Composite type category predicates

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt;</td>
<td>T is an lvalue reference or an rvalue reference</td>
<td></td>
</tr>
<tr>
<td>struct is_reference;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>T is an arithmetic type (3.9.1)</td>
<td></td>
</tr>
<tr>
<td>struct is_arithmetic;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>T is a fundamental type (3.9.1)</td>
<td></td>
</tr>
<tr>
<td>struct is_fundamental;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>T is an object type (3.9)</td>
<td></td>
</tr>
<tr>
<td>struct is_object;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>T is a scalar type (3.9)</td>
<td></td>
</tr>
<tr>
<td>struct is_scalar;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>T is a compound type (3.9.2)</td>
<td></td>
</tr>
<tr>
<td>struct is_compound;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>T is a pointer to non-static data member or non-static member function</td>
<td></td>
</tr>
<tr>
<td>struct is_member_pointer;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

20.10.4.3 Type properties [meta.unary.prop]

1 These templates provide access to some of the more important properties of types.

2 It is unspecified whether the library defines any full or partial specializations of any of these templates.

3 For all of the class templates X declared in this Clause, 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 must be a complete type.

Table 49 — 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 (3.9.3)</td>
<td></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 (3.9.3)</td>
<td></td>
</tr>
<tr>
<td>struct is_volatile;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 49 — Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>template <code>&lt;class T&gt;</code> struct is_trivial;</td>
<td>( T ) is a trivial type (3.9)</td>
<td><code>remove_all_extents_t&lt;T&gt;</code> shall be a complete type or (possibly cv-qualified) <code>void</code>.</td>
</tr>
<tr>
<td>template <code>&lt;class T&gt;</code> struct is_trivially_copyable;</td>
<td>( T ) is a trivially copyable type (3.9)</td>
<td><code>remove_all_extents_t&lt;T&gt;</code> shall be a complete type or (possibly cv-qualified) <code>void</code>.</td>
</tr>
<tr>
<td>template <code>&lt;class T&gt;</code> struct is_standard_layout;</td>
<td>( T ) is a standard-layout type (3.9)</td>
<td><code>remove_all_extents_t&lt;T&gt;</code> shall be a complete type or (possibly cv-qualified) <code>void</code>.</td>
</tr>
<tr>
<td>template <code>&lt;class T&gt;</code> struct is_pod;</td>
<td>( T ) is a POD type (3.9)</td>
<td><code>remove_all_extents_t&lt;T&gt;</code> shall be a complete type or (possibly cv-qualified) <code>void</code>.</td>
</tr>
<tr>
<td>template <code>&lt;class T&gt;</code> struct is_literal_type;</td>
<td>( T ) is a literal type (3.9)</td>
<td><code>remove_all_extents_t&lt;T&gt;</code> shall be a complete type or (possibly cv-qualified) <code>void</code>.</td>
</tr>
<tr>
<td>template <code>&lt;class T&gt;</code> struct is_empty;</td>
<td>( T ) is a class type, but not a 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 <code>is_empty&lt;B&gt;::value</code> is false.</td>
<td>If ( T ) is a non-union class type, ( T ) shall be a complete type.</td>
</tr>
<tr>
<td>template <code>&lt;class T&gt;</code> struct is_polymorphic;</td>
<td>( T ) is a polymorphic class (10.3)</td>
<td>If ( T ) is a non-union class type, ( T ) shall be a complete type.</td>
</tr>
<tr>
<td>template <code>&lt;class T&gt;</code> struct is_abstract;</td>
<td>( T ) is an abstract class (10.4)</td>
<td>If ( T ) is a non-union class type, ( T ) shall be a complete type.</td>
</tr>
<tr>
<td>template <code>&lt;class T&gt;</code> struct is_final;</td>
<td>( T ) is a class type marked with the <code>class-virt-specifier final</code> (Clause 9). [ Note: A union is a class type that can be marked with <code>final</code>. — end note ]</td>
<td>If ( T ) is a class type, ( T ) shall be a complete type.</td>
</tr>
</tbody>
</table>
Table 49 — Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;class T&gt; struct is_signed;</code></td>
<td>If <code>is_</code> arithmetic&lt;T&gt;::value is true, the same result as <code>integral_</code> constant&lt;bool, T(-1) &lt; T(0)&gt;::value; otherwise, false.</td>
<td></td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct is_unsigned;</code></td>
<td>If <code>is_</code> arithmetic&lt;T&gt;::value is true, the same result as <code>integral_</code> constant&lt;bool, T(0) &lt; T(-1)&gt;::value; otherwise, false.</td>
<td></td>
</tr>
<tr>
<td><code>template &lt;class T, class... Args&gt; struct is_constructible;</code></td>
<td>see below</td>
<td>T and all types in the parameter pack <code>Args</code> shall be complete types, (possibly cv-qualified) void, or arrays of unknown bound.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct is_default_constructible;</code></td>
<td><code>is_</code> constructible&lt;T&gt;::value is true.</td>
<td>T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct is_copy_constructible;</code></td>
<td>For a referenceable type T, the same result as `is_constructible&lt;T, const T&amp;&gt;::value, otherwise false.</td>
<td>T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct is_move_constructible;</code></td>
<td>For a referenceable type T, the same result as `is_constructible&lt;T, T&amp;&gt;::value, otherwise false.</td>
<td>T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
</tbody>
</table>
Table 49 — Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;class T, class U&gt;</code></td>
<td>The expression</td>
<td><code>T</code> and <code>U</code> shall be complete types, (possibly cv-qualified) <code>void</code>, or arrays of unknown bound.</td>
</tr>
<tr>
<td><code>struct is_assignable;</code></td>
<td><code>declval&lt;T&gt;() = declval&lt;U&gt;()</code> is well-formed when treated as an unevaluated operand (Clause 5). Access checking is performed as if in a context unrelated to <code>T</code> and <code>U</code>. Only the validity of the immediate context of the assignment expression is considered.</td>
<td><code>T</code> and <code>U</code> shall be complete types, (possibly cv-qualified) <code>void</code>, or arrays of unknown bound.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt;</code></td>
<td>For a referenceable type <code>T</code>, the same result as <code>is_assignable&lt;T&amp;&gt;, const T&amp;&gt;::value</code>, otherwise <code>false</code>.</td>
<td><code>T</code> shall be a complete type, (possibly <code>cv</code>-qualified) <code>void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td><code>struct is_copy_assignable;</code></td>
<td></td>
<td><code>T</code> shall be a complete type, (possibly <code>cv</code>-qualified) <code>void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt;</code></td>
<td>For a referenceable type <code>T</code>, the same result as <code>is_assignable&lt;T&amp;&gt;, T&amp;&amp;&gt;::value</code>, otherwise <code>false</code>.</td>
<td><code>T</code> shall be a complete type, (possibly <code>cv</code>-qualified) <code>void</code>, or an array of unknown bound.</td>
</tr>
</tbody>
</table>
Table 49 — Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>\template &lt;\class T&gt; \struct is_destructible;</td>
<td>For reference types, is_destructible&lt;T&gt;::value is true. For incomplete types and function types, is_destructible&lt;T&gt;::value is false. For object types and given ( U ) equal to remove_all_extents&lt;t&lt;&lt;T&gt;&gt;, if the expression std::declval&lt;U&amp;&gt;().-U() is well-formed when treated as an unevaluated operand (Clause 5), then is_destructible&lt;T&gt;::value is true, otherwise it is false.</td>
<td>( T ) shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>\template &lt;\class T, class... Args&gt; \struct is_trivially_constructible;</td>
<td>is_constructible&lt;T, Args...&gt;::value is true and the variable definition for is_constructible, as defined below, is known to call no operation that is not trivial (3.9, 12).</td>
<td>( T ) and all types in the parameter pack ( \text{Args} ) shall be complete types, (possibly cv-qualified) void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>\template &lt;\class T&gt; \struct is_trivially_default_constructible;</td>
<td>is_trivially_default_constructible&lt;T&gt;::value is true.</td>
<td>( T ) shall be a complete type, (possibly cv-qualified) 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&lt;T, const T&amp;&gt;::value, otherwise false.</td>
<td>( T ) shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>\template &lt;\class T&gt; \struct is_trivially_move_constructible;</td>
<td>For a referenceable type ( T ), the same result as is_trivially_constructible&lt;T, T&amp;&amp;&gt;::value, otherwise false.</td>
<td>( T ) shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
</tbody>
</table>
Table 49 — Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T, class U&gt; struct is_trivially_assignable;</td>
<td>is_assignable&lt;T, U&gt;::value is true and the assignment, as defined by is_assignable, is known to call no operation that is not trivial (3.9, 12).</td>
<td>T and U shall be complete types, (possibly cv-qualified) 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&lt;T&amp;, const T&amp;&gt;::value, otherwise false.</td>
<td>T shall be a complete type, (possibly cv-qualified) 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&lt;T&amp;, T&amp;&amp;&gt;::value, otherwise false.</td>
<td>T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_trivially_destructible;</td>
<td>is_destructible&lt;T&gt;::value is true and the indicated destructor is known to be trivial.</td>
<td>T shall be a complete type, (possibly cv-qualified) 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&lt;T, Args...&gt;::value is true and the variable definition for is_constructible, as defined below, is known not to throw any exceptions (5.3.7).</td>
<td>T and all types in the parameter pack Args shall be complete types, (possibly cv-qualified) void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_nothrow_default_constructible;</td>
<td>is_nothrow_default_constructible&lt;T&gt;::value is true.</td>
<td>T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_nothrow_copy_constructible;</td>
<td>For a referenceable type T, the same result as is_nothrow_constructible&lt;T, const T&amp;&gt;::value, otherwise false.</td>
<td>T shall be a complete type, (possibly cv-qualified) 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&lt;T, T&amp;&amp;&gt;::value, otherwise false.</td>
<td>T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
</tbody>
</table>
Table 49 — Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T, class U&gt; struct is_nothrow_assignable;</td>
<td>is_assignable&lt;T, U&gt;::value is true and the assignment is known not to throw any exceptions (5.3.7).</td>
<td>T and U shall be complete types, (possibly cv-qualified) 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&lt;T&amp;&amp;, const T&amp;&gt;::value, otherwise false.</td>
<td>T shall be a complete type, (possibly cv-qualified) 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&lt;T&amp; , T&amp;&amp;&gt;::value, otherwise false.</td>
<td>T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_nothrow_destructible;</td>
<td>is_destructible&lt;T&gt;::value is true and the indicated destructor is known not to throw any exceptions (5.3.7).</td>
<td>T shall be a complete type, (possibly cv-qualified) 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 (12.4)</td>
<td>If T is a non-union class type, T shall be a complete type.</td>
</tr>
</tbody>
</table>

---

4 [Example:]

is_const<const volatile int>::value // true
is_const<const int*>::value // false
is_const<const int&>::value // false
is_const<int[3]>::value // false
is_const<const int[3]>::value // true

—end example]

5 [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]

6 [Example:]

// Given:
struct P final { };
union U1 { };
union U2 final { };
// the following assertions hold:
static_assert(!is_final<int>::value, "Error!");
static_assert( is_final<P>::value, "Error!");
static_assert(!is_final<U1>::value, "Error!");
static_assert( is_final<U2>::value, "Error!");

— end example ]
7 Given the following function declaration:

```cpp
template <class T>
add_rvalue_reference_t<T> create() noexcept;
```

the predicate condition for a template specialization `is_constructible<T, Args...>` shall be satisfied if and only if the following variable definition would be well-formed for some invented variable `t`:

```cpp
T t(create<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]

20.10.5 Type property queries [meta.unary.prop.query]

This sub-clause contains templates that may be used to query properties of types at compile time.

Table 50 — Type property queries

<table>
<thead>
<tr>
<th>Template</th>
<th>Value</th>
</tr>
</thead>
</table>
| template <class T> struct alignment_of; | `alignof(T)`.
| Requires: `alignof(T)` shall be a valid expression (5.3.6) |
| template <class T> struct rank; | If `T` names an array type, an integer value representing the number of dimensions of `T`; otherwise, 0. |
| template <class T, unsigned I = 0> struct extent; | If `T` is not an array type, or if it has rank less than or equal to `I`, or if `I` is 0 and `T` has type “array of unknown bound of `U`”, then 0; otherwise, the bound (8.3.4) of the `I`th dimension of `T`, where indexing of `I` is zero-based |

Each of these templates shall be a UnaryTypeTrait (20.10.1) with a BaseCharacteristic of integral_constant<size_t, Value>.

[Example:

// the following assertions hold:
assert(rank<int>::value == 0);
assert(rank<int[2]>::value == 1);
assert(rank<int[][4]>::value == 2);

— end example ]

3 [Example:

// the following assertions hold:
assert(extent<int>::value == 0);
assert(extent<int[2]>::value == 2);
assert(extent<int[2][4]>::value == 2);
assert(extent<int[]>::value == 0);
assert((extent<int, 1>::value) == 0);
assert((extent<int[2], 1>::value) == 0);
assert((extent<int[2][4], 1>::value) == 4);
assert((extent<int[], 1>::value) == 4);

—end example

20.10.6 Relationships between types [meta.rel]

1 This sub-clause contains templates that may be used to query relationships between types at compile time.

2 Each of these templates shall be a BinaryTypeTrait (20.10.1) with a BaseCharacteristic of true_type if the corresponding condition is true, otherwise false_type.

Table 51 — Type relationship predicates

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T, class U&gt; struct is_same;</td>
<td>T and U name the same type with the same cv-qualifications</td>
<td></td>
</tr>
<tr>
<td>template &lt;class Base, class Derived&gt; struct is_base_of;</td>
<td>Base is a base class of Derived (Clause 10) without regard to cv-qualifiers or Base and Derived are not unions and name the same class type without regard to cv-qualifiers</td>
<td>If Base and Derived are non-union class types and are different types (ignoring possible cv-qualifiers) then Derived shall be a complete type. [Note: Base classes that are private, protected, or ambiguous are, nonetheless, base classes. —end note]</td>
</tr>
<tr>
<td>template &lt;class From, class To&gt; struct is_convertible;</td>
<td>see below</td>
<td>From and To shall be complete types, arrays of unknown bound, or (possibly cv-qualified) void types.</td>
</tr>
</tbody>
</table>

3 [Example:

struct B {};
struct B1 : B {};
struct B2 : B {};
struct D : private B1, private B2 {};

is_base_of<B, D>::value // true
is_base_of<const B, D>::value // true
is_base_of<B, const D>::value // true
is_base_of<B, const B>::value // false
is_base_of<D, B>::value // false
is_base_of<B&, D&>::value // false
is_base_of<B[3], D[3]>::value // false
is_base_of<int, int>::value // false

—end example]
Given the following function declaration:

```cpp
template <class T>
add_rvalue_reference_t<T> create() noexcept;
```

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 create<From>();
}
```

[Note: This requirement gives well defined results for reference types, void types, array types, and function types. — end note] Access checking is performed as if in a context unrelated to `To` and `From`. Only the validity of the immediate context of the expression of the `return-statement` (including conversions to the return type) is considered. [Note: The evaluation of the conversion can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the “immediate context” and can result in the program being ill-formed. — end note]

### 20.10.7 Transformations between types [meta.trans]

1. This sub-clause contains templates that may be used to transform one type to another following some predefined rule.

2. Each of the templates in this subclause shall be a `TransformationTrait` (20.10.1).

#### 20.10.7.1 Const-volatile modifications [meta.trans.cv]

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;class T&gt; struct remove_const;</code></td>
<td>The member typedef <code>type</code> shall name the same type as <code>T</code> except that any top-level const-qualifier has been removed. [Example:<code>remove_const_t&lt;const volatile int&gt;</code> evaluates to <code>volatile int</code>, whereas <code>remove_const_t&lt;const int*&gt;</code> evaluates to <code>const int*</code>. — end example]</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct remove_volatile;</code></td>
<td>The member typedef <code>type</code> shall name the same type as <code>T</code> except that any top-level volatile-qualifier has been removed. [Example:<code>remove_volatile_t&lt;const volatile int&gt;</code> evaluates to <code>const int</code>, whereas <code>remove_volatile_t&lt;volatile int*&gt;</code> evaluates to <code>volatile int*</code>. — end example]</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct remove_cv;</code></td>
<td>The member typedef <code>type</code> shall be the same as <code>T</code> except that any top-level cv-qualifier has been removed. [Example:<code>remove_cv_t&lt;const volatile int&gt;</code> evaluates to <code>int</code>, whereas <code>remove_cv_t&lt;const volatile int*&gt;</code> evaluates to <code>const volatile int*</code>. — end example]</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct add_const;</code></td>
<td>If <code>T</code> is a reference, function, or top-level const-qualified type, then <code>type</code> shall name the same type as <code>T</code>, otherwise <code>T const</code>.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct add_volatile;</code></td>
<td>If <code>T</code> is a reference, function, or top-level volatile-qualified type, then <code>type</code> shall name the same type as <code>T</code>, otherwise <code>T volatile</code>.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct add_cv;</code></td>
<td>The member typedef <code>type</code> shall name the same type as <code>add_const_t&lt;add_volatile_t&lt;T&gt;&gt;</code>.</td>
</tr>
</tbody>
</table>

§ 20.10.7.1
20.10.7.2 Reference modifications

Table 53 — 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 ( T_1 )” then the member typedef ( \text{type} ) shall name ( T_1 ); otherwise, ( \text{type} ) shall name ( T ).</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct add_lvalue_reference;</td>
<td>If ( T ) names an object or function type then the member typedef ( \text{type} ) shall name ( T ); otherwise, if ( T ) names a type “rvalue reference to ( T_1 )” then the member typedef ( \text{type} ) shall name ( T_1 ); otherwise, ( \text{type} ) shall name ( T ).</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct add_rvalue_reference;</td>
<td>If ( T ) names an object or function type then the member typedef ( \text{type} ) shall name ( T ); otherwise, ( \text{type} ) shall name ( T ). [Note: This rule reflects the semantics of reference collapsing (8.3.2). For example, when a type ( T ) names a type ( T_1 ), the type ( \text{add_rvalue_reference_t}&lt;T&gt; ) is not an rvalue reference. —end note]</td>
</tr>
</tbody>
</table>

20.10.7.3 Sign modifications

Table 54 — Sign modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt; struct make_signed;</td>
<td>If ( T ) names a (possibly cv-qualified) signed integer type (3.9.1) then the member typedef ( \text{type} ) shall name the type ( T ); otherwise, if ( T ) names a (possibly cv-qualified) unsigned integer type then ( \text{type} ) shall name the corresponding signed integer type, with the same cv-qualifiers as ( T ); otherwise, ( \text{type} ) shall name the signed integer type with smallest rank (4.13) for which ( \text{sizeof}(T) == \text{sizeof(type)} ), with the same cv-qualifiers as ( T ). Requires: ( T ) shall be a (possibly cv-qualified) integral type or enumeration but not a bool type.</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct make_unsigned;</td>
<td>If ( T ) names a (possibly cv-qualified) unsigned integer type (3.9.1) then the member typedef ( \text{type} ) shall name the type ( T ); otherwise, if ( T ) names a (possibly cv-qualified) signed integer type then ( \text{type} ) shall name the corresponding unsigned integer type, with the same cv-qualifiers as ( T ); otherwise, ( \text{type} ) shall name the unsigned integer type with smallest rank (4.13) for which ( \text{sizeof}(T) == \text{sizeof(type)} ), with the same cv-qualifiers as ( T ). Requires: ( T ) shall be a (possibly cv-qualified) integral type or enumeration but not a bool type.</td>
</tr>
</tbody>
</table>
20.10.7.4 Array modifications

Table 55 — Array modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt;</td>
<td>If T names a type “array of U”, the member typedef type shall be U, otherwise T. [Note: For multidimensional arrays, only the first array dimension is removed. For a type “array of const U”, the resulting type is const U. —end note]</td>
</tr>
<tr>
<td>struct remove_extent;</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>If T is “multi-dimensional array of U”, the resulting member typedef type is U, otherwise T.</td>
</tr>
<tr>
<td>struct remove_all_extents;</td>
<td></td>
</tr>
</tbody>
</table>

1 [Example]

```cpp
// the following assertions hold:
assert((is_same<remove_extent_t<int>, int>::value));
assert((is_same<remove_extent_t<int[2]>, int>::value));
assert((is_same<remove_extent_t<int[2][3]>, int[3]>::value));
assert((is_same<remove_extent_t<int[3]>, int[3]>::value));
```

— end example]

2 [Example]

```cpp
// the following assertions hold:
assert((is_same<remove_all_extents_t<int>, int>::value));
assert((is_same<remove_all_extents_t<int[2]>, int>::value));
assert((is_same<remove_all_extents_t<int[2][3]>, int[3]>::value));
assert((is_same<remove_all_extents_t<int[3]>, int[3]>::value));
```

— end example]

20.10.7.5 Pointer modifications

Table 56 — Pointer modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt;</td>
<td>If T has type “(possibly cv-qualified) pointer to T1” then the member typedef type shall name T1; otherwise, it shall name T.</td>
</tr>
<tr>
<td>struct remove_pointer;</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>The member typedef type shall name the same type as remove_reference_t&lt;T&gt;.*</td>
</tr>
<tr>
<td>struct add_pointer;</td>
<td></td>
</tr>
</tbody>
</table>
## 20.10.7.6 Other transformations

### Table 57 — Other transformations

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template <code>&lt;std::size_t Len,т logarithm</code>&lt;br&gt;<code>std::size_t Align</code>&lt;br&gt;<code>= default-alignment&gt;</code>&lt;br&gt;<code>struct aligned_storage;</code></td>
<td>Len shall not be zero. Align shall be equal to <code>alignof(T)</code> for some type T or to <code>default-alignment</code>.</td>
<td>The value of <code>default-alignment</code> shall be the most stringent alignment requirement for any C++ object type whose size is no greater than <code>Len</code> (3.9). The member typedef <code>type</code> shall be a POD type suitable for use as uninitialized storage for any object whose size is at most <code>Len</code> and whose alignment is a divisor of <code>Align</code>.</td>
</tr>
<tr>
<td>template <code>&lt;std::size_t Len, class... Types&gt;</code>&lt;br&gt;<code>struct aligned_union;</code></td>
<td>At least one type is provided.</td>
<td>The member typedef <code>type</code> shall be a POD type suitable for use as uninitialized storage for any object whose type is listed in <code>Types</code>; its size shall be at least <code>Len</code>. The static member <code>alignment_value</code> shall be an integral constant of type <code>std::size_t</code> whose value is the strictest alignment of all types listed in <code>Types</code>.</td>
</tr>
<tr>
<td>template <code>&lt;class T&gt;</code> <code>struct decay;</code></td>
<td>Let <code>U</code> be <code>remove_reference_t&lt;T&gt;</code>. If <code>is_array&lt;U&gt;::value</code> is true, the member typedef <code>type</code> shall equal <code>remove_extent_t&lt;U&gt;*</code>. If <code>is_function&lt;U&gt;::value</code> is true, the member typedef <code>type</code> shall equal <code>add_pointer_t&lt;U&gt;</code>. Otherwise the member typedef <code>type</code> equals <code>remove_cv_t&lt;U&gt;</code>. [Note: This behavior is similar to the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) conversions applied when an lvalue expression is used as an rvalue, but also strips <code>cv</code>-qualifiers from class types in order to more closely model by-value argument passing. — end note]</td>
<td></td>
</tr>
<tr>
<td>template <code>&lt;bool B, class T = void&gt;</code> <code>struct enable_if;</code></td>
<td>If <code>B</code> is <code>true</code>, the member typedef <code>type</code> shall equal <code>T</code>; otherwise, there shall be no member <code>type</code>.</td>
<td></td>
</tr>
<tr>
<td>template <code>&lt;bool B, class T, class F&gt;</code> <code>struct conditional;</code></td>
<td>If <code>B</code> is <code>true</code>, the member typedef <code>type</code> shall equal <code>T</code>. If <code>B</code> is <code>false</code>, the member typedef <code>type</code> shall equal <code>F</code>.</td>
<td></td>
</tr>
<tr>
<td>Template</td>
<td>Condition</td>
<td>Comments</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td><code>template &lt;class... T&gt; struct common_type;</code></td>
<td></td>
<td>The member typedef <code>type</code> shall be defined or omitted as specified below. If it is omitted, there shall be no member <code>type</code>. All types in the parameter pack T shall be complete or (possibly <code>cv</code>) void. A program may specialize this trait if at least one template parameter in the specialization is a user-defined type. [Note: Such specializations are needed when only explicit conversions are desired among the template arguments. — end note]</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct underlying_type;</code></td>
<td>T shall be a complete enumeration type (7.2)</td>
<td>The member typedef <code>type</code> shall name the underlying type of T.</td>
</tr>
<tr>
<td><code>template &lt;class Fn, class... ArgTypes&gt; struct result_of&lt;Fn(ArgTypes...)&gt;;</code></td>
<td>Fn and all types in the parameter pack <code>ArgTypes</code> shall be complete types, (possibly <code>cv</code>-qualified) void, or arrays of unknown bound. If the expression <code>INVOKE(declval&lt;Fn&gt;(), declval&lt;ArgTypes&gt;()...)</code> is well formed when treated as an unevaluated operand (Clause 5), the member typedef <code>type</code> shall name the type <code>decltype(INVOKE(declval&lt;Fn&gt;(), declval&lt;ArgTypes&gt;()...))</code>; otherwise, there shall be no member <code>type</code>. Access checking is performed as if in a context unrelated to <code>Fn</code> and <code>ArgTypes</code>. Only the validity of the immediate context of the expression is considered. [Note: The compilation of the expression can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the “immediate context” and can result in the program being ill-formed. — end note]</td>
<td></td>
</tr>
</tbody>
</table>

1 [Note: A typical implementation would define `aligned_storage` as:

```cpp
template <std::size_t Len, std::size_t Alignment>
struct aligned_storage {
    typedef struct {
        alignas(Alignment) unsigned char __data[Len];
    } type;
};
```
It is implementation-defined whether any extended alignment is supported (3.11).

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:

1. If `sizeof...(T)` is zero, there shall be no member `type`.
2. If `sizeof...(T)` is one, let `T0` denote the sole type in the pack `T`. The member typedef `type` shall denote the same type as `decay_t<T0>`.
3. If `sizeof...(T)` is greater than one, let `T1`, `T2`, and `R`, respectively, denote the first, second, and (pack of) remaining types comprising `T`. [Note: `sizeof...(R)` may be zero. — end note] Let `C` denote the type, if any, of an unevaluated conditional expression (5.16) whose first operand is an arbitrary value of type `bool`, whose second operand is an xvalue of type `T1`, and whose third operand is an xvalue of type `T2`. If there is such a type `C`, the member typedef `type` shall denote the same type, if any, as `common_type_t<C,R...>`. Otherwise, there shall be no member `type`.

Example: Given these definitions:
```cpp
typedef bool (&PF1)();
typedef short (*PF2)(long);

struct S {
    operator PF2() const;
    double operator() (char, int&);
    void fn(long) const;
    char data;
};
typedef void (S::*PMF)(long) const;
typedef char S::*PMD;
```
the following assertions will hold:
```cpp
static_assert(is_same<result_of_t<S(int)>, short>::value, "Error!");
static_assert(is_same<result_of_t<S&(unsigned char, int&)>, double>::value, "Error!");
static_assert(is_same<result_of_t<PF1()>, bool>::value, "Error!");
static_assert(is_same<result_of_t<PMF(unique_ptr<S>, int)>, void>::value, "Error!");
static_assert(is_same<result_of_t<PMD(S)>, char&>::value, "Error!");
static_assert(is_same<result_of_t<PMD(const S*)>, const char&>::value, "Error!");
```

--- end example ---

## 20.11 Compile-time rational arithmetic

This subclause describes the ratio library. It provides a class template `ratio` which exactly represents any finite rational number with a numerator and denominator representable by compile-time constants of type `intmax_t`.

Throughout this subclause, the names of template parameters are used to express type requirements. If a template parameter is named `R1` or `R2`, and the template argument is not a specialization of the `ratio` template, the program is ill-formed.

--- end note ---
20.11.2 Header <ratio> synopsis

namespace std {
    // 20.11.3, class template ratio
    template <intmax_t N, intmax_t D = 1> class ratio;

    // 20.11.4, ratio arithmetic
    template <class R1, class R2> using ratio_add = see below;
    template <class R1, class R2> using ratio_subtract = see below;
    template <class R1, class R2> using ratio_multiply = see below;
    template <class R1, class R2> using ratio_divide = see below;

    // 20.11.5, 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;

    // 20.11.6, convenience SI typedefs
    typedef ratio<1, 1'000'000'000'000'000'000'000> yocto; // see below
    typedef ratio<1, 1'000'000'000'000'000'000'000> zepto; // see below
    typedef ratio<1, 1'000'000'000'000'000'000> atto;
    typedef ratio<1, 1'000'000'000'000'000> femto;
    typedef ratio<1, 1'000'000'000'000> pico;
    typedef ratio<1, 1'000'000'000> nano;
    typedef ratio<1, 1'000'000> micro;
    typedef ratio<1, 100> milli;
    typedef ratio<1, 10> centi;
    typedef ratio<1, 1> deci;
    typedef ratio<10, 1> deca;
    typedef ratio<100, 1> hecto;
    typedef ratio<1'000, 1> kilo;
    typedef ratio<1'000'000, 1> mega;
    typedef ratio<1'000'000'000, 1> giga;
    typedef ratio<1'000'000'000'000, 1> tera;
    typedef ratio<1'000'000'000'000'000, 1> peta;
    typedef ratio<1'000'000'000'000'000'000, 1> exa;
    typedef ratio<1'000'000'000'000'000'000'000, 1> zetta; // see below
    typedef ratio<1'000'000'000'000'000'000'000'000, 1> yotta; // see below
}

20.11.3 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;
            typedef ratio<num, den> type;
    };
}

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`.

### 20.11.4 Arithmetic on ratios [ratio.arithmetic]

1 Each of the alias templates `ratio_add`, `ratio_subtract`, `ratio_multiply`, and `ratio_divide` denotes the result of an arithmetic computation on two `ratio` s `R1` and `R2`. With `X` and `Y` computed (in the absence of arithmetic overflow) as specified by Table 58, 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`.

2 If it is not possible to represent `U` or `V` with `intmax_t`, the program is ill-formed. Otherwise, an implementation should yield correct values of `U` and `V`. If it is not possible to represent `X` or `Y` with `intmax_t`, the program is ill-formed unless the implementation yields correct values of `U` and `V`.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value of <code>X</code></th>
<th>Value of <code>Y</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ratio_add&lt;R1, R2&gt;</code></td>
<td><code>R1::num * R2::den + R1::den * R2::den</code></td>
<td><code>R1::num * R1::den</code></td>
</tr>
<tr>
<td><code>ratio_subtract&lt;R1, R2&gt;</code></td>
<td><code>R1::num * R2::den - R1::den * R2::den</code></td>
<td><code>R1::num * R1::den</code></td>
</tr>
<tr>
<td><code>ratio_multiply&lt;R1, R2&gt;</code></td>
<td><code>R1::num * R2::num</code></td>
<td><code>R1::den * R2::den</code></td>
</tr>
<tr>
<td><code>ratio_divide&lt;R1, R2&gt;</code></td>
<td><code>R1::num * R2::num</code></td>
<td><code>R1::den * R2::num</code></td>
</tr>
</tbody>
</table>

3 [Example:

```c
static_assert(ratio_add<ratio<1,3>, ratio<1,6>>::num == 1, "1/3+1/6 == 1/2");
static_assert(ratio_add<ratio<1,3>, ratio<1,6>>::den == 2, "1/3+1/6 == 1/2");
static_assert(ratio_multiply<ratio<1,3>, ratio<3,2>>::num == 1, "1/3*3/2 == 1/2");
static_assert(ratio_multiply<ratio<1,3>, ratio<3,2>>::den == 2, "1/3*3/2 == 1/2");
```

// The following cases may cause the program to be ill-formed under some implementations

```c
static_assert(ratio_add<ratio<1,INT_MAX>, ratio<1,INT_MAX>>::num == 2, "1/INT_MAX+1/INT_MAX == 2/INT_MAX");
static_assert(ratio_add<ratio<1,INT_MAX>, ratio<1,INT_MAX>>::den == INT_MAX, "1/INT_MAX+1/INT_MAX == 2/INT_MAX");
static_assert(ratio_multiply<ratio<1,INT_MAX>, ratio<INT_MAX,2>>::num == 1, "1/INT_MAX*MAX/2 == 1/2");
static_assert(ratio_multiply<ratio<1,INT_MAX>, ratio<INT_MAX,2>>::den == 2, "1/INT_MAX * MAX/2 == 1/2");
```

— end example]

### 20.11.5 Comparison of ratios [ratio.comparison]

```c
template <class R1, class R2> struct ratio_equal : integral_constant<bool, see below> { };
```

1 If `R1::num` == `R2::num` and `R1::den` == `R2::den`, `ratio_equal<R1, R2>` shall be derived from `integral_constant<bool, true>`; otherwise it shall be derived from `integral_constant<bool, false>`.
template <class R1, class R2> struct ratio_not_equal
  : integral_constant<bool, !ratio_equal<R1, R2>::value> { };

template <class R1, class R2> struct ratio_less
  : integral_constant<bool, see below> { };

If R1::num * R2::den < R2::num * R1::den, ratio_less<R1, R2> shall be derived from integral_constant<bool, true>; otherwise it shall be derived from integral_constant<bool, false>. Implementations may use other algorithms to compute this relationship to avoid overflow. If overflow occurs, the program is ill-formed.

template <class R1, class R2> struct ratio_less_equal
  : integral_constant<bool, !ratio_less<R2, R1>::value> { };

template <class R1, class R2> struct ratio_greater
  : integral_constant<bool, ratio_less<R2, R1>::value> { };

template <class R1, class R2> struct ratio_greater_equal
  : integral_constant<bool, !ratio_less<R1, R2>::value> { };

20.11.6 SI types for ratio

For each of the typedefs yocto, zepto, zetta, and yotta, if both of the constants used in its specification are representable by intmax_t, the typedef shall be defined; if either of the constants is not representable by intmax_t, the typedef shall not be defined.

20.12 Time utilities

20.12.1 In general

This subclause describes the chrono library (20.12.2) and various C functions (20.12.8) that provide generally useful time utilities.

20.12.2 Header <chrono> synopsis

namespace std {
  namespace chrono {

    // 20.12.5, class template duration
template <class Rep, class Period = ratio<1> > class duration;

    // 20.12.6, class template time_point
template <class Clock, class Duration = typename Clock::duration> class time_point;
  }
}

    // 20.12.4.3 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 {

    // 20.12.4, customization traits
template <class Rep> struct treat_as_floating_point;
  template <class Rep> struct duration_values;

§ 20.12.2 625
// 20.12.5.5, duration arithmetic
template <class Rep1, class Period1, class Rep2, class Period2>
  common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
  constexpr operator+(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
  constexpr operator-(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  duration<common_type_t<Rep1, Rep2>, Period1><
    const duration<Rep1, Period1>& d, const Rep2& s);
template <class Rep1, class Rep2, class Period1>
  duration<common_type_t<Rep1, Rep2>, Period1>,
    const duration<Rep2, Period1>& d, const Rep1& s);
template <class Rep1, class Period1, class Rep2, class Period2>
  duration<common_type_t<Rep1, Rep2>, Period2>,
    const duration<Rep1, Period1>& d, const Rep2& s);
template <class Rep1, class Period1, class Rep2, class Period2>
  duration<common_type_t<Rep1, Rep2>, Period2>,
    const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

// 20.12.5.6, duration comparisons

// 20.12.5.7, duration_cast

template <class ToDuration, class Rep, class Period>
  ToDuration duration_cast(const duration<Rep, Period>& d);

// convenience typedefs
typedef duration<int, nanoseconds> nanoseconds;
typedef duration<int, microseconds> microseconds;
typedef duration<int, milliseconds> milliseconds;
typedef duration<signed integer type of at least 35 bits> seconds;
typedef duration<signed integer type of at least 29 bits, ratio< 60>> minutes;
typedef duration<signed integer type of at least 23 bits, ratio<3600>> hours;

// 20.12.6.5, 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 Rep1, class Clock, class Duration2>
constexpr time_point<Clock, common_type_t<duration<Rep1, Period1>, Duration2>> operator+(const duration<Rep1, Period1>& lhs, const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Rep2, class Period2>
constexpr time_point<Clock, common_type_t<Duration1, duration<Rep2, Period2>>> operator-(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Clock, class Duration1, class Duration2>
constexpr common_type_t<Duration1, Duration2> operator-(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

// 20.12.6.6 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);

// 20.12.6.7, time_point_cast
template <class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> time_point_cast(const time_point<Clock, Duration>& t);

// 20.12.7, clocks
class system_clock;
class steady_clock;
class high_resolution_clock;

} // namespace chrono

inline namespace literals {
inline namespace chrono_literals {

// 20.12.5.8, suffixes for duration literals
constexpr chrono::hours operator "h(unsigned long long);
20.12.3 Clock requirements [time.clock.req]

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 59.

In Table 59 C1 and C2 denote clock types. t1 and t2 are values returned by C1::now() where the call returning t1 happens before (1.10) 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 59 — 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</td>
<td>The time_point type of the clock. C1 and C2 shall refer to the same epoch.</td>
</tr>
<tr>
<td></td>
<td>chrono::time_point&lt;C2, C1::duration&gt;</td>
<td></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 (20.12.3),

(4.2) — the types TC::rep, TC::duration, and TC::time_point satisfy the requirements of EqualityComparable (Table 17), LessThanComparable (Table 18), DefaultConstructible (Table 19), CopyConstructible (Table 21), CopyAssignable (Table 23), Destructible (Table 24), and the requirements of numeric types (26.2). [Note: this means, in particular, that operations on these types will not throw exceptions. —end note]

(4.3) — lvalues of the types TC::rep, TC::duration, and TC::time_point are swappable (17.6.3.2),

(4.4) — the function TC::now() does not throw exceptions, and

(4.5) — the type TC::time_point::clock meets the TrivialClock requirements, recursively.

20.12.4 Time-related traits [time.traits]

20.12.4.1 treat_as_floating_point [time.traits.is_fp]

```cpp
template <class Rep> struct treat_as_floating_point
  : is_floating_point<Rep> { };
```

1 The duration template uses the `treat_as_floating_point` trait to help determine if a duration object can be converted to another duration with a different tick period. If `treat_as_floating_point<Rep>::value` 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<Rep>::value` is false, Rep will be treated as if it behaved like an integral type for the purpose of these conversions. —end note]

20.12.4.2 duration_values [time.traits.duration_values]

```cpp
template <class Rep>
struct duration_values {
  public:
    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();
```

2 Returns: Rep(0). [Note: Rep(0) is specified instead of Rep() because Rep() may have some other meaning, such as an uninitialized value. —end note]

3 Remark: The value returned shall be the additive identity.

```cpp
static constexpr Rep min();
```

4 Returns: numeric_limits<Rep>::lowest().

5 Remark: The value returned shall compare less than or equal to zero().
static constexpr Rep max();

Returns: numeric_limits<Rep>::max().

Remark: The value returned shall compare greater than zero().

20.12.4.3 Specializations of common_type

template <class Rep1, class Period1, class Rep2, class Period2>
struct common_type<chrono::duration<Rep1, Period1>, chrono::duration<Rep2, Period2>> {
    typedef chrono::duration<common_type_t<Rep1, Rep2>, see below> type;
};

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]

[Note: The typedef name type is a synonym for the duration with the largest tick period possible where both duration arguments will convert to it without requiring a division operation. The representation of this type is intended to be able to hold any value resulting from this conversion with no truncation error, although floating-point durations may have round-off errors. —end note]

template <class Clock, class Duration1, class Duration2>
struct common_type<chrono::time_point<Clock, Duration1>, chrono::time_point<Clock, Duration2>> {
    typedef chrono::time_point<Clock, common_type_t<Duration1, Duration2>> type;
};

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.

20.12.5 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.

template <class Rep, class Period = ratio<1>>
class duration {
public:
    typedef Rep rep;
    typedef Period period;
private:
    rep rep_; // exposition only
public:
    // 20.12.5.1, construct/copy/destroy:
    constexpr duration() = default;
    template <class Rep2>
    constexpr explicit duration(const Rep2& r);
    template <class Rep2, class Period2>
    constexpr duration(const duration<Rep2, Period2>& d);
    -duration() = default;
    duration(const duration&) = default;
    duration& operator=(const duration&) = default;

    // 20.12.5.2, observer:
    constexpr rep count() const;

    // 20.12.5.3, arithmetic:
constexpr duration operator+() const;
constexpr duration operator-() const;
duration& operator++();
duration operator++(int);
duration& operator--();
duration operator--(int);

duration& operator+=(const duration& d);
duration& operator-=(const duration& d);

duration& operator*=(const rep& rhs);
duration& operator/=(const rep& rhs);
duration& operator%=(const rep& rhs);
duration& operator%=(const duration& rhs);

// 20.12.5.4, special values:
static constexpr duration zero();
static constexpr duration min();
static constexpr duration max();
};

// time.duration.cons

template <class Rep2>
constexpr explicit duration(const Rep2& r);

Remarks: This constructor shall not participate in overload resolution unless Rep2 is implicitly convertible to rep and

1. 
   treat_as_floating_point<rep>::value is true or

2. 
   treat_as_floating_point<Rep2>::value is false.

Example:
duration<int, milli> d(3);  // OK
duration<int, milli> d(3.5);  // error

— end example]

Effects: Constructs an object of type duration.

Postcondition: 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<Rep>::value is true or both ratio_divide<Period2,
period>::den is 1 and treat_as_floating_point<Rep>::value is false. [Note: This requirement
prevents implicit truncation error when converting between integral-based duration types. Such a con-
struction 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().

20.12.5.2 duration observer

constexpr rep count() const;

Returns: rep_.

20.12.5.3 duration arithmetic

constexpr duration operator+() const;

Returns: *this.

constexpr duration operator-() const;

Returns: duration(-rep_);

duration& operator++();

Effects: ++rep_.

Returns: *this.

duration operator++(int);

Returns: duration(rep_++);

duration& operator--();

Effects: --rep_.

Returns: *this.

duration operator--(int);

Returns: duration(rep_--);.

§ 20.12.5.3
duration& operator+=(const duration& d);
   Effects: rep_ += d.count().
   Returns: *this.

duration& operator-=(const duration& d);
   Effects: rep_ -= d.count().
   Returns: *this.

duration& operator*=(const rep& rhs);
   Effects: rep_ *= rhs.
   Returns: *this.

duration& operator/=(const rep& rhs);
   Effects: rep_ /= rhs.
   Returns: *this.

duration& operator%=(const rep& rhs);
   Effects: rep_ %= rhs.
   Returns: *this.

duration& operator%=(const duration& rhs);
   Effects: rep_ %= rhs.count().
   Returns: *this.

20.12.5.4 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()).

20.12.5.5 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 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) and Rep2 is not an instantiation 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()).

20.12.5.6 duration comparisons

In the function descriptions that follow, CT represents common_type_t<A, B>, where A and B are the types of the two arguments to the function.

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator==(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

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);

20.12.5.6 634
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: CT(lhs).count() < CT(rhs).count().

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: !(lhs < rhs).

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: rhs < lhs.

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: !(lhs < rhs).

20.12.5.7 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 an instantiation of duration.

Returns: Let CF be ratio_divide<Period, typename ToDuration::period>, and CR be common_type< typename ToDuration::rep, Rep, intmax_t>::type.

(2.1) — If CF::num == 1 and CF::den == 1, returns
ToDuration(static_cast<typename ToDuration::rep>(d.count()))

(2.2) — otherwise, if CF::num != 1 and CF::den == 1, returns
ToDuration(static_cast<typename ToDuration::rep>(
    static_cast<CR>(d.count()) * static_cast<CR>(CF::num)))

(2.3) — otherwise, if CF::num == 1 and CF::den != 1, returns
ToDuration(static_cast<typename ToDuration::rep>(
    static_cast<CR>(d.count()) / static_cast<CR>(CF::den)))

(2.4) — otherwise, returns
ToDuration(static_cast<typename ToDuration::rep>(
    static_cast<CR>(d.count()) * static_cast<CR>(CF::num) / static_cast<CR>(CF::den)))

Notes: This function does not use any implicit conversions; all conversions are done with static_cast. It avoids multiplications and divisions when it is known at compile time that one or more arguments is 1. Intermediate computations are carried out in the widest representation and only converted to the destination representation at the final step.
20.12.5.8 Suffixes for duration literals

This section describes literal suffixes for constructing duration literals. The suffixes `h`, `min`, `s`, `ms`, `us`, `ns` denote duration values of the corresponding types `hours`, `minutes`, `seconds`, `milliseconds`, `microseconds`, and `nanoseconds` respectively if they are applied to integral literals.

If any of these suffixes are applied to a floating point literal the result is a `chrono::duration` literal with an unspecified floating point representation.

If any of these suffixes are applied to an integer literal and the resulting `chrono::duration` value cannot be represented in the result type because of overflow, the program is ill-formed.

[Example: The following code shows some duration literals.

```cpp
using namespace std::chrono_literals;
auto constexpr aday=24h;
auto constexpr lesson=45min;
auto constexpr halfanhour=0.5h;
```

—end example]

- `h` operator: A duration literal representing hours.
- `min` operator: A duration literal representing minutes.
- `s` operator: A duration literal representing seconds.
- `ms` operator: A duration literal representing milliseconds.
- `us` operator: A duration literal representing microseconds.
- `ns` operator: A duration literal representing nanoseconds.

20.12.6 Class template `time_point`

```cpp
template <class Clock, class Duration = typename Clock::duration>
class time_point {
  public:
    typedef Clock clock;
    typedef Duration duration;
    typedef typename duration::rep rep;
}
```

§ 20.12.6
typedef typename duration::period period;

private:
    duration d_;     // exposition only

public:
    // 20.12.6.1, 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);

    // 20.12.6.2, observer:
    constexpr duration time_since_epoch() const;

    // 20.12.6.3, arithmetic:
    time_point& operator+=(const duration& d);
    time_point& operator-=(const duration& d);

    // 20.12.6.4, special values:
    static constexpr time_point min();
    static constexpr time_point max();
};

Clock shall meet the Clock requirements (20.12).

If Duration is not an instance of duration, the program is ill-formed.

20.12.6.1  time_point constructors

constexpr time_point();
    Effects: Constructs an object of type time_point, initializing d_ with duration::zero(). Such a time_point object represents the epoch.

constexpr explicit time_point(const duration& d);
    Effects: Constructs an object of type time_point, initializing d_ with d. Such a time_point object represents the epoch + d.

template <class Duration2>
constexpr time_point(const time_point<clock, Duration2>& t);
    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().

20.12.6.2  time_point observer

constexpr duration time_since_epoch() const;
    Returns: d_.

20.12.6.3  time_point arithmetic

    time_point& operator+=(const duration& d);
        Effects: d_ += d.
        Returns: *this.
time_point& operator-=(const duration& d);

Effects: d_ -= d.
Returns: *this.

20.12.6.4 time_point special values

static constexpr time_point min();
Returns: time_point(duration::min()).

static constexpr time_point max();
Returns: time_point(duration::max()).

20.12.6.5 time_point non-member 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);

Returns: CT(lhs.time_since_epoch() + rhs), where CT is the type of the return value.

template <class Rep1, class Period1, class Clock, class Duration2>
constexpr time_point<Clock, common_type_t<duration<Rep1, Period1>, Duration2>>
operator+(const duration<Rep1, Period1>& lhs, const time_point<Clock, Duration2>& rhs);

Returns: rhs + lhs.

template <class Clock, class Duration1, class Rep2, class Period2>
constexpr time_point<Clock, common_type_t<Duration1, duration<Rep2, Period2>>>
operator-(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: lhs + (-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);

Returns: lhs.time_since_epoch() - rhs.time_since_epoch().

20.12.6.6 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).

§ 20.12.6.6
template <class Clock, class Duration1, class Duration2>
constexpr bool operator>(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

\textit{Returns:} \( \text{rhs} < \text{lhs} \).

template <class Clock, class Duration1, class Duration2>
constexpr bool operator>=(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

\textit{Returns:} \( !\text{(lhs} < \text{rhs)} \).

\subsection{time_point_cast} [time.point.cast]

\begin{verbatim}
template <class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration>
time_point_cast(const time_point<Clock, Duration>& t);
\end{verbatim}

\textit{Remarks:} This function shall not participate in overload resolution unless \texttt{ToDuration} is an instantiation of \texttt{duration}.

\textit{Returns:} \texttt{time_point<Clock, ToDuration>(duration_cast<ToDuration>(t.time_since_epoch()))}.

\subsection{Clocks} [time.clock]

The types defined in this subclause shall satisfy the \texttt{TrivialClock} requirements (20.12.3).

\subsubsection{Class system_clock} [time.clock.system]

Objects of class \texttt{system_clock} represent wall clock time from the system-wide realtime clock.

\begin{verbatim}
class system_clock {
  public:
    typedef see below rep;
    typedef ratio<unspecified, unspecified> period;
    typedef chrono::duration<rep, period> duration;
    typedef chrono::time_point<system_clock> time_point;
    static constexpr bool is_steady = unspecified;

    static time_point now() noexcept;
  // Map to C API
    static time_t to_time_t(const time_point& t) noexcept;
    static time_point from_time_t(time_t t) noexcept;
};
\end{verbatim}

\texttt{system_clock::rep};

\textit{Requires:} \texttt{system_clock::duration::min()} < \texttt{system_clock::duration::zero()} shall be \texttt{true}.

\textit{Note:} This implies that \texttt{rep} is a signed type. — end note

\texttt{system_clock::rep;

\textit{Returns:} A \texttt{time_t} object that represents the same point in time as \texttt{t} when both values are restricted to the coarser of the precisions of \texttt{time_t} and \texttt{time_point}. It is implementation defined whether values are rounded or truncated to the required precision.

\texttt{system_clock::rep;

\textit{Returns:} A \texttt{time_point} object that represents the same point in time as \texttt{t} when both values are restricted to the coarser of the precisions of \texttt{time_t} and \texttt{time_point}. It is implementation defined whether values are rounded or truncated to the required precision.

§ 20.12.7.1
20.12.7.2 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.

```cpp
class steady_clock {
  public:
    typedef unspecified rep;
    typedef ratio<unspecified, unspecified> period;
    typedef chrono::duration<rep, period> duration;
    typedef chrono::time_point<unspecified, duration> time_point;
    static constexpr bool is_steady = true;

    static time_point now() noexcept;
};
```

20.12.7.3 Class high_resolution_clock

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
class high_resolution_clock {
  public:
    typedef unspecified rep;
    typedef ratio<unspecified, unspecified> period;
    typedef chrono::duration<rep, period> duration;
    typedef chrono::time_point<unspecified, duration> time_point;
    static constexpr bool is_steady = unspecified;

    static time_point now() noexcept;
};
```

20.12.8 Date and time functions

Table 60 describes the header <ctime>.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros</td>
<td>NULL   CLOCKS_PER_SEC</td>
</tr>
<tr>
<td>Types</td>
<td>size_t   clock_t   time_t</td>
</tr>
<tr>
<td>Struct</td>
<td>tm</td>
</tr>
<tr>
<td>Functions</td>
<td>asctime   clock   difftime   localtime   strftime</td>
</tr>
<tr>
<td></td>
<td>ctime   gmtime   mktime   time</td>
</tr>
</tbody>
</table>

2 The contents are the same as the Standard C library header <time.h>. The functions asctime, ctime, gmtime, and localtime are not required to avoid data races (17.6.5.9).

See also: ISO C Clause 7.12, Amendment 1 Clause 4.6.4.

20.13 Class template scoped_allocator_adaptor

20.13.1 Header <scoped_allocator> synopsis

231) strftime supports the C conversion specifiers C, D, e, F, g, G, h, r, R, t, T, u, V, and z, and the modifiers E and 0.
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 `scoped_allocator_adaptor` itself, thus using the same allocator resource for the container and every element within the container and, if the elements themselves are containers, each of their elements recursively. If instantiated with more than one allocator, the first allocator is the outer allocator for use by the container, the second allocator is passed to the constructors of the container’s elements, and, if the elements themselves are containers, the third allocator is passed to the elements’ elements, and so on. If containers are nested to a depth greater than the number of allocators, the last allocator is used repeatedly, as in the single-allocator case, for any remaining recursions. [Note: The `scoped_allocator_adaptor` is derived from the outer allocator type so it can be substituted for the outer allocator type in most expressions. — end note]

```cpp
namespace std {

    template <class OuterAlloc, class... InnerAllocs>
    class scoped_allocator_adaptor : public OuterAlloc {
    
    private:
        typedef allocator_traits<OuterAlloc> OuterTraits;  // exposition only
        scoped_allocator_adaptor<InnerAllocs...> inner;  // exposition only

    public:
        typedef OuterAlloc outer_allocator_type;

        typedef see below inner_allocator_type;

        typedef typename OuterTraits::value_type value_type;
        typedef typename OuterTraits::size_type size_type;
        typedef typename OuterTraits::difference_type difference_type;
        typedef typename OuterTraits::pointer pointer;
        typedef typename OuterTraits::const_pointer const_pointer;
        typedef typename OuterTraits::void_pointer void_pointer;
        typedef typename OuterTraits::const_void_pointer const_void_pointer;

        typedef see below propagate_on_container_copy_assignment;
        typedef see below propagate_on_container_move_assignment;
        typedef see below propagate_on_container_swap;
        typedef see below is_always_equal;

        template <class Tp>
        struct rebind {
            typedef scoped_allocator_adaptor<
                OuterTraits::template rebind_alloc<Tp>, InnerAllocs...> other;
        };

        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;

template <class OuterA2>
scoped_allocator_adaptor<
const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& other> noexcept;

scoped_allocator_adaptor<
scoped_allocator_adaptor<OuterA2, InnerAllocs...>&& other> noexcept;

~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;

pointer allocate(size_type n);
pointer allocate(size_type n, const void_pointer hint);
void deallocate(pointer p, size_type n);
size_type max_size() const;

template <class T, class... Args>
void construct(T* p, Args&&... args);

template <class 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>
void construct(pair<T1, T2>* p);

template <class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, const pair<U, V>& x);

template <class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, pair<U, V>&& x);

template <class T>
void destroy(T* p);

scoped_allocator_adaptor select_on_container_copy_construction() const;


template <class OuterA1, class OuterA2, class... InnerAllocs>
bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
    const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;

template <class OuterA1, class OuterA2, class... InnerAllocs>
bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
    const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;

}

20.13.2 Scoped allocator adaptor member types [allocator.adaptor.types]
typedef see below inner_allocator_type;

Type: scoped_allocator_adaptor<OuterAlloc> if sizeof...(InnerAllocs) is zero; otherwise, scoped_allocator_adaptor<InnerAllocs...>.

typedef see below propagate_on_container_copy_assignment;

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.

typedef see below propagate_on_container_move_assignment;

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.

typedef see below propagate_on_container_swap;

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.

typedef see below is_always_equal;

Type: true_type if allocator_traits<A>::is_always_equal::value is true for every A in the set of OuterAlloc and InnerAllocs...; otherwise, false_type.

20.13.3 Scoped allocator adaptor constructors

scoped_allocator_adaptor();

Effects: value-initializes the OuterAlloc base class and the inner allocator object.

template <class OuterA2>
scoped_allocator_adaptor(OuterA2&& outerAlloc,
const InnerAllocs&... innerAllocs) noexcept;

Requires: OuterAlloc shall be constructible from OuterA2.

Effects: initializes the OuterAlloc base class with std::forward<OuterA2>(outerAlloc) and inner with innerAllocs... (hence recursively initializing each allocator within the adaptor with the corresponding allocator from the argument list).

scoped_allocator_adaptor(const scoped_allocator_adaptor& other) noexcept;

Effects: initializes each allocator within the adaptor with the corresponding allocator from other.

scoped_allocator_adaptor(scoped_allocator_adaptor&& other) noexcept;

Effects: move constructs each allocator within the adaptor with the corresponding allocator from other.

template <class OuterA2>
scoped_allocator_adaptor(const scoped_allocator_adaptor<OuterA2,
InnerAllocs...>& other) noexcept;

Requires: OuterAlloc shall be constructible from OuterA2.

Effects: initializes each allocator within the adaptor with the corresponding allocator from other.

template <class OuterA2>
scoped_allocator_adaptor(scoped_allocator_adaptor<OuterA2,
InnerAllocs...>&& other) noexcept;

Requires: OuterAlloc shall be constructible from OuterA2.

Effects: initializes each allocator within the adaptor with the corresponding allocator rvalue from other.

§ 20.13.3
20.13.4 Scoped allocator adaptor members

In the `construct` member functions, \( \text{OUTERMOST}(x) \) is \( x \) if \( x \) does not have an `outer_allocator()` member function and \( \text{OUTERMOST}(x.outer_allocator()) \) otherwise; \( \text{OUTERMOST}_\text{ALLOC_TRAITS}(x) \) is \( \text{allocator_traits}<\text{decltype}(\text{OUTERMOST}(x))> \). [Note: \( \text{OUTERMOST}(x) \) and \( \text{OUTERMOST}_\text{ALLOC_TRAITS}(x) \) 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
class inner_allocator_type& inner_allocator() noexcept;
class const inner_allocator_type& inner_allocator() const noexcept;
```

Returns: `*this` if `sizeof...(InnerAllocs)` is zero; otherwise, `inner`.

```cpp
class outer_allocator_type& outer_allocator() noexcept;
class const outer_allocator_type& outer_allocator() const noexcept;
```

Returns: `static_cast<OuterAlloc&>(*this)`.

```cpp
pointer allocate(size_type n);
```

Returns: `allocator_traits<OuterAlloc>::allocate(outer_allocator(), n)`.

```cpp
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: `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<T, inner_allocator_type>::value` is false and `is_constructible<T, Args...>::value` is true, calls `OUTERMOST_ALLOC_TRAITS(*this)::construct(OUTERMOST(*this), p, std::forward<Args>(args)...)`.

(9.2) — Otherwise, if `uses_allocator<T, inner_allocator_type>::value` is true and `is_constructible<T, allocator_arg_t, inner_allocator_type, Args...>::value` 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<T, inner_allocator_type>::value` is true and `is_constructible<T, Args..., inner_allocator_type>::value` 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 21).

Effects: Constructs a tuple object xprime from x by the following rules:

(11.1) If uses_allocator<T1, inner_allocator_type>::value is false and is_constructible<T1, Args1...>::value is true, then xprime is x.

(11.2) Otherwise, if uses_allocator<T1, inner_allocator_type>::value is true and is_constructible<T1, allocator_arg_t, inner_allocator_type, Args1...>::value 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<T1, inner_allocator_type>::value is true and is_constructible<T1, Args1..., inner_allocator_type>::value 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<T2, inner_allocator_type>::value is false and is_constructible<T2, Args2...>::value is true, then yprime is y.

(11.6) Otherwise, if uses_allocator<T2, inner_allocator_type>::value is true and is_constructible<T2, allocator_arg_t, inner_allocator_type, Args2...>::value 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<T2, inner_allocator_type>::value is true and is_constructible<T2, Args2..., inner_allocator_type>::value 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 this->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 this->construct(p, piecewise_construct, forward_as_tuple(std::forward<U>(x)), forward_as_tuple(std::forward<V>(y))).

Effects: Equivalent to this->construct(p, piecewise_construct, forward_as_tuple(x.first), forward_as_tuple(x.second)).

Effects: Equivalent to this->construct(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.

20.13.5 Scoped allocator operators

template <class OuterA1, class OuterA2, class... InnerAllocs>
    bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
        const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
    Returns: a.outer_allocator() == b.outer_allocator() if sizeof...(InnerAllocs) is zero; oth-
erwise, a.outer_allocator() == b.outer_allocator() && a.inner_allocator() == b.inner_-

    operator!=(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
        const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
    Returns: !(a == b).

20.14 Class type_index

20.14.1 Header <typeindex> synopsis

namespace std {
    class type_index;
    template <class T> struct hash;
    template<> struct hash<type_index>;
}

20.14.2 type_index overview

namespace std {
    class type_index {
        public:
            type_index(const type_info& rhs) noexcept;
            bool operator==(const type_index& rhs) const noexcept;
            bool operator!=(const type_index& rhs) const noexcept;
            bool operator<(const type_info& rhs) const noexcept;
            bool operator<= (const type_info& rhs) const noexcept;
            bool operator> (const type_info& rhs) const noexcept;
            bool operator>= (const type_info& 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.
    }
}

§ 20.14.2
The class `type_index` provides a simple wrapper for `type_info` which can be used as an index type in associative containers (23.4) and in unordered associative containers (23.5).

### 20.14.3 type_index members

**type_index(const type_info& rhs) noexcept;**

*Effects:* constructs a `type_index` object, the equivalent of `target = &rhs`.

**bool operator==(const type_index& rhs) const noexcept;**

*Returns:* `*target == *rhs.target`

**bool operator!=(const type_index& rhs) const noexcept;**

*Returns:* `*target != *rhs.target`

**bool operator<(const type_index& rhs) const noexcept;**

*Returns:* `target->before(*rhs.target)`

**bool operator<=(const type_index& rhs) const noexcept;**

*Returns:* `!rhs.target->before(*target)`

**bool operator>(const type_index& rhs) const noexcept;**

*Returns:* `rhs.target->before(*target)`

**bool operator>=(const type_index& rhs) const noexcept;**

*Returns:* `!target->before(*rhs.target)`

**size_t hash_code() const noexcept;**

*Returns:* `target->hash_code()`

**const char* name() const noexcept;**

*Returns:* `target->name()`

### 20.14.4 Hash support

```cpp
template <> struct hash<type_index>;
```

*The template specialization shall meet the requirements of class template `hash` (20.9.13). For an object `index` of type `type_index`, `hash<type_index>()(index)` shall evaluate to the same result as `index.hash_code()`.*
21  Strings library

21.1  General

1 This Clause describes components for manipulating sequences of any non-array POD (3.9) type. In this Clause 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, a string class, and null-terminated sequence utilities, as summarized in Table 61.

Table 61 — Strings library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.2 Character traits</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>21.3 String classes</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cctype&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cwctype&gt;</td>
</tr>
<tr>
<td>21.8 Null-terminated sequence utilities</td>
<td>&lt;cstring&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cwchar&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cuchar&gt;</td>
</tr>
</tbody>
</table>

21.2  Character traits

1 This subclause defines requirements on classes representing character traits, and defines a class template char_traits<T>, 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 Clauses 21.3 and 27 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 typedefs 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 struct template, char_traits<T>, 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.

21.2.1  Character traits requirements

1 In Table 62, 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 std::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.

§ 21.2.1
### Table 62 — Character traits requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::char_type</td>
<td>charT</td>
<td>(described in 21.2.2)</td>
<td>compile-time</td>
</tr>
<tr>
<td>X::int_type</td>
<td></td>
<td></td>
<td>compile-time</td>
</tr>
<tr>
<td>X::off_type</td>
<td></td>
<td></td>
<td>compile-time</td>
</tr>
<tr>
<td>X::pos_type</td>
<td></td>
<td></td>
<td>compile-time</td>
</tr>
<tr>
<td>X::state_type</td>
<td></td>
<td></td>
<td>compile-time</td>
</tr>
<tr>
<td>X::eq(c,d)</td>
<td>bool</td>
<td>yields: 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>yields: 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>yields: 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>std::size_t</td>
<td>yields: 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>yields: 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. yields: s.</td>
<td>linear</td>
</tr>
<tr>
<td>X::copy(s,p,n)</td>
<td>X::char_type*</td>
<td>pre: p not in [s,s+n). yields: 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). yields: s.</td>
<td>linear</td>
</tr>
<tr>
<td>X::not_eof(e)</td>
<td>int_type</td>
<td>yields: 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>yields: 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>
</tbody>
</table>
Table 62 — 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><code>X::to_int_type(c)</code></td>
<td><code>X::int_type</code></td>
<td>yields: some value e, constrained by the definitions of to_char_type and eq_int_type.</td>
<td>constant</td>
</tr>
<tr>
<td><code>X::eq_int_type(e,f)</code></td>
<td>bool</td>
<td>yields: for all c and d, (X::eq(c,d)) is equal to (X::eq_int_type(X::to_int_type(c), X::to_int_type(d))); otherwise, yields true if e and f are both copies of X::eof(); otherwise, yields false if one of e and f is a copy of X::eof() and the other is not; otherwise the value is unspecified.</td>
<td>constant</td>
</tr>
<tr>
<td><code>X::eof()</code></td>
<td><code>X::int_type</code></td>
<td>yields: a value e such that (X::eq_int_type(e,X::to_int_type(c))) is false for all values c.</td>
<td>constant</td>
</tr>
</tbody>
</table>

2 The struct template

```cpp
template<class charT> struct char_traits;
```

shall be provided in the header `<string>` as a basis for explicit specializations.

21.2.2 traits typedefs [char.traits typedefs]

typedef CHAR_T char_type;

1 The type `char_type` is used to refer to the character container type in the implementation of the library classes defined in 21.3 and Clause 27.

typedef INT_T int_type;

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.²³²

```
typedef implementation-defined off_type;
typedef implementation-defined pos_type;
```

3 Requires: Requirements for `off_type` and `pos_type` are described in 27.2.2 and 27.3.

typedef STATE_T state_type;

4 Requires: `state_type` shall meet the requirements of `CopyAssignable` (Table 23), `CopyConstructible` (Table 21), and `DefaultConstructible` (Table 19) types.

²³² If `eof()` can be held in `char_type` then some iostreams operations may give surprising results.
21.2.3 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 template struct `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 Clause 21.2.1.

21.2.3.1 struct char_traits<char>

```cpp
namespace std {
    template<> struct char_traits<char> {
        typedef char char_type;
        typedef int int_type;
        typedef streamoff off_type;
        typedef streampos pos_type;
        typedef mbstate_t state_type;

        static 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 int compare(const char_type* s1, const char_type* s2, size_t n);
        static size_t length(const char_type* s);
        static 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 27.2.2 and 27.3.

3. The type `streamoff` shall be an implementation-defined type that satisfies the requirements for `off_type` in 27.2.2 and 27.3.

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`.

§ 21.2.3.1

651
21.2.3.2 struct char_traits<char16_t>

namespace std {

    template<> struct char_traits<char16_t> {
        typedef char16_t char_type;
        typedef uint_least16_t int_type;
        typedef streamoff off_type;
        typedef u16streampos pos_type;
        typedef mbstate_t state_type;

        static 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 int compare(const char_type* s1, const char_type* s2, size_t n);
        static size_t length(const char_type* s);
        static 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 type u16streampos shall be an implementation-defined type that satisfies the requirements for pos_type in 27.2.2 and 27.3.

The two-argument members assign, eq, and lt shall be defined identically to the built-in operators =, ==, and < respectively.

The member eof() shall return an implementation-defined constant that cannot appear as a valid UTF-16 code unit.

21.2.3.3 struct char_traits<char32_t>

namespace std {

    template<> struct char_traits<char32_t> {
        typedef char32_t char_type;
        typedef uint_least32_t int_type;
        typedef streamoff off_type;
        typedef u32streampos pos_type;
        typedef mbstate_t state_type;

        static 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 int compare(const char_type* s1, const char_type* s2, size_t n);
        static size_t length(const char_type* s);
        static 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;
    };
}
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 type u32streampos shall be an implementation-defined type that satisfies the requirements for pos_type in 27.2.2 and 27.3.

2 The two-argument members assign, eq, and lt shall be defined identically to the built-in operators =, ==, and < respectively.

3 The member eof() shall return an implementation-defined constant that cannot appear as a Unicode code point.

21.2.3.4 struct char_traits<wchar_t> [char.traits.specializations.wchar.t]
namespace std {
    template<> struct char_traits<wchar_t> {
        typedef wchar_t char_type;
        typedef wint_t int_type;
        typedef streamoff off_type;
        typedef wstreampos pos_type;
        typedef mbstate_t state_type;

        static 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 int compare(const char_type* s1, const char_type* s2, size_t n);
        static size_t length(const char_type* s);
        static 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, and state_type shall be wint_t, wstreampos, and mbstate_t respectively.

2 The type wstreampos shall be an implementation-defined type that satisfies the requirements for pos_type in 27.2.2 and 27.3.
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`.

### 21.3 String classes

The header `<string>` defines the `basic_string` class template for manipulating varying-length sequences of char-like objects and four typedefs, `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.

**Header `<string>` synopsis**

```
#include <initializer_list>

namespace std {

    // 21.2, character traits:
    template<class charT> struct char_traits;
    template <> struct char_traits<char>;
    template <> struct char_traits<char16_t>;
    template <> struct char_traits<char32_t>;
    template <> struct char_traits<wchar_t>;

    // 21.4, basic_string:
    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT> >
        class basic_string;

    template<class charT, class traits, class Allocator>
        basic_string<
            charT,traits,Allocator>
            operator+(const basic_string<
                charT,traits,Allocator>& lhs,
                const basic_string<
                    charT,traits,Allocator>& rhs);

    template<class charT, class traits, class Allocator>
        basic_string<
            charT,traits,Allocator>
            operator+(const basic_string<
                charT,traits,Allocator>&& lhs,
                const basic_string<
                    charT,traits,Allocator>& rhs);

    template<class charT, class traits, class Allocator>
        basic_string<
            charT,traits,Allocator>
            operator+(const charT* lhs,
                const 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* rhs,
                const basic_string<
                    charT,traits,Allocator>& lhs);

    template<class charT, class traits, class Allocator>
        basic_string<
            charT,traits,Allocator>
            operator+(const charT* rhs,
                const basic_string<
                    charT,traits,Allocator>&& lhs);

    template<class charT, class traits, class Allocator>
        basic_string<
            charT,traits,Allocator>
            operator+(basic_string<
                charT,traits,Allocator>& lhs,
                const basic_string<
                    charT,traits,Allocator>& rhs);

    template<class charT, class traits, class Allocator>
        basic_string<
            charT,traits,Allocator>
            operator+(basic_string<
                charT,traits,Allocator>&& lhs,
                const basic_string<
                    charT,traits,Allocator>& rhs);

    template<class charT, class traits, class Allocator>
        basic_string<
            charT,traits,Allocator>
            operator+(const basic_string<
                charT,traits,Allocator>& rhs,
                basic_string<
                    charT,traits,Allocator>& lhs);

    template<class charT, class traits, class Allocator>
        basic_string<
            charT,traits,Allocator>
            operator+(const basic_string<
                charT,traits,Allocator>&& rhs,
                basic_string<
                    charT,traits,Allocator>& lhs);

    template<class charT, class traits, class Allocator>
        basic_string<
            charT,traits,Allocator>
            operator+(basic_string<
                charT,traits,Allocator>& rhs,
                basic_string<
                    charT,traits,Allocator>&& lhs);

    template<class charT, class traits, class Allocator>
        basic_string<
            charT,traits,Allocator>
            operator+(basic_string<
                charT,traits,Allocator>&& rhs,
                basic_string<
                    charT,traits,Allocator>&& lhs);

```

§ 21.3
basic_string<
class charT, class traits, class Allocator>
operator+(charT lhs, const basic_string<
class charT, class traits, class Allocator>& rhs);
template<class charT, class traits, class Allocator>
operator+(charT lhs, basic_string<
class charT, class traits, class Allocator>&& rhs);
template<class charT, class traits, class Allocator>
operator+(const basic_string<
class charT, class traits, class Allocator>& lhs,
const charT* rhs);
template<class charT, class traits, class Allocator>
operator+(basic_string<
class charT, traits, Allocator>&& lhs,
const charT* rhs);
template<class charT, class traits, class Allocator>
operator+(const basic_string<
class charT, traits, Allocator>&& lhs,
const charT* rhs);
template<class charT, class traits, class Allocator>
operator+(const basic_string<
class charT, traits, Allocator>& lhs,
const charT* rhs);
template<class charT, class traits, class Allocator>
operator+(basic_string<
class charT, traits, Allocator>&& lhs,
const charT* rhs);

§ 21.3
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;

// 21.4.8.8, 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)));

// 21.4.8.9, 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);

// 21.4.8.9, inserters and extractors:
template<class charT, class traits, class Allocator>
basic_istream<charT,traits>&
    getline(basic_istream<charT,traits>&& is,
    basic_string<charT,traits,Allocator>& str,
    charT delim);
// basic_string typedef names
typedef basic_string<char> string;
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
typedef basic_string<wchar_t> wstring;

// 21.5, numeric conversions:
int stoi(const string& str, size_t* idx = 0, int base = 10);
long stol(const string& str, size_t* idx = 0, int base = 10);
unsigned long stoul(const string& str, size_t* idx = 0, int base = 10);
long long stoll(const string& str, size_t* idx = 0, int base = 10);
unsigned long long stoull(const string& str, size_t* idx = 0);
float stof(const string& str, size_t* idx = 0);
double stod(const string& str, size_t* idx = 0);
long double stold(const string& str, size_t* idx = 0);
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 = 0, int base = 10);
long stol(const wstring& str, size_t* idx = 0, int base = 10);
unsigned long stoul(const wstring& str, size_t* idx = 0, int base = 10);
long long stoll(const wstring& str, size_t* idx = 0, int base = 10);
unsigned long long stoull(const wstring& str, size_t* idx = 0);
float stof(const wstring& str, size_t* idx = 0);
double stod(const wstring& str, size_t* idx = 0);
long double stold(const wstring& str, size_t* idx = 0);
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);

// 21.6, hash support:
template <class T> struct hash;
template <> struct hash<string>;
template <> struct hash<u16string>;
template <> struct hash<u32string>;
template <> struct hash<wstring>;

inline namespace literals {
inline namespace string_literals {

// 21.7, suffix for basic_string literals:

§ 21.3 657
21.4 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.

A basic_string is a contiguous container (23.2.1).

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 (19.2.4);
- an out-of-range error is associated with exceptions of type out_of_range (19.2.5).

```cpp
namespace std {
    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT> >
    class basic_string {
    public:
        // types:
        typedef traits traits_type;
        typedef typename traits::char_type value_type;
        typedef Allocator allocator_type;
        typedef typename allocator_traits<Allocator>::size_type size_type;
        typedef typename allocator_traits<Allocator>::difference_type difference_type;

        typedef value_type& reference;
        typedef const value_type& const_reference;
        typedef typename allocator_traits<Allocator>::pointer pointer;
        typedef typename allocator_traits<Allocator>::const_pointer const_pointer;

        typedef implementation-defined iterator; // See 23.2
        typedef implementation-defined const_iterator; // See 23.2
        typedef std::reverse_iterator<iterator> reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

        static const size_type npos = -1;

        // 21.4.2, construct/copy/destroy:
        basic_string() noexcept : 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, size_type n = npos,
    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<class 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)
optimal(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
    allocator_traits<Allocator>::is_always_equal::value);
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);
basic_string& operator=(initializer_list<charT>);

// 21.4.3, iterators:
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// 21.4.4, 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;
bool empty() const noexcept;

// 21.4.5, element access:
const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
const_reference at(size_type n) const;
reference at(size_type n);

const charT& front() const;
charT& front();
const charT& back() const;
charT& back();

// 21.4.6, modifiers:
basic_string& operator+=(const basic_string& str);
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(const charT* s, size_type n);
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;
basic_string& assign(const basic_string& str, 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 pos1, 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, const charT* s, size_type n);
basic_string& insert(size_type pos, const charT* s);
basic_string& insert(size_type pos, size_type n, charT c);
iterator insert(const_iterator p, charT c);
iterator insert(const_iterator p, size_type n, charT c);
template<class InputIterator>
iterator insert(const_iterator p, InputIterator first, InputIterator last);
iterator insert(const_iterator p, initializer_list<charT>);

basic_string& erase(size_type pos = 0, size_type n = npos);
iterator erase(const_iterator p);
iterator erase(const_iterator first, const_iterator last);

void pop_back();

basic_string& replace(size_type pos1, size_type n1,
    const basic_string& str);
basic_string& replace(size_type pos1, size_type n1, const basic_string& str, size_type pos2, size_type n2 = npos);
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);
basic_string& replace(const_iterator i1, const_iterator i2, const basic_string& str);
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);

// 21.4.7, string operations:
const charT* c_str() const noexcept;
const charT* data() const noexcept;
allocator_type get_allocator() 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(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(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_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(const basic_string& str, size_type pos = 0) const noexcept;
size_type find_first_not_of(const charT* s, size_type pos, 
    size_type n) const;
size_type find_first_not_of(const charT* s, size_type pos = 0) const;
size_type find_first_not_of(charT c, size_type pos = 0) const;
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(const basic_string& str) const noexcept;
int compare(size_type pos1, size_type n1, 
    const basic_string& str) const;
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; 
};

} 

21.4.1 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 23.2.1.

4 References, pointers, and iterators referring to the elements of a basic_string sequence may be invalidated by the following uses of that basic_string object:

(4.1) — as an argument to any standard library function taking a reference to non-const basic_string as an argument.

(4.2) — Calling non-const member functions, except operator[], at, front, back, begin, rbegin, end, and rend.

21.4.2 basic_string constructors and assignment operators

explicit basic_string(const Allocator& a) noexcept;

Effects: Constructs an object of class basic_string. The postconditions of this function are indicated in Table 63.

For example, as an argument to non-member functions swap() (21.4.8.8), operator>>() (21.4.8.9), and getline() (21.4.8.9), or as an argument to basic_string::swap()
Table 63 — 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>

basic_string(const basic_string& str);

basic_string(basic_string&& str) noexcept;

Effects: Constructs an object of class basic_string as indicated in Table 64. In the second form, str is left in a valid state with an unspecified value.

Table 64 — 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, size_type n = npos, const Allocator& a = Allocator());

Requires: pos <= str.size()

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 the smaller of n and str.size() - pos, as indicated in Table 65.

Table 65 — 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 rlen consecutive elements of the string controlled by str beginning at position pos</td>
</tr>
<tr>
<td>size()</td>
<td>rlen</td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as size()</td>
</tr>
</tbody>
</table>

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 66.
Table 66 — `basic_string(const charT*, size_type, const Allocator&)` effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data()</code></td>
<td>points at the first element of an allocated copy of the array whose first element is pointed at by <code>s</code></td>
</tr>
<tr>
<td><code>size()</code></td>
<td><code>n</code></td>
</tr>
<tr>
<td><code>capacity()</code></td>
<td>a value at least as large as <code>size()</code></td>
</tr>
</tbody>
</table>

Requires: `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

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 67.

Table 67 — `basic_string(const charT*, const Allocator&)` effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data()</code></td>
<td>points at the first element of an allocated copy of the array whose first element is pointed at by <code>s</code></td>
</tr>
<tr>
<td><code>size()</code></td>
<td><code>traits::length(s)</code></td>
</tr>
<tr>
<td><code>capacity()</code></td>
<td>a value at least as large as <code>size()</code></td>
</tr>
</tbody>
</table>

Remarks: Uses `traits::length()`.

`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 68.

Table 68 — `basic_string(size_t, charT, const Allocator&)` effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data()</code></td>
<td>points at the first element of an allocated array of <code>n</code> elements, each storing the initial value <code>c</code></td>
</tr>
<tr>
<td><code>size()</code></td>
<td><code>n</code></td>
</tr>
<tr>
<td><code>capacity()</code></td>
<td>a value at least as large as <code>size()</code></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 23.2.3).

`basic_string(initializer_list<charT> il, const Allocator& a = Allocator());`

Effects: Same as `basic_string(il.begin(), il.end(), a)`.

§ 21.4.2
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 69. The stored allocator is constructed from `alloc`. In the second form, `str` is left in a valid state with an unspecified value.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data()</code></td>
<td>points at the first element of an allocated copy of the array</td>
</tr>
<tr>
<td></td>
<td>whose first element is pointed at by the original value of <code>str.data()</code></td>
</tr>
<tr>
<td><code>size()</code></td>
<td>the original value of <code>str.size()</code></td>
</tr>
<tr>
<td><code>capacity()</code></td>
<td>a value at least as large as <code>size()</code></td>
</tr>
<tr>
<td><code>get_allocator()</code></td>
<td><code>alloc</code></td>
</tr>
</tbody>
</table>

*Throws:* The second form throws nothing if `alloc == str.get_allocator()`.

basic_string& operator=(const basic_string& str);

*Effects:* If `*this` and `str` are not the same object, modifies `*this` as shown in Table 70. If `*this` and `str` are the same object, the member has no effect.

*Returns:* `*this`

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data()</code></td>
<td>points at the first element of an allocated copy of the array</td>
</tr>
<tr>
<td></td>
<td>whose first element is pointed at by <code>str.data()</code></td>
</tr>
<tr>
<td><code>size()</code></td>
<td><code>str.size()</code></td>
</tr>
<tr>
<td><code>capacity()</code></td>
<td>a value at least as large as <code>size()</code></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:* If `*this` and `str` are not the same object, modifies `*this` as shown in Table 71. [Note: A valid implementation is `swap(str)`. — end note]

If `*this` and `str` are the same object, the member has no effect.

*Returns:* `*this`

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data()</code></td>
<td>points at the array whose first element was pointed at by <code>str.data()</code></td>
</tr>
<tr>
<td><code>size()</code></td>
<td>previous value of <code>str.size()</code></td>
</tr>
<tr>
<td><code>capacity()</code></td>
<td>a value at least as large as <code>size()</code></td>
</tr>
</tbody>
</table>
basic_string& operator=(const charT* s);

24 Returns: *this = basic_string(s).
Remarks: Uses traits::length().

basic_string& operator=(charT c);

26 Returns: *this = basic_string(1,c).

basic_string& operator=(initializer_list<charT> il);

27 Effects: *this = basic_string(il).
Returns: *this.

21.4.3 basic_string iterator support

[058] iterator begin() noexcept;
[058] const_iterator begin() const noexcept;
[058] 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()).

21.4.4 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 size of the largest possible string.
Complexity: Constant time.

void resize(size_type n, charT c);
6  
7  Requires: n <= max_size()
8  Throws: length_error if n > max_size().
9  Effects: Alters the length of the string designated by *this as follows:
10  — If n <= size(), the function replaces the string designated by *this with a string of length n
11     whose elements are a copy of the initial elements of the original string designated by *this.
12  — If n > size(), the function replaces the string designated by *this with a string of length n
13     whose first size() elements are a copy of the original string designated by *this, and whose
14     remaining elements are all initialized to c.
15
16  void resize(size_type n);
17  Effects: resize(n,charT()).
18
19  size_type capacity() const noexcept;
20  Returns: The size of the allocated storage in the string.
21
22  void reserve(size_type res_arg=0);
23  The member function reserve() is a directive that informs a basic_string object of a planned change
24  in size, so that it can manage the storage allocation accordingly.
25  Effects: After reserve(), capacity() is greater or equal to the argument of reserve. [ Note: Calling
26     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 ]
27  Throws: length_error if res_arg > max_size().²³⁵
28
29  void shrink_to_fit();
30  Remarks: 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 ]
31
32  void clear() noexcept;
33  Effects: Behaves as if the function calls:
34      erase(begin(), end());
35
36  bool empty() const noexcept;
37  Returns: size() == 0.
38
39  21.4.5 basic_string element access [string.access]
40
41  const_reference operator[](size_type pos) const;
42  reference     operator[](size_type pos);
43  Requires: pos <= size().
44  Returns: *(begin() + pos) if pos < size(). Otherwise, returns a reference to an object of type
45          charT with value charT(), where modifying the object leads to undefined behavior.
46  Throws: Nothing.
47  Complexity: Constant time.
48
²³⁵) reserve() uses allocator_traits<Allocator>::allocate() which may throw an appropriate exception.
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 operator[](0).

const charT& back() const;
charT& back();

Requires: !empty()
Effects: Equivalent to operator[](size() - 1).

21.4.6 basic_string modifiers

21.4.6.1 basic_string::operator+=

basic_string&
operator+=(const basic_string& str);

Effects: Calls append(str).
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.

21.4.6.2 basic_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);
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 append(basic_string(n, c)).
    Returns: *this.

template<class InputIterator>
    basic_string& append(InputIterator first, InputIterator last);

    Requires: [first, last) is a valid range.
    Effects: Equivalent to append(basic_string(first, last)).
    Returns: *this.

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).

21.4.6.3 basic_string::assign

    basic_string& assign(const basic_string& str);

    Effects: Equivalent to assign(str, 0, npos).
    Returns: *this.

basic_string& assign(basic_string&& str) noexcept;

§ 21.4.6.3
Effects: The function replaces the string controlled by \*this with a string of length str.size() whose elements are a copy of the string controlled by str. [Note: A valid implementation is \texttt{swap(str)}. — end note]

Returns: \*this.

\begin{verbatim}
basic_string&
assign(const basic_string& str, size_type pos, 
size_type n = npos);
\end{verbatim}

Requires: pos <= str.size()

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.

\begin{verbatim}
basic_string& assign(const charT* s, size_type n);
\end{verbatim}

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.

\begin{verbatim}
basic_string& assign(const charT* s);
\end{verbatim}

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.

\begin{verbatim}
basic_string& assign(initializer_list<charT> il);
\end{verbatim}

Effects: Calls assign(il.begin(), il.size()).

*this.

\begin{verbatim}
basic_string& assign(size_type n, charT c);
\end{verbatim}

Effects: Equivalent to assign(basic_string(n, c)).

Returns: \*this.

\begin{verbatim}
template<class InputIterator>
basic_string& assign(InputIterator first, InputIterator last);
\end{verbatim}

Effects: Equivalent to assign(basic_string(first, last)).

Returns: \*this.
21.4.6.4 basic_string::insert

basic_string&
insert(size_type pos1,
    const basic_string& str);
1 Requires: pos <= size().
2 Throws: out_of_range if pos > size().
3 Effects: Calls insert(pos, str.data(), str.size()).
4 Returns: *this.

basic_string&
insert(size_type pos1,
    const basic_string& str,
    size_type pos2, size_type n = npos);
5 Requires: pos1 <= size() and pos2 <= str.size()
6 Throws: out_of_range if pos1 > size() or pos2 > str.size().
7 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).
8 Returns: *this.

basic_string&
insert(size_type pos, const charT* s, size_type n);
9 Requires: s points to an array of at least n elements of charT and pos <= size().
10 Throws: out_of_range if pos > size() or length_error if size() + n > max_size().
11 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.
12 Returns: *this.

basic_string&
insert(size_type pos, const charT* s);
13 Requires: pos <= size() and s points to an array of at least traits::length(s) + 1 elements of charT.
14 Effects: Equivalent to insert(pos, s, traits::length(s)).
15 Returns: *this.

basic_string&
insert(size_type pos, size_type n, charT c);
16 Effects: Equivalent to insert(pos, basic_string(n, c)).
17 Returns: *this.

iterator insert(const_iterator p, charT c);
18 Requires: p is a valid iterator on *this.
19 Effects: inserts a copy of c before the character referred to by p.
20 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)).
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: 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.

## 21.4.6.5  basic_string::erase  [string::erase]

basic_string& erase(size_type pos = 0, size_type n = npos);

Requires: pos <= size()
Throws: out_of_range if pos > size().
Effects: Determines the effective length xlen of the string to be removed as the smaller of n and size() - pos.
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.
Returns: *this.

iterator erase(const_iterator p);

Throws: Nothing.
Effects: removes the character referred to by p.
Returns: An iterator which points to the element immediately following p prior to the element being erased. If no such element exists, end() is returned.

iterator erase(const_iterator first, const_iterator last);

Requires: first and last are valid iterators on *this, defining a range [first,last).
Throws: Nothing.
Effects: removes the characters in the range [first,last).
Returns: An iterator which points to the element pointed to by last prior to the other elements being erased. If no such element exists, end() is returned.

void pop_back();

Requires: !empty()
Throws: Nothing.
Effects: Equivalent to erase(size() - 1, 1).
21.4.6.6 basic_string::replace

basic_string&
    replace(size_type pos1, size_type n1,
            const basic_string& str);

1 Requires: pos1 <= size().
2 Throws: out_of_range if pos1 > size().
3 Effects: Calls replace(pos1, n1, str.data(), str.size()).
4 Returns: *this.

basic_string&
    replace(size_type pos1, size_type n1,
            const basic_string& str,
            size_type pos2, size_type n2 = npos);

5 Requires: pos1 <= size() and pos2 <= str.size().
6 Throws: out_of_range if pos1 > size() or pos2 > str.size().
7 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).
8 Returns: *this.

basic_string&
    replace(size_type pos1, size_type n1, const charT* s, size_type n2);

9 Requires: pos1 <= size() and s points to an array of at least n2 elements of charT.
10 Throws: out_of_range if pos1 > size() or length_error if the length of the resulting string would exceed max_size() (see below).
11 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 n2 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.
12 Returns: *this.

basic_string&
    replace(size_type pos, size_type n, const charT* s);

13 Requires: pos <= size() and s points to an array of at least traits::length(s) + 1 elements of charT.
14 Effects: Calls replace(pos, n, s, traits::length(s)).
15 Returns: *this.

basic_string&
    replace(size_type pos1, size_type n1,
            size_type n2, charT c);

16 Effects: Equivalent to replace(pos1, n1, basic_string(n2, c)).
17 Returns: *this.

basic_string& replace(const_iterator i1, const_iterator i2, const basic_string& str);
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) elements of charT.
Effects: Calls replace(i1 - begin(), i2 - i1, s, traits::length(s)).
Returns: *this.

basic_string& replace(const_iterator i1, const_iterator i2, size_type n, charT c);

Requires: [begin(), i1) and [i1, i2) are valid ranges.
Effects: Calls replace(i1 - begin(), i2 - i1, basic_string(n, c)).
Returns: *this.

template<class InputIterator>
basic_string& replace(const_iterator i1, const_iterator i2, InputIterator j1, InputIterator j2);

Requires: [begin(), i1), [i1, i2) and [j1, j2) are valid ranges.
Effects: Calls replace(i1 - begin(), i2 - i1, basic_string(j1, j2)).
Returns: *this.

basic_string& replace(const_iterator i1, const_iterator i2, initializer_list<charT> il);

Requires: [begin(), i1) and [i1, i2) are valid ranges.
Effects: Calls replace(i1 - begin(), i2 - i1, il.begin(), il.size()).
Returns: *this.

21.4.6.7 basic_string::copy

size_type copy(charT* s, size_type n, size_type pos = 0) const;

Requires: pos <= size()

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. s shall designate an array of at least rlen elements.
The function then replaces the string designated by s with a string of length rlen whose elements are a copy of the string controlled by *this beginning at position pos.
The function does not append a null object to the string designated by s.

Returns: rlen.
21.4.6.8 basic_string::swap

```cpp
void swap(basic_string& s)
    noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
               allocator_traits<Allocator>::is_always_equal::value);
```

1. **Postcondition:** *this contains the same sequence of characters that was in s, s contains the same sequence of characters that was in *this.
2. **Throws:** Nothing.
3. **Complexity:** Constant time.

21.4.7 basic_string string operations

21.4.7.1 basic_string accessors

```cpp
const charT* c_str() const noexcept;
const charT* data() const noexcept;
```

1. **Returns:** A pointer p such that p + i == &operator[](i) for each i in [0,size()].
2. **Complexity:** Constant time.
3. **Requires:** The program shall not alter any of the values stored in the character array.

```cpp
allocator_type get_allocator() const noexcept;
```

4. **Returns:** A copy of the Allocator object used to construct the string or, if that allocator has been replaced, a copy of the most recent replacement.

21.4.7.2 basic_string::find

```cpp
size_type find(const basic_string& str,
              size_type pos = 0) const noexcept;
```

1. **Effects:** Determines the lowest position xpos, if possible, such that both of the following conditions obtain:
   - (1.1) pos <= xpos and xpos + str.size() <= size();
   - (1.2) traits::eq(str.at(xpos+i), str.at(I)) for all elements I of the string controlled by str.
2. **Returns:** xpos if the function can determine such a value for xpos. Otherwise, returns npos.
3. **Remarks:** Uses traits::eq().

```cpp
size_type find(const charT* s, size_type pos = 0) const;
```

4. **Returns:** find(basic_string(s,n),pos).

```cpp
size_type find(const charT* s, size_type pos = 0) const;
```

5. **Requires:** s points to an array of at least traits::length(s) + 1 elements of charT.
6. **Returns:** find(basic_string(s), pos).

```cpp
size_type find(charT c, size_type pos = 0) const;
```

7. **Returns:** find(basic_string(1,c), pos).
21.4.7.3 basic_string::rfind

size_type rfind(const basic_string& str, 
    size_type pos = npos) const noexcept;

1  Effects: Determines the highest position xpos, if possible, such that both of the following conditions obtain:
2    (1.1) xpos <= pos and xpos + str.size() <= size();
3    (1.2) traits::eq(at(xpos+I), str.at(I)) for all elements I of the string controlled by str.
4  Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.
5  Remarks: Uses traits::eq().

size_type rfind(const charT* s, size_type pos, size_type n) const;
6  Returns: rfind(basic_string(s, n), pos).

size_type rfind(const charT* s, size_type pos = npos) const;
7  Requires: s points to an array of at least traits::length(s) + 1 elements of charT.
8  Returns: rfind(basic_string(s), pos).

size_type rfind(charT c, size_type pos = npos) const;
9  Returns: rfind(basic_string(1,c),pos).

21.4.7.4 basic_string::find_first_of

size_type 
    find_first_of(const basic_string& str, 
        size_type pos = 0) const noexcept;
1  Effects: Determines the lowest position xpos, if possible, such that both of the following conditions obtain:
2    (1.1) pos <= xpos and xpos < size();
3    (1.2) traits::eq(at(xpos), str.at(I)) for some element I of the string controlled by str.
4  Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.
5  Remarks: Uses traits::eq().

size_type 
    find_first_of(const charT* s, size_type pos, size_type n) const;
6  Returns: find_first_of(basic_string(s, n), pos).

size_type find_first_of(const charT* s, size_type pos = 0) const;
7  Requires: s points to an array of at least traits::length(s) + 1 elements of charT.
8  Returns: find_first_of(basic_string(s), pos).

size_type find_first_of(charT c, size_type pos = 0) const;
9  Returns: find_first_of(basic_string(1,c), pos).
21.4.7.5 basic_string::find_last_of

```cpp
size_type
find_last_of(const basic_string& str,
             size_type pos = npos) const noexcept;
```

**Effects:** Determines the highest position `xpos`, if possible, such that both of the following conditions obtain:

1. `xpos <= pos` and `xpos < size();`
2. `traits::eq(at(xpos), str.at(I))` for some element `I` of the string controlled by `str`.

**Returns:** `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

**Remarks:** Uses `traits::eq()`.

```cpp
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;
```  
**Requires:** `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

```cpp
size_type find_first_not_of(const basic_string& str,
                           size_type pos = 0) const noexcept;
```

**Effects:** Determines the lowest position `xpos`, if possible, such that both of the following conditions obtain:

1. `pos <= xpos` and `xpos < size();`
2. `traits::eq(at(xpos), str.at(I))` for no element `I` of the string controlled by `str`.

**Returns:** `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

**Remarks:** Uses `traits::eq()`.

```cpp
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;
```  
**Requires:** `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

```cpp
size_type find_first_not_of(charT c, size_type pos = 0) const;
```  
**Returns:** `find_first_not_of(basic_string(1, c), pos)`.

§ 21.4.7.6
21.4.7.7 basic_string::find_last_not_of

size_type find_last_not_of(const basic_string& str, size_type pos = npos) const noexcept;

Effects: Determines the highest position xpos, if possible, such that both of the following conditions obtain:
(1.1) xpos <= pos and xpos < size();
(1.2) traits::eq(at(xpos), str.at(I)) for no element I of the string controlled by str.

Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.
Remarks: Uses traits::eq().

size_type find_last_not_of(const charT* s, size_type pos, size_type n) const;

Returns: find_last_not_of(basic_string(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(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).

21.4.7.8 basic_string::substr

basic_string substr(size_type pos = 0, size_type n = npos) const;

Requires: pos <= size()

Throws: out_of_range if pos > size().

Effects: Determines the effective length rlen of the string to copy as the smaller of n and size() - pos.

Returns: basic_string(data()+pos,rlen).

21.4.7.9 basic_string::compare

int compare(const basic_string& str) const noexcept;

Effects: Determines the effective length rlen of the strings to compare as the smallest of size() and str.size(). The function then compares the two strings by calling traits::compare(data(), str.data(), rlen).

Returns: The nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table 72.

Table 72 — compare() results

<table>
<thead>
<tr>
<th>Condition</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>size() &lt; str.size()</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>size() == str.size()</td>
<td>0</td>
</tr>
<tr>
<td>size() &gt; str.size()</td>
<td>&gt; 0</td>
</tr>
</tbody>
</table>
int compare(size_type pos1, size_type n1, const basic_string& str) const;

3 Returns: basic_string(*this, pos1, n1).compare(str).

int compare(size_type pos1, size_type n1, const basic_string& str,
            size_type pos2, size_type n2 = npos) const;

4 Returns: basic_string(*this, pos1, n1).compare(basic_string(str, pos2, n2)).

int compare(const charT* s) const;

5 Returns: compare(basic_string(s)).

int compare(size_type pos, size_type n1, const charT* s) const;

6 Returns: basic_string(*this, pos, n1).compare(basic_string(s)).

int compare(size_type pos, size_type n1, const charT* s, size_type n2) const;

7 Returns: basic_string(*this, pos, n1).compare(basic_string(s, n2)).

21.4.8 basic_string non-member functions [string.nonmembers]

21.4.8.1 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);

1 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);

2 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);

3 Returns: std::move(rhs.insert(0, lhs)) [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);

§ 21.4.8.1
Returns: `basic_string<charT,traits,Allocator>(lhs) + rhs.`

Remarks: Uses `traits::length()`.

```cpp
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()`.
```

```cpp
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, 1, lhs)).`
```

```cpp
§ 21.4.8.1 680
```
21.4.8.2 operator==

\[
\begin{align*}
\text{template}\langle\text{class charT, class traits, class Allocator}\rangle \\
\quad \text{bool operator==}(\text{const basic_string}<$\text{charT,traits,Allocator}$>$\& \text{lhs}, \\
\quad \quad \quad \text{const basic_string}<$\text{charT,traits,Allocator}$>$\& \text{rhs}) \text{ noexcept}; \\
\end{align*}
\]

1 Returns: \( \text{lhs.compare(rhs)} == 0 \).

\[
\begin{align*}
\text{template}\langle\text{class charT, class traits, class Allocator}\rangle \\
\quad \text{bool operator==}(\text{const charT*} \text{lhs}, \\
\quad \quad \quad \text{const basic_string}<$\text{charT,traits,Allocator}$>$\& \text{rhs}); \\
\end{align*}
\]

2 Returns: \( \text{rhs == lhs} \).

\[
\begin{align*}
\text{template}\langle\text{class charT, class traits, class Allocator}\rangle \\
\quad \text{bool operator==}(\text{const basic_string}<$\text{charT,traits,Allocator}$>$\& \text{lhs}, \\
\quad \quad \quad \text{const charT*} \text{rhs}); \\
\end{align*}
\]

3 Requires: \( \text{rhs points to an array of at least traits::length(rhs) + 1 elements of charT} \).

4 Returns: \( \text{lhs.compare(rhs)} == 0 \).

21.4.8.3 operator!=

\[
\begin{align*}
\text{template}\langle\text{class charT, class traits, class Allocator}\rangle \\
\quad \text{bool operator!=}(\text{const basic_string}<$\text{charT,traits,Allocator}$>$\& \text{lhs}, \\
\quad \quad \quad \text{const basic_string}<$\text{charT,traits,Allocator}$>$\& \text{rhs}) \text{ noexcept}; \\
\end{align*}
\]

1 Returns: \( !(\text{lhs == rhs}) \).

\[
\begin{align*}
\text{template}\langle\text{class charT, class traits, class Allocator}\rangle \\
\quad \text{bool operator!=}(\text{const charT*} \text{lhs}, \\
\quad \quad \quad \text{const basic_string}<$\text{charT,traits,Allocator}$>$\& \text{rhs}); \\
\end{align*}
\]

2 Returns: \( \text{rhs != lhs} \).

\[
\begin{align*}
\text{template}\langle\text{class charT, class traits, class Allocator}\rangle \\
\quad \text{bool operator!=}(\text{const basic_string}<$\text{charT,traits,Allocator}$>$\& \text{lhs}, \\
\quad \quad \quad \text{const charT*} \text{rhs}); \\
\end{align*}
\]

3 Requires: \( \text{rhs points to an array of at least traits::length(rhs) + 1 elements of charT} \).

4 Returns: \( \text{lhs.compare(rhs)} != 0 \).

21.4.8.4 operator<

\[
\begin{align*}
\text{template}\langle\text{class charT, class traits, class Allocator}\rangle \\
\quad \text{bool operator<}(\text{const basic_string}<$\text{charT,traits,Allocator}$>$\& \text{lhs}, \\
\quad \quad \quad \text{const basic_string}<$\text{charT,traits,Allocator}$>$\& \text{rhs}) \text{ noexcept}; \\
\end{align*}
\]

1 Returns: \( \text{lhs.compare(rhs)} < 0 \).

\[
\begin{align*}
\text{template}\langle\text{class charT, class traits, class Allocator}\rangle \\
\quad \text{bool operator<}(\text{const charT*} \text{lhs}, \\
\quad \quad \quad \text{const basic_string}<$\text{charT,traits,Allocator}$>$\& \text{rhs}); \\
\end{align*}
\]

2 Returns: \( \text{rhs.compare(lhs)} > 0 \).

\[
\begin{align*}
\text{template}\langle\text{class charT, class traits, class Allocator}\rangle \\
\quad \text{bool operator<}(\text{const basic_string}<$\text{charT,traits,Allocator}$>$\& \text{lhs}, \\
\quad \quad \quad \text{const charT*} \text{rhs}); \\
\end{align*}
\]

3 Returns: \( \text{lhs.compare(rhs)} < 0 \).
21.4.8.5 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;
```

1 Returns: lhs.compare(rhs) > 0.

```
template<class charT, class traits, class Allocator>
bool operator> (const charT* lhs,
                const basic_string<charT,traits,Allocator>& rhs);
```

2 Returns: rhs.compare(lhs) < 0.

```
template<class charT, class traits, class Allocator>
bool operator> (const basic_string<charT,traits,Allocator>& lhs,
                const charT* rhs);
```

3 Returns: lhs.compare(rhs) > 0.

21.4.8.6 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;
```

1 Returns: lhs.compare(rhs) <= 0.

```
template<class charT, class traits, class Allocator>
bool operator<= (const charT* lhs,
                const basic_string<charT,traits,Allocator>& rhs);
```

2 Returns: rhs.compare(lhs) >= 0.

```
template<class charT, class traits, class Allocator>
bool operator<= (const basic_string<charT,traits,Allocator>& lhs,
                const charT* rhs);
```

3 Returns: lhs.compare(rhs) <= 0.

21.4.8.7 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;
```

1 Returns: lhs.compare(rhs) >= 0.

```
template<class charT, class traits, class Allocator>
bool operator>= (const charT* lhs,
                const basic_string<charT,traits,Allocator>& rhs);
```

2 Returns: rhs.compare(lhs) <= 0.

```
template<class charT, class traits, class Allocator>
bool operator>= (const basic_string<charT,traits,Allocator>& lhs,
                const charT* rhs);
```

3 Returns: lhs.compare(rhs) >= 0.
21.4.8.8 swap [string.special]

template<class charT, class traits, class Allocator>
void swap(basic_string<charT,traits,Allocator>& lhs,
        basic_string<charT,traits,Allocator>& rhs)
    noexcept(noexcept(lhs.swap(rhs)));

    Effects: Equivalent to lhs.swap(rhs);

21.4.8.9 Inserters and extractors [string.io]

template<class charT, class traits, class Allocator>
    basic_istream<charT,traits>&
        operator>>(basic_istream<charT,traits>&& is,
                 basic_string<charT,traits,Allocator>& str);

    Effects: Behaves as a formatted input function (27.7.2.2.1). After constructing a sentry object, if the
sentry converts to true, calls str.erase() and then extracts characters from is and appends them to
str as if by calling str.append(1,c). If is.width() is greater than zero, the maximum number n
of characters appended is is.width(); otherwise n is str.max_size(). Characters are extracted and
appended until any of the following occurs:
(1.1)  — n characters are stored;
(1.2)  — end-of-file occurs on the input sequence;
(1.3)  — isspace(c,is.getloc()) is true for the next available input character c.

After the last character (if any) is extracted, is.width(0) is called and the sentry object k is de-
stroyed.

If the function extracts no characters, it calls is.setstate(ios::failbit), which may throw ios_-base::failure (27.5.5.4).

Returns: is

    template<class charT, class traits, class Allocator>
        basic_ostream<charT, traits>&
            operator<<(basic_ostream<charT, traits>&& os,
                      const basic_string<charT, traits, Allocator>& str);

    Effects: Behaves as a formatted output function (27.7.3.6.1) 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 27.7.3.6.1. 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

    template<class charT, class traits, class Allocator>
        basic_istream<charT, traits>&
            getline(basic_istream<charT, traits>&& is,
                    basic_string<charT, traits, Allocator>& str, charT delim);

    template<class charT, class traits, class Allocator>
        basic_istream<charT, traits>&
            getline(basic_istream<charT, traits>&& is,
                    basic_string<charT, traits, Allocator>& str, charT delim);

Effects: Behaves as an unformatted input function (27.7.2.3), 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:
— end-of-file occurs on the input sequence (in which case, the `getline` function calls `is.setstate(ios_base::eofbit)`).

— `traits::eq(c, delim)` for the next available input character `c` (in which case, `c` is extracted but not appended) (27.5.5.4)

— `str.max_size()` characters are stored (in which case, the function calls `is.setstate(ios_base::failbit)`) (27.5.5.4)

The conditions are tested in the order shown. In any case, after the last character is extracted, the `sentry` object `k` is destroyed.

If the function extracts no characters, it calls `is.setstate(ios_base::failbit)` which may throw `ios_base::failure` (27.5.5.4).

Returns:

```cpp
template<class charT, class traits, class Allocator>
basic_istream<charT,traits>&
ggetline(basic_istream<charT,traits>& is,
basic_string<charT,traits,Allocator>& str);

template<class charT, class traits, class Allocator>
basic_istream<charT,traits>&
ggetline(basic_istream<charT,traits>&& is,
basic_string<charT,traits,Allocator>& str);
```

Returns: `getline(is,str,is.widen(’\n’))`

### 21.5 Numeric conversions

```cpp
int stoi(const string& str, size_t* idx = 0, int base = 10);
long stol(const string& str, size_t* idx = 0, int base = 10);
unsigned long stoul(const string& str, size_t* idx = 0, int base = 10);
long long stoll(const string& str, size_t* idx = 0, int base = 10);
unsigned long long stoull(const string& str, size_t* idx = 0, 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`, `strtoll`, or `strtoull` reports that no conversion could be performed. Throws `out_of_range` if `strtol`, `strtoul`, `strtoll` or `strtoull` sets `errno` to `ERANGE`, or if the converted value is outside the range of representable values for the return type.

```cpp
float stof(const string& str, size_t* idx = 0);
double stod(const string& str, size_t* idx = 0);
long double stold(const string& str, size_t* idx = 0);
```

Effects: the first two functions call `strtod(str.c_str(), ptr)` and the third function calls `strtold(str.c_str(), ptr)`. 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 `strtod` or `strtold` reports that no conversion could be performed.
Throws out_of_range if `strtod` or `strtold` sets `errno` to `ERANGE` or if the converted value is outside the range of representable values for the return type.

```cpp
string to_string(int val);
string to_string(unsigned val);
string to_string(long val);
string to_string(unsigned long val);
string to_string(long long val);
string to_string(unsigned long long val);
string to_string(float val);
string to_string(double val);
string to_string(long double val);
```

Returns: Each function returns a string object holding the character representation of the value of its argument that would be generated by calling `sprintf(buf, fmt, val)` with a format specifier of "%d", "%u", "%ld", "%lu", "%lld", "%llu", "%f", "%f", or "%Lf", respectively, where `buf` designates an internal character buffer of sufficient size.

```cpp
int stoi(const wstring& str, size_t* idx = 0, int base = 10);
long stol(const wstring& str, size_t* idx = 0, int base = 10);
unsigned long stoul(const wstring& str, size_t* idx = 0, int base = 10);
long long stoll(const wstring& str, size_t* idx = 0, int base = 10);
unsigned long long stoull(const wstring& str, size_t* idx = 0, 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.

```cpp
float stof(const wstring& str, size_t* idx = 0);
double stod(const wstring& str, size_t* idx = 0);
long double stold(const wstring& str, size_t* idx = 0);
```

Effects: the first two functions call `wcstod(str.c_str(), ptr)` and the third function calls `wcstold(str.c_str(), ptr)`. 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 `wcstod` or `wcstold` reports that no conversion could be performed. Throws out_of_range if `wcstod` or `wcstold` sets `errno` to `ERANGE`.

```cpp
wstring to_wstring(int val);
wstring to_wstring(unsigned val);
wstring to_wstring(long val);
wstring to_wstring(unsigned long val);
wstring to_wstring(long long val);
```
wstring to_wstring(unsigned long long val);
wstring to_wstring(float val);
wstring to_wstring(double val);
wstring to_wstring(long double val);

Returns: Each function returns a wstring object holding the character representation of the value of its argument that would be generated by calling swprintf(buf, buffsz, fmt, val) with a format specifier of L"%d", L"%u", L"%ld", L"%lu", L"%lld", L"%llu", L"%f", L"%f", or L"%Lf", respectively, where buf designates an internal character buffer of sufficient size buffsz.

21.6 Hash support

template <> struct hash<string>;
template <> struct hash<u16string>;
template <> struct hash<u32string>;
template <> struct hash<wstring>;

The template specializations shall meet the requirements of class template hash (20.9.13).

21.7 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]

21.8 Null-terminated sequence utilities

Table 74, 75, 76, 77, 78, and 79 describe headers <cctype>, <cwctype>, <cstring>, <cwchar>, <cstdlib>, and <cuchar>, respectively.

The contents of these headers shall be the same as the Standard C Library headers <ctype.h>, <wctype.h>, <string.h>, <wchar.h>, and <stdlib.h> and the C Unicode TR header <uchar.h>, respectively, with the following modifications:

The headers shall not define the types char16_t, char32_t, and wchar_t (2.11).

The function signature strchr(const char*, int) shall be replaced by the two declarations:

    const char* strchr(const char* s, int c);
    char* strchr(char* s, int c);

both of which shall have the same behavior as the original declaration.

The function signature strpbrk(const char*, const char*) shall be replaced by the two declarations:

    const char* strpbrk(const char* s1, const char* s2);
    char* strpbrk(char* s1, const char* s2);

§ 21.8
both of which shall have the same behavior as the original declaration.

6 The function signature `strrchr(const char*, int)` shall be replaced by the two declarations:
   ```c
   const char* strrchr(const char* s, int c);
   char* strrchr(char* s, int c);
   ```
   both of which shall have the same behavior as the original declaration.

7 The function signature `strstr(const char*, const char*)` shall be replaced by the two declarations:
   ```c
   const char* strstr(const char* s1, const char* s2);
   char* strstr(char* s1, const char* s2);
   ```
   both of which shall have the same behavior as the original declaration.

8 The function signature `memchr(const void*, int, size_t)` shall be replaced by the two declarations:
   ```c
   const void* memchr(const void* s, int c, size_t n);
   void* memchr(void* s, int c, size_t n);
   ```
   both of which shall have the same behavior as the original declaration.

9 The function signature `wcschr(const wchar_t*, wchar_t)` shall be replaced by the two declarations:
   ```c
   const wchar_t* wcschr(const wchar_t* s, wchar_t c);
   wchar_t* wcschr(wchar_t* s, wchar_t c);
   ```
   both of which shall have the same behavior as the original declaration.

10 The function signature `wcspbrk(const wchar_t*, const wchar_t*)` shall be replaced by the two declarations:
    ```c
    const wchar_t* wcspbrk(const wchar_t* s1, const wchar_t* s2);
    wchar_t* wcspbrk(wchar_t* s1, const wchar_t* s2);
    ```
    both of which shall have the same behavior as the original declaration.

11 The function signature `wcsrchr(const wchar_t*, wchar_t)` shall be replaced by the two declarations:
    ```c
    const wchar_t* wcsrchr(const wchar_t* s, wchar_t c);
    wchar_t* wcsrchr(wchar_t* s, wchar_t c);
    ```
    both of which shall have the same behavior as the original declaration.

12 The function signature `wcsstr(const wchar_t*, const wchar_t*)` shall be replaced by the two declarations:
    ```c
    const wchar_t* wcsstr(const wchar_t* s1, const wchar_t* s2);
    wchar_t* wcsstr(wchar_t* s1, const wchar_t* s2);
    ```
    both of which shall have the same behavior as the original declaration.

13 The function signature `wmemchr(const wchar_t*, int, size_t)` shall be replaced by the two declarations:
    ```c
    const wchar_t* wmemchr(const wchar_t* s, wchar_t c, size_t n);
    wchar_t* wmemchr(wchar_t* s, wchar_t c, size_t n);
    ```
    both of which shall have the same behavior as the original declaration.

14 The functions `strerror` and `strtok` are not required to avoid data races (17.6.5.9).

15 Calling the functions listed in Table 73 with an `mbstate_t*` argument of NULL may introduce a data race (17.6.5.9) with other calls to these functions with an `mbstate_t*` argument of NULL.

See also: ISO C 7.3, 7.10.7, 7.10.8, and 7.11. Amendment 1 4.4, 4.5, and 4.6.
Table 73 — Potential `mbstate_t` data races

<table>
<thead>
<tr>
<th>mbrelen</th>
<th>mbrtowc</th>
<th>mbsrtowc</th>
<th>mbtowc</th>
<th>wcrtomb</th>
<th>wcsrtomb</th>
<th>wctomb</th>
</tr>
</thead>
</table>

Table 74 — Header `<cctype>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functions:</strong></td>
<td></td>
</tr>
<tr>
<td>isalnum</td>
<td>isblank</td>
</tr>
<tr>
<td>tolower</td>
<td>isalpha</td>
</tr>
<tr>
<td>toupper</td>
<td>iscntrl</td>
</tr>
</tbody>
</table>

Table 75 — Header `<cwctype>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Macro:</strong></td>
<td>WEOF</td>
</tr>
<tr>
<td><strong>Types:</strong></td>
<td>wctrans_t</td>
</tr>
<tr>
<td><strong>Functions:</strong></td>
<td></td>
</tr>
<tr>
<td>iswalnum</td>
<td>iswctype</td>
</tr>
<tr>
<td>iswalpha</td>
<td>iswdigit</td>
</tr>
<tr>
<td>iswblank</td>
<td>iswgraph</td>
</tr>
<tr>
<td>iswcntrl</td>
<td>iswlower</td>
</tr>
</tbody>
</table>

Table 76 — Header `<cstring>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Macro:</strong></td>
<td>NULL <code>&lt;cstring&gt;</code></td>
</tr>
<tr>
<td><strong>Type:</strong></td>
<td>size_t <code>&lt;cstring&gt;</code></td>
</tr>
<tr>
<td><strong>Functions:</strong></td>
<td></td>
</tr>
<tr>
<td>memchr</td>
<td>strcat</td>
</tr>
<tr>
<td>memcmp</td>
<td>strchr</td>
</tr>
<tr>
<td>memcpy</td>
<td>strcmp</td>
</tr>
<tr>
<td>memmove</td>
<td>strcoll</td>
</tr>
<tr>
<td>memset</td>
<td>strcpy</td>
</tr>
</tbody>
</table>
Table 77 — Header `<cwchar>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td>NULL WCHAR_MAX WCHAR_MIN WEOF</td>
</tr>
<tr>
<td>Types:</td>
<td>mbstate_t wint_t size_t tm</td>
</tr>
<tr>
<td>Functions:</td>
<td>btowc mbsinit vwscanf wcsncmp wcstoull</td>
</tr>
<tr>
<td></td>
<td>fgetwc mbartowcs wcrtomb wcsprk wcstoul</td>
</tr>
<tr>
<td></td>
<td>fgetws putwchar wcscat wcsrchr wcsxfrm</td>
</tr>
<tr>
<td></td>
<td>fputwc putwc wcschr wcstombs wctob</td>
</tr>
<tr>
<td></td>
<td>fputws swprintf wcsncmp wcsspn wmemchr</td>
</tr>
<tr>
<td></td>
<td>fwrite swscanf wcscolm wcsstr wmemcmp</td>
</tr>
<tr>
<td></td>
<td>fwprintf ungetwc wcsncpy wcstod wmemcpy</td>
</tr>
<tr>
<td></td>
<td>fwscanf vfscanf wcsncpy wcstof wmemmove</td>
</tr>
<tr>
<td></td>
<td>getwchar vfscanf wcsftime wcstok wmemset</td>
</tr>
<tr>
<td></td>
<td>getwc vsprintf wcslen wcstold wprintf</td>
</tr>
<tr>
<td></td>
<td>mbrlen vsscanf wcsncmp wcstoll wcharf</td>
</tr>
<tr>
<td></td>
<td>mbtowc vwprintf wcsncmp wcstol</td>
</tr>
</tbody>
</table>

Table 78 — Header `<cstdlib>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td>MB_CUR_MAX</td>
</tr>
<tr>
<td>Functions:</td>
<td>atof mblen strtof strtoul</td>
</tr>
<tr>
<td></td>
<td>atoi mbtowc strtol strtoull</td>
</tr>
<tr>
<td></td>
<td>atol mbstowcs strtold wctomb</td>
</tr>
<tr>
<td></td>
<td>atoll strtod strtoll wcstombs</td>
</tr>
</tbody>
</table>

Table 79 — Header `<cuchar>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td><strong>STDC_UTF_16</strong> <strong>STDC_UTF_32</strong></td>
</tr>
<tr>
<td>Functions:</td>
<td>mbtocol16 c16rtomb</td>
</tr>
<tr>
<td></td>
<td>mbtocol32 c32rtomb</td>
</tr>
</tbody>
</table>
22 Localization library [localization]

22.1 General [localization.general]

1 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.

2 The following subclauses describe components for locales themselves, the standard facets, and facilities from the ISO C library, as summarized in Table 80.

Table 80 — Localization library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.3 Locales</td>
<td><code>&lt;locale&gt;</code></td>
</tr>
<tr>
<td>22.4 Standard locale Categories</td>
<td><code>&lt;locale&gt;</code></td>
</tr>
<tr>
<td>22.5 Standard code conversion facets</td>
<td><code>&lt;codecvt&gt;</code></td>
</tr>
<tr>
<td>22.6 C library locales</td>
<td><code>&lt;clocale&gt;</code></td>
</tr>
</tbody>
</table>

22.2 Header `<locale>` synopsis [locale.syn]

namespace std {

// 22.3.1, locale:
  class locale;
  template <class Facet> const Facet& use_facet(const locale&);
  template <class Facet> bool has_facet(const locale&) noexcept;

// 22.3.3, 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 isalpha (charT c, const locale& loc);
  template <class charT> bool isgraph (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);

// 22.4.1, ctype:
  class ctype_base;

// 22.3.3.1, 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 isalpha (charT c, const locale& loc);
  template <class charT> bool isgraph (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);

// 22.4.1, ctype:
  class ctype_base;

§ 22.2

690
The header `<locale>` defines classes and declares functions that encapsulate and manipulate the information peculiar to a locale.\(^1\)

22.3 Locales

22.3.1 Class locale

namespace std {
   class locale {
      public:
         // types:
         class facet;
         class id;
         typedef int category;

\(^1\) In this subclause, the type name `struct tm` is an incomplete type that is defined in `<ctime>`.
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);
locale(const locale& other, const locale& one, category);
~locale();  // not virtual
const locale& operator=(const locale& other) noexcept;
template <class Facet> locale combine(const locale& other) const;

// locale operations:
basic_string<char> name() const;
bool operator==(const locale& other) const;
bool operator!=(const locale& other) const;
template <class charT, class traits, class Allocator>
  bool operator()(const basic_string<charT,traits,Allocator>& s1,
                 const basic_string<charT,traits,Allocator>& s2) const;

// global locale objects:
static locale global(const locale&);
static const locale& classic();

1 Class locale implements a type-safe polymorphic set of facets, indexed by facet type. In other words, a facet has a dual role: in one sense, it’s just a class interface; at the same time, it’s an index into a locale’s set of facets.

2 Access to the facets of a locale is via two function templates, use_facet<> and has_facet<>.

3 [ Example: An iostream operator<< might be implemented as:237

```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, fill(), err, &tmbuf, 'x');
    s.setstate(err);  // might throw
  }
}
```

237) Note that in the call to put the stream is implicitly converted to an ostreambuf_iterator<charT,traits>.

§ 22.3.1
4 In the call to `use_facet<Facet>(loc)`, the type argument chooses a facet, making available all members of the named type. If `Facet` is not present in a locale, it throws the standard exception `bad_cast`. A C++ program can check if a locale implements a particular facet with the function template `has_facet<Facet>()`. User-defined facets may be installed in a locale, and used identically as may standard facets (22.4.8).

5 [ Note: All locale semantics are accessed via `use_facet<>` and `has_facet<>`, except that:

(5.1) — A member operator template `operator()(const basic_string<C, T, A>&, const basic_string<C, T, A>&)` is provided so that a locale may be used as a predicate argument to the standard collections, to collate strings.

(5.2) — Convenient global interfaces are provided for traditional `ctype` functions such as `isdigit()` and `isspace()`, so that given a locale object `loc` a C++ program can call `isspace(c, loc)`. (This eases upgrading existing extractors (27.7.2.2).) — end note ]

6 Once a facet reference is obtained from a locale object by calling `use_facet<>`, that reference remains usable, and the results from member functions of it may be cached and re-used, as long as some locale object refers to that facet.

7 In successive calls to a locale facet member function on a facet object installed in the same locale, the returned result shall be identical.

8 A `locale` constructed from a name string (such as "POSIX"), or from parts of two named locales, has a name; all others do not. Named locales may be compared for equality; an unnamed locale is equal only to (copies of) itself. For an unnamed locale, `locale::name()` returns the string "*".

9 Whether there is one global locale object for the entire program or one global locale object per thread is implementation-defined. Implementations should provide one global locale object per thread. If there is a single global locale object for the entire program, implementations are not required to avoid data races on it (17.6.5.9).

22.3.1.1 locale types

22.3.1.1.1 Type `locale::category`

typedef int category;

1 Valid `category` values include the `locale` member bitmask elements `collate`, `ctype`, `monetary`, `numeric`, `time`, and `messages`, each of which represents a single locale category. In addition, `locale` member bitmask constant `none` is defined as zero and represents no category. And `locale` member bitmask constant `all` is defined such that the expression

\[
\text{(collate | ctype | monetary | numeric | time | messages | all)} \equiv \text{all}
\]

is true, and represents the union of all categories. Further, the expression \( X \mid Y \), where \( X \) and \( Y \) each represent a single category, represents the union of the two categories.

2 `locale` member functions expecting a `category` argument require one of the `category` values defined above, or the union of two or more such values. Such a `category` value identifies a set of locale categories. Each locale category, in turn, identifies a set of locale facets, including at least those shown in Table 81.

3 For any locale `loc` either constructed, or returned by `locale::classic()`, and any facet `Facet` shown in Table 81, `has_facet<Facet>(loc)` is true. Each `locale` member function which takes a `locale::category` argument operates on the corresponding set of facets.
Table 81 — Locale category facets

<table>
<thead>
<tr>
<th>Category</th>
<th>Includes facets</th>
</tr>
</thead>
<tbody>
<tr>
<td>collate</td>
<td>collate&lt;char&gt;, collate&lt;wchar_t&gt;</td>
</tr>
<tr>
<td>ctype</td>
<td>ctype&lt;char&gt;, ctype&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt&lt;char,char,mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt&lt;char16_t,char,mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt&lt;char32_t,char,mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt&lt;wchar_t,char,mbstate_t&gt;</td>
</tr>
<tr>
<td>monetary</td>
<td>moneypunct&lt;char&gt;, moneypunct&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>moneypunct&lt;char,true&gt;, moneypunct&lt;wchar_t,true&gt;</td>
</tr>
<tr>
<td></td>
<td>money_get&lt;char&gt;, money_get&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>money_put&lt;char&gt;, money_put&lt;wchar_t&gt;</td>
</tr>
<tr>
<td>numeric</td>
<td>numpunct&lt;char&gt;, numpunct&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>num_get&lt;char&gt;, num_get&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>num_put&lt;char&gt;, num_put&lt;wchar_t&gt;</td>
</tr>
<tr>
<td>time</td>
<td>time_get&lt;char&gt;, time_get&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put&lt;char&gt;, time_put&lt;wchar_t&gt;</td>
</tr>
<tr>
<td>messages</td>
<td>messages&lt;char&gt;, messages&lt;wchar_t&gt;</td>
</tr>
</tbody>
</table>

4 An implementation is required to provide those specializations for facet templates identified as members of a category, and for those shown in Table 82.

5 The provided implementation of members of facets num_get<charT> and num_put<charT> calls use_facet<F>(l) (1) only for facet F of types numpunct<charT> and ctype<charT>, and for locale l the value obtained by calling member getloc() on the ios_base& argument to these functions.

6 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 Output Iterator, respectively (24.2). 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.

22.3.1.1.2 Class locale::facet

```
namespace std {
    class locale::facet {
        protected:
            explicit facet(size_t refs = 0);
            virtual ~facet();
            facet(const facet&) = delete;
            void operator=(const facet&) = delete;
        }
    }
```

1 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.

2 The refs argument to the constructor is used for lifetime management.

(2.1) — 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,
Table 82 — 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>

the implementation never destroys the facet.

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.

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.

22.3.1.1.3 Class `locale::id`

namespace std {
    class locale::id {
    public:
        id();
        void operator=(const id&) = delete;
        id(const id&) = delete;
    }
}

The class `locale::id` provides identification of a locale facet interface, used as an index for lookup and to encapsulate initialization.
2 [Note: 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 (3.6.2). — end note]

22.3.1.2 locale constructors and destructor

```cpp
locale() noexcept;
```

Default constructor: a snapshot of the current global locale.

```cpp
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;
```

```cpp
Effects: Constructs a locale which is a copy of `other`.
```

```cpp
explicit locale(const char* std_name);
```

```cpp
Effects: Constructs a locale using standard C locale names, e.g., "POSIX". The resulting locale implements semantics defined to be associated with that name.
```

```cpp
Throws: runtime_error if the argument is not valid, or is null.
```

```cpp
Remarks: The set of valid string argument values is "C", ",", and any implementation-defined values.
```

```cpp
explicit locale(const string& std_name);
```

```cpp
Effects: The same as `locale(std_name.c_str())`.
```

```cpp
locale(const locale& other, const char* std_name, category);
```

```cpp
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)`.
```

```cpp
Throws: runtime_error if the argument is not valid, or is null.
```

```cpp
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);
```

```cpp
Effects: The same as `locale(other, std_name.c_str(), cat)`.
```

```cpp
template <class Facet> locale(const locale& other, Facet* f);
```

```cpp
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`.
```

```cpp
Remarks: The resulting locale has no name.
```

```cpp
locale(const locale& other, const locale& one, category cats);
```

```cpp
Effects: Constructs a locale incorporating all facets from the first argument except those that implement `cats`, which are instead incorporated from the second argument.
```

```cpp
Remarks: The resulting locale has a name if and only if the first two arguments have names.
```

```cpp
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.

### 22.3.1.3 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.

```cpp
detail::basic_string<char> name() const;  
```

**Returns:** The name of *this, if it has one; otherwise, the string "*". If *this has a name, then locale(name().c_str()) is equivalent to *this. Details of the contents of the resulting string are otherwise implementation-defined.

### 22.3.1.4 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:** The result of the expression: !(*this == other).

**template <class charT, class traits, class Allocator>**

**bool operator()(const basic_string<charT,traits,Allocator>& s1,**

```cpp
const basic_string<charT,traits,Allocator>& s2) const;  
```

**Effects:** Compares two strings according to the collate<charT> facet.

**Remarks:** This member operator template (and therefore locale itself) satisfies requirements for a comparator predicate template argument (Clause 25) applied to strings.

**Returns:** The result of the following expression:

```cpp
use_facet< collate<charT> >(*this).compare
    (s1.data(), s1.data()+s1.size(), s2.data(), s2.data()+s2.size()) < 0;  
```

**Example:** A vector of strings v can be collated according to collation rules in locale loc simply by (25.4.1, 23.3.6):

```cpp
std::sort(v.begin(), v.end(), loc);  
```

— end example [ ]

§ 22.3.1.4 697
22.3.1.5  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

       std::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 22.6 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.

22.3.2  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 22.3.1.1.2.
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.
5
    template <class Facet> bool has_facet(const locale& loc) noexcept;
6
   Returns: True if the facet requested is present in loc; otherwise false.

22.3.3  Convenience interfaces

22.3.3.1  Character classification

    template <class charT> bool isspace (charT c, const locale& loc);
1
    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);
1
   Each of these functions isSF 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 (22.4.1).\(^{238}\)

### 22.3.3.2 Conversions

#### 22.3.3.2.1 Character conversions

```cpp
template <class charT> charT toupper(charT c, const locale& loc);
Returns: use_facet<ctype<charT> >(loc).toupper(c).
```

```cpp
template <class charT> charT tolower(charT c, const locale& loc);
Returns: use_facet<ctype<charT> >(loc).tolower(c).
```

#### 22.3.3.2.2 String conversions

1 Class template `wstring_convert` performs conversions between a wide string and a byte string. It lets you specify a code conversion facet (like class template `codecvt`) to perform the conversions, without affecting any streams or locales. [Example: If you want to use the code conversion facet `codecvt_utf8` to output to `cout` a UTF-8 multibyte sequence corresponding to a wide string, but you don’t want to alter the locale for `cout`, you can write something like:

```cpp
wstring_convert<std::codecvt_utf8<wchar_t>> myconv;
std::string mbstring = myconv.to_bytes(L"Hello\n");
std::cout << mbstring;
```

— end example ]

2 Class template `wstring_convert` synopsis

```cpp
namespace std {
    template<class Codecvt, class Elem = wchar_t, 
        class Wide_alloc = std::allocator<Elem>, 
        class Byte_alloc = std::allocator<char> > class wstring_convert {
        public:
            typedef std::basic_string<char, char_traits<char>, Byte_alloc> byte_string;
            typedef std::basic_string<Elem, char_traits<Elem>, Wide_alloc> wide_string;
            typedef typename Codecvt::state_type state_type;
            typedef typename wide_string::traits_type::int_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);
        }
    }
}
```

\(^{238}\) 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`. 

§ 22.3.3.2.2
The class template describes an object that controls conversions between wide string objects of class `std::basic_string<Elem, char_traits<Elem>, Wide_alloc>` and byte string objects of class `std::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 `std::codecvt<Elem, char, std::mbstate_t>`.

An object of this class template stores:

- (4.1) `byte_err_string` — a byte string to display on errors
- (4.2) `wide_err_string` — a wide string to display on errors
- (4.3) `cvtptr` — a pointer to the allocated conversion object (which is freed when the `wstring_convert` object is destroyed)
- (4.4) `cvtstate` — a conversion state object
- (4.5) `cvtcount` — a conversion count

```cpp
typedef std::basic_string<char, char_traits<char>, Byte_alloc> byte_string;
```

The type shall be a synonym for `std::basic_string<char, char_traits<char>, Byte_alloc>`.

```cpp
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:

- (8.1) 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.
- (8.2) 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 `std::range_error`.
typedef typename wide_string::traits_type::int_type int_type;

The type shall be a synonym for wide_string::traits_type::int_type.

state_type state() const;
  returns cvtstate.

typedef typename Codecvt::state_type state_type;

The type shall be a synonym for Codecvt::state_type.

byte_string to_bytes(Elem wchar);
byte_string to_bytes(const Elem* wptr);
byte_string to_bytes(const wide_string& wstr);
byte_string to_bytes(const Elem* first, const Elem* last);

  Effects: The first member function shall convert the single-element sequence wchar to a byte string. The second member function shall convert the null-terminated sequence beginning at wptr to a byte string. The third member function shall convert the sequence stored in wstr to a byte string. The fourth member function shall convert the sequence defined by the range [first,last) to a byte string.

In all cases:
  — 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 std::range_error.

typedef std::basic_string<Elem, char_traits<Elem>, Wide_alloc> wide_string;

The type shall be a synonym for std::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.
22.3.3.2.3 Buffer conversions

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.

Class template `wbuffer_convert` synopsis

```cpp
namespace std {
  template<class Codecvt,
           class Elem = wchar_t,
           class Tr = std::char_traits<Elem> >
  class wbuffer_convert
    : public std::basic_streambuf<Elem, Tr> {
    public:
      typedef typename Codecvt::state_type state_type;

      explicit wbuffer_convert(std::streambuf* bytebuf = 0,
                               Codecvt* pcvt = new Codecvt,
                               state_type state = state_type());
      ~wbuffer_convert();
      wbuffer_convert(const wbuffer_convert&) = delete;
      wbuffer_convert& operator=(const wbuffer_convert&) = delete;
      std::streambuf* rdbuf() const;
      std::streambuf* rdbuf(std::streambuf* bytebuf);

      state_type state() const;

    private:
      std::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 `std::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 `std::codecvt<Elem, char, std::mbstate_t>`. An object of this class template stores:

- `bufptr` — a pointer to its underlying byte stream buffer
- `cvtptr` — a pointer to the allocated conversion object (which is freed when the `wbuffer_convert` object is destroyed)
- `cvtstate` — a conversion state object

Returns: `cvtstate`.

```cpp
std::streambuf* rdbuf() const;
```
Returns: bufptr.

std::streambuf* rdbuf(std::streambuf* bytebuf);

Effects: stores bytebuf in bufptr.

Returns: The previous value of bufptr.

typedef typename Codecvt::state_type state_type;

The type shall be a synonym for Codecvt::state_type.

explicit wbuffer_convert(std::streambuf* bytebuf = 0,
                        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.

22.4 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 (27.5.3). Those functions which need to use
other facets call its member getloc() to retrieve the locale imbued there. Formatting facets use the character
argument fill to fill out the specified width where necessary.

The put() members make no provision for error reporting. (Any failures of the OutputIterator argument
must be extracted from the returned iterator.) The get() members take an ios_base::iostate& argument
whose value they ignore, but set to ios_base::failbit in case of a parse error.

Within this clause it is unspecified whether one virtual function calls another virtual function.

22.4.1 The ctype category

namespace std {

  class ctype_base {
    protected:
      typedef T mask;

      // 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;

  };
}

§ 22.4.1
1 The type mask is a bitmask type (17.5.2.1.3).

### 22.4.1.1 Class template ctype

```cpp
namespace std {
    template <class charT>
    class ctype : public locale::facet, public ctype_base {
    public:
        typedef charT char_type;

        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 topper(charT c) const;
        const charT* topper(const charT* low, const charT* high) const;
        charT tolower(charT c) const;
        const charT* tolower(const charT* low, const charT* high) const;
        charT widen(char c) const;
        const char* widen(const char* low, const char* high, charT* to) const;
        char narrow(charT c, char dfault) const;
        const charT* narrow(const charT* low, const charT*, 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_topper(charT) const;
        virtual const charT* do_topper(const charT* low, const charT* high) const;
        virtual charT do_tolower(charT) const;
        virtual const charT* do_tolower(const charT* low, const charT* high) const;
        virtual charT do_widen(char) const;
        virtual const char* do_widen(const char* low, const char* high, charT* dest) const;
        virtual char do_narrow(charT c, char dfault) const;
        virtual const charT* do_narrow(const charT* low, const charT*, 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 81 (22.3.1.1.1), namely `ctype<char>` and `ctype<wchar_t>` implement
character classing appropriate to the implementation’s native character set.

### 22.4.1.1.1 ctype members

```cpp
bool is(mask m, charT c) const;
const charT* is(const charT* low, const charT* high,
    mask* vec) const;

1 Returns: do_is(m,c) or do_is(low,high,vec)
```

```cpp
const charT* scan_is(mask m,
    const charT* low, const charT* high) const;

2 Returns: do_scan_is(m,low,high)
```

```cpp
const charT* scan_not(mask m,
    const charT* low, const charT* high) const;

3 Returns: do_scan_not(m,low,high)
```

```cpp
charT toupper(charT) const;
const charT* toupper(charT* low, const charT* high) const;

4 Returns: do_toupper(c) or do_toupper(low,high)
```

```cpp
charT tolower(charT c) const;
const charT* tolower(charT* low, const charT* high) const;

5 Returns: do_tolower(c) or do_tolower(low,high)
```

```cpp
charT widen(char c) const;
const char* widen(const char* low, const char* high, charT* to) const;

6 Returns: do_widen(c) or do_widen(low,high,to)
```

```cpp
char narrow(charT c, char dfault) const;
const charT* narrow(const charT* low, const charT* high, charT* to) const;

7 Returns: do_narrow(c,dfault) or do_narrow(low,high,dfault,to)
```

### 22.4.1.1.2 ctype virtual functions

```cpp
bool do_is(mask m, charT c) const;
const charT* do_is(const charT* low, const charT* high,
    mask* vec) const;

1 Effects: Classifies a character or sequence of characters. For each argument character, identifies a value \( M \) of type `ctype_base::mask`. The second form identifies a value \( M \) of type `ctype_base::mask` for each \( *p \) where \( (low<=p && p<high) \), and places it into `vec[p-low]`.

2 Returns: The first form returns the result of the expression \( (M & m) \neq 0 \); i.e., `true` if the character has the characteristics specified. The second form returns `high`.

const charT* do_scan_is(mask m,
    const charT* low, const charT* high) const;

3 Effects: Locates a character in a buffer that conforms to a classification \( m \).

4 Returns: The smallest pointer \( p \) in the range \( [low, high) \) such that `is(m,*p)` would return `true`; otherwise, returns `high`.

§ 22.4.1.1.2 705
const charT* do_scan_not(mask m, const charT* low, const charT* high) const;

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 [low, high) such that is(m, *p) would return false; otherwise, returns high.

charT do_toupper(charT c) const;
const charT* do_toupper(charT* low, const charT* high) const;

Effects: Converts a character or characters to upper case. The second form replaces each character *p in the range [low, high) for which a corresponding upper-case character exists, with that character.

Returns: The first form returns the corresponding upper-case character if it is known to exist, or its argument if not. The second form returns high.

charT do_tolower(charT c) const;
const charT* do_tolower(charT* low, const charT* high) const;

Effects: Converts a character or characters to lower case. The second form replaces each character *p in the range [low, high) and for which a corresponding lower-case character exists, with that character.

Returns: The first form returns the corresponding lower-case character if it is known to exist, or its argument if not. The second form returns high.

charT do_widen(char c) const;
const charT* do_widen(const charT* low, const charT* high, charT* dest) const;

Effects: Applies the simplest reasonable transformation from a char value or sequence of char values to the corresponding charT value or values. The only characters for which unique transformations are required are those in the basic source character set (2.3).

For any named ctype category with a ctype<charT> facet ctc and valid ctype_base::mask value M, (ctc.is(M, c) || !is(M, do_widen(c))) is true. The second form transforms each character *p in the range [low, high), placing the result in dest[p-low].

Returns: The first form returns the transformed value. The second form returns high.

char do_narrow(charT c, char dfault) const;
const charT* do_narrow(const charT* low, const charT* high, char dfault, char* dest) const;

Effects: Applies the simplest reasonable transformation from a charT value or sequence of charT values to the corresponding char value or values.

For any character c in the basic source character set (2.3) the transformation is such that do_widen(do_narrow(c, 0)) == c

For any named ctype category with a ctype<char> facet ctc however, and ctype_base::mask value M, (is(M, c) || !ctc.is(M, do_narrow(c, dfault)))

239) The char argument of do_widen is intended to accept values derived from character literals for conversion to the locale’s encoding.
240) In other words, the transformed character is not a member of any character classification that c is not also a member of.
is true (unless do_narrow returns dfault). In addition, for any digit character \( c \), the expression \((\text{do\_narrow}(c, \text{dfault}) - '0')\) evaluates to the digit value of the character. The second form transforms each character \( \cdot p \) in the range \([\text{low}, \text{high})\), placing the result (or \text{default} if no simple transformation is readily available) in \( \text{dest}[\cdot p - \text{low}] \).

**Returns:** The first form returns the transformed value; or \text{default} if no mapping is readily available. The second form returns \text{high}.

### 22.4.1.2 Class template ctype_byname

```cpp
namespace std {
    template <class charT>
    class ctype_byname : public ctype<charT> {
        public:
            typedef typename ctype<charT>::mask mask;
            explicit ctype_byname(const char*, size_t refs = 0);
            explicit ctype_byname(const string&, size_t refs = 0);
        protected:
            ~ctype_byname();
    };
}
```

### 22.4.1.3 ctype specializations

```cpp
namespace std {
    template <> class ctype<char> : public locale::facet, public ctype_base {
        public:
            typedef char char_type;
            explicit ctype(const mask* tab = 0, bool del = false, size_t refs = 0);
            bool is(mask m, char c) const;
            const char* is(const char* low, const char* high, mask* vec) const;
            const char* scan_is(mask m, const char* low, const char* high) const;
            const char* scan_not(mask m, const char* low, const char* high) const;
            char toupper(char c) const;
            const char* toupper(char* low, const char* high) const;
            char tolower(char c) const;
            const char* tolower(char* low, const char* high) const;
            char widen(char c) const;
            const char* widen(const char* low, const char* high, char* to) const;
            char narrow(char c, char dfault) const;
            const char* narrow(const char* low, const char* high, char dfault, char* to) const;
        static locale::id id;
        static const size_t table_size = implementation-defined;
        const mask* table() const noexcept;
        static const mask* classic_table() noexcept;
    };
```
protected:
~ctype();
virtual char do_toupper(char c) const;
virtual const char* do_toupper(char* low, const char* high) const;
virtual char dotolower(char c) const;
virtual const char* dotolower(char* low, const char* high) const;

virtual char do_widen(char c) const;
virtual const char* do_widen(const char* low, 
  const char* high, 
  char* to) const;
virtual char do_narrow(char c, char dfault) const;
virtual const char* do_narrow(const char* low, 
  const char* high, 
  char dfault, char* to) const;
};

1 A specialization ctype<char> is provided so that the member functions on type char can be implemented inline.241 The implementation-defined value of member table_size is at least 256.

22.4.1.3.1 ctype<char> destructor [facet.ctype.char.dtor]
~ctype();

1 Effects: If the constructor’s first argument was nonzero, and its second argument was true, does delete [] table().

22.4.1.3.2 ctype<char> members [facet.ctype.char.members]
1 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 = 0, bool del = false, 
  size_t refs = 0);

2 Requires: tbl either 0 or an array of at least table_size elements.

3 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;

4 Effects: The second form, for all *p in the range [low,high), assigns into vec[p-low] the value table()[(unsigned char)*p].

5 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;

6 Returns: The smallest p in the range [low,high) such that 
  table()[(unsigned char) *p] & m
is true.

241) 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>.
const char* scan_not(mask m,
    const char* low, const char* high) const;

Returns: The smallest p in the range [low,high) such that
    \text{table()}[(\text{unsigned char}) \ast p] \& m

is false.

char    toupper(char c) const;
const char* toupper(char* low, const char* high) const;

Returns: do_toupper(c) or do_toupper(low,high), respectively.

char    tolower(char c) const;
const char* tolower(char* low, const char* high) const;

Returns: do_tolower(c) or do_tolower(low,high), respectively.

char    widen(char c) const;
const char* widen(const char* low, const char* high,
    char* to) const;

Returns: do_widen(c) or do_widen(low, high, to), respectively.

char    narrow(char c, char dfault) const;
const char* narrow(const char* low, const char* high,
    char dfault, char* to) const;

Returns: do_narrow(c, dfault) or do_narrow(low, high, dfault, to), respectively.

const mask* table() const noexcept;

Returns: The first constructor argument, if it was non-zero, otherwise classic_table().

22.4.1.3.3 \texttt{ctype\langle char\rangle} static members

static const mask* classic_table() noexcept;

Returns: A pointer to the initial element of an array of size \texttt{table\_size} which represents the classifi-
cations of characters in the "C" locale.

22.4.1.3.4 \texttt{ctype\langle char\rangle} virtual functions

virtual char    do_widen(char c) const;
virtual const char* do_widen(const char* low,
    const char* high,
    char* to) const;
virtual char    do_narrow(char c, char dfault) const;
virtual const char* do_narrow(const char* low,
    const char* high,
    char dfault, char* to) const;

These functions are described identically as those members of the same name in the \texttt{ctype} class tem-
plate (22.4.1.1.1).

22.4.1.4 Class template codecvt

These functions are described identically as those members of the same name in the \texttt{ctype} class tem-
plate (22.4.1.1.1).
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:
    typedef internT intern_type;
    typedef externT extern_type;
    typedef stateT state_type;

    explicit codecvt(size_t refs = 0);
    result out(stateT& state,
               const internT* from, const internT* from_end, const internT*& from_next,
               externT* to, externT* to_end, externT*& to_next) const;
    result unshift(stateT& state,
                   externT* to, externT* to_end, externT*& to_next) const;
    result in(stateT& state,
              const externT* from, const externT* from_end, const externT*& from_next,
              internT* to, internT* to_end, internT*& to_next) const;
    int encoding() const noexcept;
    bool always_noconv() const noexcept;
    int length(stateT& state, const externT* from, const externT* end,
               size_t max) const;
    int max_length() const noexcept;

    static locale::id id;
}

protected:
    ~codecvt();
    virtual result do_out(stateT& state,
                           const internT* from, const internT* from_end, const internT*& from_next,
                           externT* to, externT* to_end, externT*& to_next) const;
    virtual result do_in(stateT& state,
                         const externT* from, const externT* from_end, const externT*& from_next,
                         internT* to, internT* to_end, internT*& to_next) const;
    virtual result do_unshift(stateT& state,
                               externT* to, externT* to_end, externT*& to_next) const;
    virtual int do_encoding() const noexcept;
    virtual bool do_always_noconv() const noexcept;
    virtual int do_length(stateT& state, const externT* from,
                           const externT* end, size_t max) const;
    virtual int do_max_length() const noexcept;
};

1 The class codecvt<internT,externT,stateT> is for use when converting from one character encoding to another, such as from wide characters to multibyte characters or between wide character encodings such as Unicode and EUC.

2 The stateT argument selects the pair of character encodings being mapped between.
The specializations required in Table 81 (22.3.1.1.1) 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<char, 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.

### 22.4.1.4.1 codecvt members

[result stateT state,
  const internT* from, const internT* from_end, const internT*& from_next,
  externT* to, externT* to_end, externT*& to_next) const;]

1. Returns: `do_out(state, from, from_end, from_next, to, to_end, to_next)`

[result stateT state,
  externT* to, externT* to_end, externT*& to_next) const;]

2. Returns: `do_unshift(state, to, to_end, to_next)`

[result stateT state,
  const externT* from, const externT* from_end, const externT*& from_next,
  internT* to, internT* to_end, internT*& to_next) const;]

3. Returns: `do_in(state, from, from_end, from_next, to, to_end, to_next)`

int encoding() const noexcept;

4. Returns: `do_encoding()`

bool always_noconv() const noexcept;

5. Returns: `do_always_noconv()`

int length(stateT& state, const externT* from, const externT* from_end,
  size_t max) const;

6. Returns: `do_length(state, from, from_end, max)`

int max_length() const noexcept;

7. Returns: `do_max_length()`

### 22.4.1.4.2 codecvt virtual functions

[result stateT state,
  const internT* from, const internT* from_end, const internT*& from_next,
  externT* to, externT* to_end, externT*& to_next) const;]

[result 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.
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.

Studies 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).

A codecvt facet that is used by basic_filebuf (27.9) shall have the property that if

\[\text{do_out}(\text{state}, \text{from}, \text{from_end}, \text{from_next}, \text{to}, \text{to_end}, \text{to_next})\]

would return ok, where from != from_end, then

\[\text{do_out}(\text{state}, \text{from}, \text{from} + 1, \text{from_next}, \text{to}, \text{to_end}, \text{to_next})\]

shall also return ok, and that if

\[\text{do_in}(\text{state}, \text{from}, \text{from_end}, \text{from_next}, \text{to}, \text{to_end}, \text{to_next})\]

would return ok, where to != to_end, then

\[\text{do_in}(\text{state}, \text{from}, \text{from_end}, \text{from_next}, \text{to}, \text{to} + 1, \text{to_next})\]

shall also return ok.242 [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]

Remarks: Its operations on state are unspecified. [Note: This argument can be used, for example, to maintain shift state, to specify conversion options (such as count only), or to identify a cache of seek offsets. — end note]

Returns: An enumeration value, as summarized in Table 83.

<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.

\[\text{result do_unshift}(\text{stateT}& \text{state}, \text{externT} \ast \text{to}, \text{externT} \ast \text{to_end}, \text{externT} \& \text{to_next}) \text{const};\]

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.

242) 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.
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 84.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ok</td>
<td>completed the sequence</td>
</tr>
<tr>
<td>partial</td>
<td>space for more than to_end-to destination elements was needed to terminate a sequence given the value of state</td>
</tr>
<tr>
<td>error</td>
<td>an unspecified error has occurred</td>
</tr>
<tr>
<td>noconv</td>
<td>no termination is needed for this state_type</td>
</tr>
</tbody>
</table>

int do_encoding() const noexcept;

Returns: -1 if the encoding of the externT sequence is state-dependent; else the constant number of externT characters needed to produce an internal character; or 0 if this number is not a constant.

bool do_always_noconv() const noexcept;

Returns: true if do_in() and do_out() return noconv for all valid argument values. codecvt<char, char, mbstate_t> returns true.

int do_length(stateT& state, const externT* from, const externT* from_end, size_t max) const;

Requires: (from<=from_end) well-defined and true; state initialized, if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.

Effects: The effect on the state argument is “as if” it called do_in(state, from, from_end, from, to, to+max, to) for to pointing to a buffer of at least max elements.

Returns: (from_next-from) where from_next is the largest value in the range [from,from_end] such that the sequence of values in the range [from,from_next) represents max or fewer valid complete characters of type internT. The specialization codecvt<char, char, mbstate_t>, returns the lesser of max and (from_end-from).

int do_max_length() const noexcept;

Returns: The maximum value that do_length(state, from, from_end, 1) can return for any valid range [from, from_end] and stateT value state. The specialization codecvt<char, char, mbstate_t>::do_max_length() returns 1.

### 22.4.1.5 Class template codecvt_byname

```cpp
namespace std {
    template <class internT, class externT, class stateT>
    class codecvt_byname : public codecvt<internT, externT, stateT> {
        public:

        243) Typically these will be characters to return the state to stateT()
        244) 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.
```
explicit codecvt_byname(const char*, size_t refs = 0);
explicit codecvt_byname(const string&, size_t refs = 0);
protected:
- codecvt_byname();
};

22.4.2 The numeric category
[category.numeric]

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.\(^{245}\)

All specifications of member functions for `num_put` and `num_get` in the subclauses of 22.4.2 only apply to the specializations required in Tables 81 and 82 (22.3.1.1.1), 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 (22.4), 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 (27.7.2.2.1, 27.7.3.6.1).

22.4.2.1 Class template `num_get` [locale.num.get]

namespace std {
    template <class charT, class InputIterator = istreambuf_iterator<charT> >
    class num_get : public locale::facet {
    public:
        typedef charT char_type;
        typedef InputIterator iter_type;

        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;

245) Parsing "-1" correctly into, e.g., an unsigned short requires that the corresponding member `get()` at least extract the sign before delegating.

§ 22.4.2.1
The facet `num_get` is used to parse numeric values from an input sequence such as an istream.

### 22.4.2.1.1 num_get members

```cpp
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, bool& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, long long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, unsigned short& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, unsigned int& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, unsigned long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, unsigned long long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, float& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, double& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, long double& val) const;
```

§ 22.4.2.1.1
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).

22.4.2.1.2 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;

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

\[
\text{fmtflags flags} = \text{str.flags();}
\]
\[
\text{fmtflags basefield} = (\text{flags} \& \text{ios_base::basefield});
\]
\[
\text{fmtflags uppercase} = (\text{flags} \& \text{ios_base::uppercase});
\]
\[
\text{fmtflags boolalpha} = (\text{flags} \& \text{ios_base::boolalpha});
\]

For conversion to an integral type, the function determines the integral conversion specifier as
indicated in Table 85. 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 86.

Table 86 — Length modifier

<table>
<thead>
<tr>
<th>Type</th>
<th>Length modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>h</td>
</tr>
<tr>
<td>unsigned short</td>
<td>h</td>
</tr>
<tr>
<td>long</td>
<td>l</td>
</tr>
<tr>
<td>unsigned long</td>
<td>l</td>
</tr>
<tr>
<td>long long</td>
<td>ll</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>ll</td>
</tr>
<tr>
<td>double</td>
<td>l</td>
</tr>
<tr>
<td>long double</td>
<td>L</td>
</tr>
</tbody>
</table>

Stage 2: If in==end then stage 2 terminates. Otherwise a charT is taken from in and local variables are initialized as if by

```c
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:

```c
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 loc.
If discard is true, then if '.' has not yet been accumulated, then the position of the character is remembered, but the character is otherwise ignored. Otherwise, if '.' has already been accumulated, the character is discarded and Stage 2 terminates. If it is not discarded, then a check is made to determine if c is allowed as the next character of an input field of the conversion specifier returned by Stage 1. If so, it is accumulated.
If the character is either discarded or accumulated then in is advanced by ++in and processing returns to the beginning of stage 2.

Stage 3: The sequence of chars accumulated in stage 2 (the field) is converted to a numeric value by the rules of one of the functions declared in the header <cstdlib>:

(3.1) For a signed integer value, the function strtoll.
(3.2) For an unsigned integer value, the function strtoull.
(3.3) For a floating-point value, the function strtold.

The numeric value to be stored can be one of:

(3.4) zero, if the conversion function fails to convert the entire field. ios_base::failbit is assigned to err.
(3.5) the most positive representable value, if the field represents a value too large positive to be represented in val. ios_base::failbit is assigned to err.
— the most negative representable value or zero for an unsigned integer type, if the field represents a value too large negative to be represented in \texttt{val}. \texttt{ios\_base::failbit} is assigned to \texttt{err}.

— the converted value, otherwise.

The resultant numeric value is stored in \texttt{val}.

Digit grouping is checked. That is, the positions of discarded separators is examined for consistency with \texttt{use\_facet\langle\texttt{numpunct<charT>;}>(loc).grouping().} If they are not consistent then \texttt{ios\_base::failbit} is assigned to \texttt{err}.

In any case, if stage 2 processing was terminated by the test for \texttt{in==end} then \texttt{err |=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\langle\texttt{numpunct<charT>;}>(str.getloc())}. Successive characters in the range \texttt{[in,end)} (see \ref{23.2.3}) are obtained and matched against corresponding positions in the target sequences only as necessary to identify a unique match. The input iterator \texttt{in} is compared to \texttt{end} only when necessary to obtain a character. If a target sequence is uniquely matched, \texttt{val} is set to the corresponding value. Otherwise \texttt{false} is stored and \texttt{ios\_base::failbit} is assigned to \texttt{err}.

The \texttt{in} iterator is always left pointing one position beyond the last character successfully matched. If \texttt{val} is set, then \texttt{err} is set to \texttt{str.goodbit}; or to \texttt{str.eofbit} if, when seeking another character to match, it is found that \texttt{(in == end)}. If \texttt{val} is not set, then \texttt{err} is set to \texttt{str.failbit}; or to \texttt{(str.failbit|str.eofbit)} if the reason for the failure was that \texttt{(in == end)}. [Example: For targets \texttt{true}: "a" and \texttt{false}: "abb", the input sequence "a" yields \texttt{val == true} and \texttt{err == str.eofbit}; the input sequence "abc" yields \texttt{err = str.failbit}, with \texttt{in} ending at the 'c' element. For targets \texttt{true}: "1" and \texttt{false}: "0", the input sequence "1" yields \texttt{val == true} and \texttt{err == str.goodbit}. For empty targets (""), any input sequence yields \texttt{err == str.failbit}. — end example]

\textbf{Returns:} \texttt{in}.

\section{22.4.2.2 Class template \texttt{num\_put}}

\begin{verbatim}
namespace std {
    template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
    class num\_put : public locale::facet {
    public:
        typedef charT char_type;
        typedef OutputIterator iter_type;

        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;
    }
}
\end{verbatim}
The facet `num_put` is used to format numeric values to a character sequence such as an ostream.

### 22.4.2.2.1 num_put members

```cpp
ter_type put(iter_type s, ios_base& f, char_type fill, double v) const;
ter_type put(iter_type s, ios_base& f, char_type fill, long double v) const;
ter_type put(iter_type s, ios_base& f, char_type fill, const void* v) const;
static locale::id id;
```

protected:
```cpp
~num_put();
virtual ter_type do_put(iter_type out, ios_base& str, char_type fill, bool v) const;
virtual ter_type do_put(iter_type out, ios_base& str, char_type fill, long v) const;
virtual ter_type do_put(iter_type out, ios_base& str, char_type fill, long long v) const;
virtual ter_type do_put(iter_type out, ios_base& str, char_type fill, unsigned long v) const;
virtual ter_type do_put(iter_type out, ios_base& str, char_type fill, unsigned long long v) const;
virtual ter_type do_put(iter_type out, ios_base& str, char_type fill, double v) const;
virtual ter_type do_put(iter_type out, ios_base& str, char_type fill, long double v) const;
virtual ter_type do_put(iter_type out, ios_base& str, char_type fill, const void* v) const;
};
```

1 The facet `num_put` is used to format numeric values to a character sequence such as an ostream.

### 22.4.2.2.2 num_put virtual functions

```cpp
iter_type do_put(iter_type out, ios_base& str, char_type fill, bool val) const;
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 Returns: `do_put(out, str, fill, val)`.
effects: Writes characters to the sequence out, formatting val as desired. In the following description, a local variable initialized with

locale loc = str.getloc();

The details of this operation occur in several stages:

(2.1) Stage 1: Determine a printf conversion specifier spec and determining the characters that would be printed by printf (27.9.2) given this conversion specifier for

printf(spec, val )

assuming that the current locale is the "C" locale.

(2.2) Stage 2: Adjust the representation by converting each char determined by stage 1 to a charT using a conversion and values returned by members of use_facet< numpunct<charT> >(str.getloc())

(2.3) Stage 3: Determine where padding is required.

(2.4) 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));

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 87.

For conversion from a floating-point type, the function determines the floating-point conversion specifier as indicated in Table 88.

For conversions from an integral or floating-point type a length modifier is added to the conversion specifier as indicated in Table 89.

The conversion specifier has the following optional additional qualifiers prepended as indicated in Table 90.
Table 87 — Integer conversions

<table>
<thead>
<tr>
<th>State</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>basefield == ios_base::oct</td>
<td>%o</td>
</tr>
<tr>
<td>(basefield == ios_base::hex) &amp;&amp; !uppercase</td>
<td>%x</td>
</tr>
<tr>
<td>(basefield == ios_base::hex)</td>
<td>%X</td>
</tr>
<tr>
<td>for a signed integral type</td>
<td>%d</td>
</tr>
<tr>
<td>for an unsigned integral type</td>
<td>%u</td>
</tr>
</tbody>
</table>

Table 88 — Floating-point conversions

<table>
<thead>
<tr>
<th>State</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>floatfield == ios_base::fixed</td>
<td>%f</td>
</tr>
<tr>
<td>floatfield == ios_base::scientific &amp; !uppercase</td>
<td>%e</td>
</tr>
<tr>
<td>floatfield == ios_base::scientific</td>
<td>%E</td>
</tr>
<tr>
<td>floatfield == (ios_base::fixed</td>
<td>ios_base::scientific) &amp; !uppercase</td>
</tr>
<tr>
<td>floatfield == (ios_base::fixed</td>
<td>ios_base::scientific)</td>
</tr>
<tr>
<td>!uppercase</td>
<td>%g</td>
</tr>
<tr>
<td>otherwise</td>
<td>%G</td>
</tr>
</tbody>
</table>

Table 89 — 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>

For conversion from a floating-point type, if floatfield != (ios_base::fixed | ios_base::scientific), str.precision() is specified as precision in the conversion specification. Otherwise, no precision is specified.

For conversion from void* the specifier is %p.

The representations at the end of stage 1 consists of the char's that would be printed by a call of printf(s, val) where s is the conversion specifier determined above.

Stage 2: Any character c other than a decimal point(.) is converted to a charT via use_facet<ctype<charT> >(loc).widen( c )

A local variable punct is initialized via

    const numpunct<charT>& punct = use_facet<numpunct<charT> >(str.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 22.4.3.1.2

Table 90 — 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>flags &amp; showpos</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>flags &amp; showbase</td>
<td>#</td>
</tr>
<tr>
<td>a floating-point type</td>
<td>flags &amp; showpos</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>flags &amp; showpoint</td>
<td>#</td>
</tr>
</tbody>
</table>
Decimal point characters(.) are replaced by `punct.decimal_point()`.

**Stage 3:** A local variable is initialized as

```c
fmtflags adjustfield = (flags & (ios_base::adjustfield));
```

The location of any padding is determined according to Table 91.

<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>after stage 1 began with 0x or 0X</td>
<td>pad after x or X</td>
</tr>
<tr>
<td>otherwise</td>
<td>pad before</td>
</tr>
</tbody>
</table>

If `str.width()` is nonzero and the number of `charT`'s in the sequence after stage 2 is less than `str.width()`, then enough fill characters are added to the sequence at the position indicated for padding to bring the length of the sequence to `str.width()`.

`str.width(0)` is called.

**Stage 4:** The sequence of `charT`'s at the end of stage 3 are output via

```c
*out++ = c
```

---

22.4.3 The numeric punctuation facet

22.4.3.1 Class template `numpunct`

```c
namespace std {
    template <class charT>
    class numpunct : public locale::facet {
    public:
        typedef charT char_type;
        typedef basic_string<charT> string_type;

        explicit numpunct(size_t refs = 0);
        char_type decimal_point() const;
        char_type thousands_sep() const;
        string grouping() const;
    }
}
```

---

246) The conversion specification `#o` generates a leading 0 which is not a padding character.
numpunct<> specifies numeric punctuation. The specializations required in Table 81 (22.3.1.1.1), 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:

\[
\text{integer ::= } [\text{sign}] \text{ units} \\
\text{sign ::= } \text{plusminus} \\
\text{plusminus ::= } '+' | '-' \\
\text{units ::= } \text{digits } [\text{thousands-sep units}] \\
\text{digits ::= } \text{digit } [\text{digits}] \\
\]

and floating-point values have:

\[
\text{floatval ::= } [\text{sign}] \text{ units } [\text{decimal-point } \text{[digits]}] [e [\text{sign}] \text{ digits}] | \\
[\text{sign}] \text{ decimal-point } \text{digits } [e [\text{sign}] \text{ digits}] \\
\text{e ::= } 'e' | 'E'
\]

where the number of digits between thousands-seps is as specified by do_grouping(). For parsing, if the digits portion contains no thousands-separators, no grouping constraint is applied.

22.4.3.1.1 numpunct members

| 1 | char_type decimal_point() const; |
| 2 | char_type thousands_sep() const; |
| 3 | string grouping() const; |
| 4 | string_type truename() const; |

\[\text{Returns: do_decimal_point()}\]

\[\text{Returns: do_thousands_sep()}\]

\[\text{Returns: do_grouping()}\]

\[\text{Returns: do_truename()} \text{ or do_falsename()}, respectively.}\]

§ 22.4.3.1.1
22.4.3.1.2 numpunct virtual functions

char_type do_decimal_point() const;

Returns: A character for use as the decimal radix separator. The required specializations return ‘.’ or L’.’.

cchar_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".

22.4.3.2 Class template numpunct_byname

namespace std {
    template <class charT>
    class numpunct_byname : public numpunct<charT> {
        // this class is specialized for char and wchar_t.
        public:
            typedef charT char_type;
            typedef basic_string<charT> string_type;
            explicit numpunct_byname(const char*, size_t refs = 0);
            explicit numpunct_byname(const string&, size_t refs = 0);
            protected:
                ~numpunct_byname();
            }
}

22.4.4 The collate category

22.4.4.1 Class template collate

namespace std {
    template <class charT>
    class collate : public locale::facet {
        public:
            typedef charT char_type;
            typedef basic_string<charT> string_type;
            explicit collate(size_t refs = 0);
    }
}

---

247) 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<char>` provides features for use in the collation (comparison) and hashing of strings. A locale member function template, `operator()`, uses the collate facet to allow a locale to act directly as the predicate argument for standard algorithms (Clause 25) and containers operating on strings. The specializations required in Table 81 (22.3.1.1.1), namely `collate<char>` and `collate<wchar_t>`, apply lexicographic ordering (25.4.8).

Each function compares a string of characters `*p` in the range `[low, high)`.

### 22.4.4.1.1 collate members

- `int compare(const charT* low1, const charT* high1, const charT* low2, const charT* high2) const;`  
  **Returns:** `do_compare(low1, high1, low2, high2)`

- `string_type transform(const charT* low, const charT* high) const;`  
  **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(const charT* low, const charT* high) const;`  
  **Returns:** `do_hash(low, high)`

### 22.4.4.1.2 collate virtual functions

- `int do_compare(const charT* low1, const charT* high1, const charT* low2, const charT* high2) const;`  
  **Returns:** 1 if the first string is greater than the second, -1 if less, zero otherwise. The specializations required in Table 81 (22.3.1.1.1), namely `collate<char>` and `collate<wchar_t>`, implement a lexicographical comparison (25.4.8).

- `string_type do_transform(const charT* low, const charT* high) const;`  
  **Returns:** A `basic_string<charT>` value that, compared lexicographically with the result of calling `transform()` on another string, yields the same result as calling `do_compare()` on the same two strings.

- `long do_hash(const charT* low, const charT* high) const;`
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 \((1.0/\text{numeric\_limits<unsigned\_long>::max()})\). — end note]

### 22.4.4.2 Class template `collate_byname`

```cpp
namespace std {
    template <class charT>
    class collate_byname : public collate<charT> {
        public:
            typedef basic_string<charT> string_type;
            explicit collate_byname(const char*, size_t refs = 0);
            explicit collate_byname(const string&, size_t refs = 0);
        protected:
            ~collate_byname();
    };
}
```

### 22.4.5 The time category

Templates `time_get<charT,InputIterator>` and `time_put<charT,OutputIterator>` provide date and time formatting and parsing. All specifications of member functions for `time_put` and `time_get` in the subclauses of 22.4.5 only apply to the specializations required in Tables 81 and 82 (22.3.1.1.1). Their members use their `ios_base&, ios_base::iostate&`, and `fill` arguments as described in (22.4), and the `ctype<>` facet, to determine formatting details.

#### 22.4.5.1 Class template `time_get`

```cpp
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:
            typedef charT char_type;
            typedef InputIterator iter_type;

            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;
        }
```
ios_base::iostate& err, tm* t, const char_type* fmt,
    const char_type* fmtend) const;

static locale::id id;

protected:
    ~time_get();
    virtual dateorder do_date_order() const;
    virtual iter_type do_get_time(iter_type s, iter_type end, ios_base&,
        ios_base::iostate& err, tm* t) const;
    virtual iter_type do_get_date(iter_type s, iter_type end, ios_base&,
        ios_base::iostate& err, tm* t) const;
    virtual iter_type do_get_weekday(iter_type s, iter_type end, ios_base&,
        ios_base::iostate& err, tm* t) const;
    virtual iter_type do_get_monthname(iter_type s, iter_type end, ios_base&,
        ios_base::iostate& err, tm* t) const;
    virtual iter_type do_get_year(iter_type s, iter_type end, ios_base&,
        ios_base::iostate& err, tm* t) const;
    virtual iter_type do_get(iter_type s, iter_type end, ios_base& f,
        ios_base::iostate& err, tm* t, char format, char modifier) const;
};

1 time_get is used to parse a character sequence, extracting components of a time or date into a struct tm record. 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.249

2 If the end iterator is reached during parsing by any of the get() member functions, the member sets ios_base::eofbit in err.

22.4.5.1.1 time_get members

[locale.time.get.members]

dateorder date_order() const;

1 Returns: do_date_order()

iter_type get_time(iter_type s, iter_type end, ios_base& str,
    ios_base::iostate& err, tm* t) const;

2 Returns: do_get_time(s, end, str, err, t)

iter_type get_date(iter_type s, iter_type end, ios_base& str,
    ios_base::iostate& err, tm* t) const;

3 Returns: do_get_date(s, end, str, err, t)

iter_type get_weekday(iter_type s, iter_type end, ios_base& str,
    ios_base::iostate& err, tm* t) const;

4 Returns: do_get_weekday(s, end, str, err, t) or do_get_monthname(s, end, str, err, t)

249 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.
iter_type get_year(iter_type s, iter_type end, ios_base& str, 
    ios_base::iostate& err, tm* t) const;

Returns: do_get_year(s, end, str, err, t)

iter_type get(iter_type s, iter_type end, ios_base& f, 
    ios_base::iostate& err, tm* t, char format, char modifier = 0) const;

Returns: do_get(s, end, f, err, t, format, modifier)

iter_type get(iter_type s, iter_type end, ios_base& f, 
    ios_base::iostate& err, tm* t, const char_type* fmt, const char_type* fmtend) const;

Requires: [fmt,fmtend) shall be a valid range.

Effects: The function starts by evaluating err = ios_base::goodbit. It then enters a loop, reading zero or more characters from s at each iteration. Unless otherwise specified below, the loop terminates when the first of the following conditions holds:

1. The expression fmt == fmtend evaluates to true.
2. The expression err == ios_base::goodbit evaluates to false.
3. The expression s == end evaluates to true, in which case the function evaluates err = ios_base::eofbit | ios_base::failbit.
4. The next element of fmt is equal to ‘%’, optionally followed by a modifier character, followed by a conversion specifier character, format, together forming a conversion specification valid for the ISO/IEC 9945 function strptime. If the number of elements in the range [fmt,fmtend) is not sufficient to unambiguously determine whether the conversion specification is complete and valid, the function evaluates err = ios_base::failbit. Otherwise, the function evaluates s = do_get(s, end, f, err, t, format, modifier), where the value of modifier is ‘0’ when the optional modifier is absent from the conversion specification. If err == ios_base::goodbit holds after the evaluation of the expression, the function increments fmt to point just past the end of the conversion specification and continues looping.
5. The expression isspace(*fmt, f.getloc()) evaluates to true, in which case the function first increments fmt until fmt == fmtend || !isspace(*fmt, f.getloc()) evaluates to true, then advances s until s == end || !isspace(*s, f.getloc()) is true, and finally resumes looping.
6. The next character read from s matches the element pointed to by fmt in a case-insensitive comparison, in which case the function evaluates ++fmt, ++s and continues looping. Otherwise, the function evaluates err = ios_base::failbit.

[Note: The function uses the ctype<charT> facet installed in f’s locale to determine valid whitespace characters. It is unspecified by what means the function performs case-insensitive comparison or whether multi-character sequences are considered while doing so. — end note]

Returns: s

22.4.5.1.2 time_get virtual functions [locale.time.get.virtuals]

dateorder do_date_order() const;

Returns: An enumeration value indicating the preferred order of components for those date formats that are composed of day, month, and year. Returns no_order if the date format specified by ‘x’ contains other variable components (e.g., Julian day, week number, week day).

iter_type do_get_time(iter_type s, iter_type end, ios_base& str, 
    ios_base::iostate& err, tm* t) const;

250) This function is intended as a convenience only, for common formats, and may return no_order in valid locales.
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.

\[
\text{iter_type do_get_date(iter_type } \ s, \ \text{iter_type end, ios_base}& \ \text{str,}
\ \\text{ios_base}& \ \text{::iostate}& \ \text{err, tm* t) const;}
\]

Effects: Reads characters starting at \( s \) until it has extracted those \( \text{struct tm} \) members and remaining format characters used by \( \text{time_put<>::put} \) to produce one of the following formats, or until it encounters an error. The format depends on the value returned by \( \text{date_order()} \) as shown in Table 92.

<table>
<thead>
<tr>
<th>( \text{date_order()} )</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>no_order</td>
<td>&quot;%m%d%y&quot;</td>
</tr>
<tr>
<td>dmy</td>
<td>&quot;%d%m%y&quot;</td>
</tr>
<tr>
<td>mdy</td>
<td>&quot;%m%d%y&quot;</td>
</tr>
<tr>
<td>ymd</td>
<td>&quot;%y%m%d&quot;</td>
</tr>
<tr>
<td>ydm</td>
<td>&quot;%y%d%m&quot;</td>
</tr>
</tbody>
</table>

Table 92 — do_get_date effects

An implementation may also accept additional implementation-defined formats.

Returns: An iterator pointing immediately beyond the last character recognized as possibly part of a valid date.

\[
\text{iter_type do_get_weekday(iter_type } \ s, \ \text{iter_type end, ios_base}& \ \text{str,}
\ \\text{ios_base}& \ \text{::iostate}& \ \text{err, tm* t) const;}
\]

Effects: Reads characters starting at \( s \) until it has extracted the (perhaps abbreviated) name of a weekday or month. If it finds an abbreviation that is followed by characters that could match a full name, it continues reading until it matches the full name or fails. It sets the appropriate \( \text{struct tm} \) member accordingly.

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid name.

\[
\text{iter_type do_get_monthname(iter_type } \ s, \ \text{iter_type end, ios_base}& \ \text{str,}
\ \\text{ios_base}& \ \text{::iostate}& \ \text{err, tm* t) const;}
\]

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 \( t->\text{tm_year} \) member accordingly.

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid year identifier.

\[
\text{iter_type do_get_year(iter_type } \ s, \ \text{iter_type end, ios_base}& \ \text{str,}
\ \\text{ios_base}& \ \text{::iostate}& \ \text{err, tm* t) const;}
\]

Effects: The function starts by evaluating \( \text{err = ios_base::goodbit} \). It then reads characters starting at \( s \) until it encounters an error, or until it has extracted and assigned those \( \text{struct tm} \) members,
and any remaining format characters, corresponding to a conversion directive appropriate for the
ISO/IEC 9945 function `strptime`, formed by concatenating '%', the modifier character, when non-
null, 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`.

13 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.

14 Remark: 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 must zero out the object before invoking the function.

15 Returns: An iterator pointing immediately beyond the last character recognized as possibly part of a
valid input sequence for the given format and modifier.

### 22.4.5.2 Class template `time_getbyname` [locale.time.getbyname]

```cpp
namespace std {
    template <class charT, class InputIterator = istreambuf_iterator<charT> >
    class time_getbyname : public time_get<charT, InputIterator> {
        public:
            typedef time_base::dateorder dateorder;
            typedef InputIterator iter_type;
            
            explicit time_getbyname(const char*, size_t refs = 0);
            explicit time_getbyname(const string&, size_t refs = 0);
            
            protected:
            -time_getbyname();
        }
} // namespace std
```

### 22.4.5.3 Class template `time_put` [locale.time.put]

```cpp
namespace std {
    template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
    class time_put : public locale::facet {
        public:
            typedef charT char_type;
            typedef OutputIterator iter_type;
            
            explicit time_put(size_t refs = 0);
            
            protected:
            static locale::id id;
            
            -time_put();
        
        virtual iter_type do_put(iter_type s, ios_base& f, char_type fill, const tm* tmb,
            const charT* pattern, const charT* pat_end) const;
        
        virtual iter_type do_put(iter_type s, ios_base& f, char_type fill,
            const tm* tmb, char format, char modifier = 0) const;
    }
} // namespace std
```

§ 22.4.5.3 730
22.4.5.3.1 time_put members

```cpp
iter_type put(iter_type s, ios_base& str, char_type fill, const tm* t,
        const charT* pattern, const charT* pat_end) const;
iter_type put(iter_type s, ios_base& str, char_type fill, const tm* t,
        char format, char modifier = 0) const;
```

Effects: The first form steps through the sequence from `pattern` to `pat_end`, identifying characters that are part of a format sequence. Each character that is not part of a format sequence is written to `s` immediately, and each format sequence, as it is identified, results in a call to `do_put`; thus, format elements and other characters are interleaved in the output in the order in which they appear in the pattern. Format sequences are identified by converting each character `c` to a `char` value as if by `ct.narrow(c, 0)`, where `ct` is a reference to `ctype<charT>` obtained from `str.getloc()`. The first character of each sequence is equal to `'%'`, followed by an optional modifier character `mod` and a format specifier character `spec` as defined for the function `strftime`. If no modifier character is present, `mod` is zero. For each valid format sequence identified, calls `do_put(s, str, fill, t, spec, mod)`.

The second form calls `do_put(s, str, fill, t, format, modifier)`.

[Note: The `fill` argument may be used in the implementation-defined formats or by derivations. A space character is a reasonable default for this argument. — end note]

Returns: An iterator pointing immediately after the last character produced.

22.4.5.3.2 time_put virtual functions

```cpp
iter_type do_put(iter_type s, ios_base& str, char_type fill, const tm* t,
        char format, char modifier) const;
```

Effects: Formats the contents of the parameter `t` into characters placed on the output sequence `s`. Formatting is controlled by the parameters `format` and `modifier`, interpreted identically as the format specifiers in the string argument to the standard library function `strftime()`\(^{252}\), except that the sequence of characters produced for those specifiers that are described as depending on the C locale are instead implementation-defined.\(^{253}\)

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]

22.4.5.4 Class template time_put_byname

```cpp
namespace std {
    template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
    class time_put_byname : public time_put<charT, OutputIterator>
    {
        public:
            typedef charT char_type;
            typedef OutputIterator iter_type;

            explicit time_put_byname(const char*, size_t refs = 0);
    }
}
```

---

\(^{251}\) Although the C programming language defines no modifiers, most vendors do.

\(^{252}\) Interpretation of the `modifier` argument is implementation-defined, but should follow POSIX conventions.

\(^{253}\) Implementations are encouraged to refer to other standards such as POSIX for these definitions.
22.4.6 The monetary category [category.monetary]

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 22.4.6 only apply to the specializations required in Tables 81 and 82 (22.3.1.1.1). Their members use their `ios_base&`, `ios_base::iostate&`, and `fill` arguments as described in (22.4), and the `moneypunct<>` and `ctype<>` facets, to determine formatting details.

22.4.6.1 Class template `money_get` [locale.money.get]

```cpp
namespace std {
    template <class charT,
             class InputIterator = istreambuf_iterator<charT> >
    class money_get : public locale::facet {
    public:
        typedef charT char_type;
        typedef InputIterator iter_type;
        typedef basic_string<charT> string_type;
        explicit money_get(size_t refs = 0);
        iter_type get(iter_type s, iter_type end, bool intl,
                      ios_base& f, ios_base::iostate& err,
                      long double& units) const;
        iter_type get(iter_type s, iter_type end, bool intl,
                      ios_base& f, ios_base::iostate& err,
                      string_type& digits) const;
        static locale::id id;
    protected:
        ~money_get();
        virtual iter_type do_get(iter_type, iter_type, bool, ios_base&,
                                 ios_base::iostate& err, long double& units) const;
        virtual iter_type do_get(iter_type, iter_type, bool, ios_base&,
                                 ios_base::iostate& err, string_type& digits) const;
    };
}
```

22.4.6.1.1 `money_get` members [locale.money.get.members]

```cpp
iter_type get(iter_type s, iter_type end, bool intl,
              ios_base& f, ios_base::iostate& err,
              long double& quant) const;
iter_type get(s, iter_type end, bool intl, ios_base&f,
              ios_base::iostate& err, string_type& quant) const;
```

Returns: `do_get(s, end, intl, f, err, quant)`

22.4.6.1.2 `money_get` virtual functions [locale.money.get.virtuals]

§ 22.4.6.1.2
Effects: Reads characters from \( s \) to parse and construct a monetary value according to the format specified by a `moneypunct<charT,Intl>` facet reference \( \text{mp} \) and the character mapping specified by a `ctype<charT>` facet reference \( \text{ct} \) obtained from the locale returned by \( \text{str}.\text{getloc}() \), and \( \text{str}.\text{flags}() \). If a valid sequence is recognized, does not change \( \text{err} \); otherwise, sets \( \text{err} \) to \( (\text{err} | \text{str}.\text{failbit}) \), or \( (\text{err} | \text{str}.\text{failbit} | \text{str}.\text{eofbit}) \) if no more characters are available, and does not change \( \text{units} \) or \( \text{digits} \). Uses the pattern returned by \( \text{mp}.\text{neg_format}() \) to parse all values. The result is returned as an integral value stored in \( \text{units} \) or as a sequence of digits possibly preceded by a minus sign (as produced by \( \text{ct}.\text{widen(c)} \) where \( c \) is '−' or in the range from '0' through '9', inclusive) stored in \( \text{digits} \). [Example: The sequence $1,056.23$ in a common United States locale would yield, for \( \text{units} \), 105623, or, for \( \text{digits} \), "105623". — end example] If \( \text{mp}.\text{grouping()} \) 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 \( (\text{str}.\text{flags()} & \text{str}.\text{showbase}) \) 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 \( \text{pos} \) returned by \( \text{mp}.\text{positive_sign}() \) or the string \( \text{neg} \) returned by \( \text{mp}.\text{negative_sign}() \) is recognized in the position indicated by \( \text{sign} \) in the format pattern, it is consumed and any remaining characters in the string are required after all the other format components. [Example: If \( \text{showbase} \) is off, then for a \( \text{neg} \) value of "()" and a currency symbol of "L", in "(100 L)" the "L" is consumed; but if \( \text{neg} \) is "−", the "L" in "−100 L" is not consumed. — end example] If \( \text{pos} \) or \( \text{neg} \) 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 \( \text{pos} \) or \( \text{neg} \), and the result is given the corresponding sign. If the first character of \( \text{pos} \) is equal to the first character of \( \text{neg} \), or if both strings are empty, the result is given a positive sign.

Digits in the numeric monetary component are extracted and placed in \( \text{digits} \), or into a character buffer \( \text{buf1} \) for conversion to produce a value for \( \text{units} \), in the order in which they appear, preceded by a minus sign if and only if the result is negative. The value \( \text{units} \) 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 \( \text{buf1} \), \( \text{buf2} \) is a character buffer, and the values \( \text{src} \) and \( \text{atoms} \) are defined as if by

```c
static const char src[] = "0123456789-";
charT atoms[sizeof(src)];
ct.widen(src, src + sizeof(src) - 1, atoms);
```

\( ^{254} \) The semantics here are different from \( \text{ct}.\text{narrow} \).
Returns: An iterator pointing immediately beyond the last character recognized as part of a valid monetary quantity.

22.4.6.2 Class template money_put

```cpp
namespace std {
    template <class charT,
             class OutputIterator = ostreambuf_iterator<charT> >
    class money_put : public locale::facet {
public:
    typedef charT char_type;
    typedef OutputIterator iter_type;
    typedef basic_string<charT> string_type;

    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;
    }
}
```

22.4.6.2.1 money_put members

```cpp
iter_type put(iter_type s, bool intl, ios_base& f,
              char_type fill, long double quant) const;
iter_type put(iter_type s, bool intl, ios_base& f,
              char_type fill, const string_type& quant) const;
```

Returns: do_put(s, intl, f, loc, quant)

22.4.6.2.2 money_put virtual functions

```cpp
iter_type do_put(iter_type s, bool intl, ios_base& str,
                 char_type fill, long double units) const;
iter_type do_put(iter_type s, bool intl, ios_base& str,
                 char_type fill, const string_type& digits) const;
```

Effects: Writes characters to s according to the format specified by a moneypunct< charT , Intl > facet reference mp and the character mapping specified by a ctype< charT > facet reference ct obtained from the locale returned by str.getloc(), and str.flags(). The argument units is transformed into a sequence of wide characters as if by

```cpp
cf.widen(buf1, buf1 + sprintf(buf1, "%.0Lf", units), buf2)
```

for character buffers buf1 and buf2. If the first character in digits or buf2 is equal to cf.widen(‘-‘), then the pattern used for formatting is the result of mp.neg_format(); otherwise the pattern is the result of mp.pos_format(). Digit characters are written, interspersed with any thousands separators
and decimal point specified by the format, in the order they appear (after the optional leading minus sign) in digits or buf2. In digits, only the optional leading minus sign and the immediately subsequent digit characters (as classified according to ct) are used; any trailing characters (including digits appearing after a non-digit character) are ignored. Calls str.width(0).

Remarks: The currency symbol is generated if and only if \((str.flags() \& str.showbase)\) is nonzero. If the number of characters generated for the specified format is less than the value returned by str.width() on entry to the function, then copies of fill are inserted as necessary to pad to the specified width. For the value \(af\) equal to \((str.flags() \& str.adjustfield)\), if \((af == str.internal)\) is true, the fill characters are placed where none or space appears in the formatting pattern; otherwise if \((af == str.left)\) is true, they are placed after the other characters; otherwise, they are placed before the other characters. [Note: It is possible, with some combinations of format patterns and flag values, to produce output that cannot be parsed using num_get<>::get. —end note]

Returns: An iterator pointing immediately after the last character produced.

### 22.4.6.3 Class template moneypunct

```cpp
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:
        typedef charT char_type;
        typedef basic_string<charT> string_type;

        explicit moneypunct(size_t refs = 0);

        charT decimal_point() const;
        charT thousands_sep() const;
        string grouping() const;
        string_type curr_symbol() const;
        string_type positive_sign() const;
        string_type negative_sign() const;
        int frac_digits() const;
        pattern pos_format() const;
        pattern neg_format() const;

        static locale::id id;
        static const bool intl = International;

        protected:
            ~moneypunct();
            virtual charT do_decimal_point() const;
            virtual charT do_thousands_sep() const;
            virtual string do_grouping() const;
            virtual string_type do_curr_symbol() const;
            virtual string_type do_positive_sign() const;
            virtual string_type do_negative_sign() const;
            virtual int do_frac_digits() const;
            virtual pattern do_pos_format() const;

    };
}
```

§ 22.4.6.3
The `moneypunct<>` facet defines monetary formatting parameters used by `money_get<>` and `money_put<>`. A monetary format is a sequence of four components, specified by a `pattern` value `p`, such that the `part` value `static_cast<part>(p.field[i])` determines the `i`th component of the format. In the `field` member of a `pattern` object, each value `symbol`, `sign`, `value`, and either `space` or `none` appears exactly once. The value `none`, if present, is not first; the value `space`, if present, is neither first nor last.

Where `none` or `space` appears, white space is permitted in the format, except where `none` appears at the end, in which case no white space is permitted. The value `space` indicates that at least one space is required at that position. Where `symbol` appears, the sequence of characters returned by `curr_symbol()` is permitted, and can be required. Where `sign` appears, the first (if any) of the sequence of characters returned by `positive_sign()` or `negative_sign()` (respectively as the monetary value is non-negative or negative) is required. Any remaining characters of the sign sequence are required after all other format components. Where `value` appears, the absolute numeric monetary value is required.

The format of the numeric monetary value is a decimal number:

```
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:

```
units ::= digits [ thousands-sep units ]
digits ::= adigit [ digits ]
```

In the syntax specification, the symbol `adigit` is any of the values `ct.widen(c)` for `c` in the range '0' through '9', inclusive, and `ct` is a reference of type `const ctype<charT>&` obtained as described in the definitions of `money_get<>` and `money_put<>`. The symbol `thousands-sep` is the character returned by `thousands_sep()`. The space character used is the value `ct.widen(' ')`. White space characters are those characters `c` for which `ci.is(space,c)` returns `true`. The number of digits required after the decimal point (if any) is exactly the value returned by `frac_digits()`.

The placement of thousands-separator characters (if any) is determined by the value returned by `grouping()`, defined identically as the member `numpunct<>::do_grouping()`.

Each of these functions `F` returns the result of calling the corresponding virtual member function `do_F()`.
charT do_decimal_point() const;

Returns: The radix separator to use in case do_frac_digits() is greater than zero.\textsuperscript{256}

charT do_thousands_sep() const;

Returns: The digit group separator to use in case do_grouping() specifies a digit grouping pattern.\textsuperscript{257}

string do_grouping() const;

Returns: A pattern defined identically as, but not necessarily equal to, the result of numpunct<charT>::
do_grouping().\textsuperscript{258}

string_type do_curr_symbol() const;

Returns: A string to use as the currency identifier symbol.\textsuperscript{259}

string_type do_positive_sign() const;
string_type do_negative_sign() const;

Returns: do_positive_sign() returns the string to use to indicate a positive monetary value;\textsuperscript{260}
do_negative_sign() returns the string to use to indicate a negative value.

int do_frac_digits() const;

Returns: The number of digits after the decimal radix separator, if any.\textsuperscript{261}

pattern do_pos_format() const;
pattern do_neg_format() const;

Returns: The specializations required in Table 82 (22.3.1.1.1), namely moneypunct<char>, moneypunct<
wchar_t>, moneypunct<char,true>, and moneypunct<wchar_t,true>, return an object of type pattern
initialized to \{ symbol, sign, none, value \}.\textsuperscript{262}

\section*{22.4.6.4 Class template moneypunct_byname}

\section*{22.4.7 The message retrieval category}

\begin{verbatim}
namespace std {
    template <class charT, bool Intl = false>
    class moneypunct_byname : public moneypunct<charT, Intl> {
        public:
            typedef money_base::pattern pattern;
            typedef basic_string<charT> string_type;

            explicit moneypunct_byname(const char*, size_t refs = 0);
            explicit moneypunct_byname(const string&, size_t refs = 0);

            protected:
                ~moneypunct_byname();
            
            
    }
}
\end{verbatim}

Class messages<charT> implements retrieval of strings from message catalogs.

\textsuperscript{256} In common U.S. locales this is ".".
\textsuperscript{257} In common U.S. locales this is ",".
\textsuperscript{258} To specify grouping by 3s, the value is "\003" not "3".
\textsuperscript{259} For international specializations (second template parameter true) this is typically four characters long, usually three
letters and a space.
\textsuperscript{260} This is usually the empty string.
\textsuperscript{261} In common U.S. locales, this is 2.
\textsuperscript{262} Note that the international symbol returned by do_curr_sym() usually contains a space, itself; for example, "USD ".

\S 22.4.7
22.4.7.1 Class template messages

namespace std {
    class messages_base {
        public:
            typedef unspecified signed integer type catalog;
    };

    template <class charT>
    class messages : public locale::facet, public messages_base {
        public:
            typedef charT char_type;
            typedef basic_string<charT> string_type;
            explicit messages(size_t refs = 0);
            catalog open(const basic_string<char>& fn, const locale&) const;
            string_type get(catalog c, int set, intmsgid, const string_type& dfault) const;
            void close(catalog c) const;
            static locale::id id;
        protected:
            messages();
            virtual catalog do_open(const basic_string<char>&, const locale&) const;
            virtual string_type do_get(catalog, int set, intmsgid, const string_type& dfault) const;
            virtual void do_close(catalog) const;
    };
}

1 Values of type messages_base::catalog usable as arguments to members get and close can be obtained only by calling member open.

22.4.7.1.1 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, intmsgid, const string_type& dfault) const;
2       Returns: do_get(cat, set,msgid, dfault).

void close(catalog cat) const;
3       Effects: Calls do_close(cat).

22.4.7.1.2 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.
*Remarks:* The locale argument `loc` is used for character set code conversion when retrieving messages, if needed.

```cpp
template <class charT>
class messages_byname : public messages<charT> {
public:
    typedef messages_base::catalog catalog;
    typedef basic_string<charT> string_type;

    explicit messages_byname(const char*, size_t refs = 0);
    explicit messages_byname(const string&, size_t refs = 0);

private:
    messages_byname();
};
```

### 22.4.8 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`.

**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
    // imbue it on all the std streams

    cin.imbue(locale());
    cout.imbue(locale());
    cerr.imbue(locale());
    wcin.imbue(locale());
    wcout.imbue(locale());
    wcerr.imbue(locale());

    return MyObject(argc, argv).doit();
}
```
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;
    return (cin.fail() != 0);
}
```

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.
A locale object may be extended with a new facet simply by constructing it with an instance of a class derived from \texttt{locale::facet}. The only member a C++ program must define is the static member \texttt{id}, which identifies your class interface as a new facet.

\textbf{Example:} Classifying Japanese characters:

\begin{verbatim}
#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;
            JCtype() {}
        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;
        typedef ctype<wchar_t> wctype;
        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;
        return 0;
    }
\end{verbatim}

The new facet is used exactly like the built-in facets. — \textit{end example}]

\textbf{Example:} Replacing an existing facet is even easier. The code does not define a member \texttt{id} because it is reusing the \texttt{numpunct<charT>} facet interface:

\begin{verbatim}
// file: my_bool.C
#include <iostream>
#include <locale>
#include <string>
namespace My {
    using namespace std;
    typedef numpunct_byname<char> cnumpunct;
\end{verbatim}

\section*{22.4.8}
22.5 Standard code conversion facets

The header `<codecvt>` provides code conversion facets for various character encodings.

Header `<codecvt>` synopsis

```cpp
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:
```
For each of the three code conversion facets codecvt_utf8, codecvt_utf16, and codecvt_utf8_utf16:

1. **Elem** is the wide-character type, such as wchar_t, char16_t, or char32_t.

2. **Maxcode** is the largest wide-character code that the facet will read or write without reporting a conversion error.

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.

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.

5. If (Mode & little_endian), the facet shall generate a multibyte sequence in little-endian order, as opposed to the default big-endian order.

For the facet codecvt_utf8:

1. The facet shall convert between UTF-8 multibyte sequences and UCS2 or UCS4 (depending on the size of **Elem**) within the program.

2. Endianness shall not affect how multibyte sequences are read or written.

3. The multibyte sequences may be written as either a text or a binary file.

For the facet codecvt_utf16:

1. The facet shall convert between UTF-16 multibyte sequences and UCS2 or UCS4 (depending on the size of **Elem**) within the program.

2. Multibyte sequences shall be read or written according to the **Mode** flag, as set out above.

3. The multibyte sequences may be written only as a binary file. Attempting to write to a text file produces undefined behavior.

For the facet codecvt_utf8_utf16:

1. The facet shall convert between UTF-8 multibyte sequences and UTF-16 (one or two 16-bit codes) within the program.

2. Endianness shall not affect how multibyte sequences are read or written.

3. The multibyte sequences may be written as either a text or a binary file.

See also: ISO/IEC 10646-1:1993.

### 22.6 C library locales

1. Table 93 describes header `<cllocale>`.

2. The contents are the same as the Standard C library header `<locale.h>`.

3. Calls to the function `setlocale` may introduce a data race (17.6.5.9) with other calls to `setlocale` or with calls to the functions listed in Table 94.

See also: ISO C Clause 7.4.

§ 22.6
Table 93 — Header `<locale>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td>LC_ALL</td>
</tr>
<tr>
<td></td>
<td>LC_COLLATE</td>
</tr>
<tr>
<td></td>
<td>LC_CTYPE</td>
</tr>
<tr>
<td></td>
<td>LC_MONETARY</td>
</tr>
<tr>
<td></td>
<td>LC_NUMERIC</td>
</tr>
<tr>
<td></td>
<td>LC_TIME</td>
</tr>
<tr>
<td></td>
<td>NULL</td>
</tr>
<tr>
<td>Struct:</td>
<td>lconv</td>
</tr>
<tr>
<td>Functions:</td>
<td>localeconv</td>
</tr>
<tr>
<td></td>
<td>setlocale</td>
</tr>
</tbody>
</table>

Table 94 — Potential `setlocale` data races

<table>
<thead>
<tr>
<th>fprintf</th>
<th>isprint</th>
<th>iswdigit</th>
<th>localeconv</th>
<th>tolower</th>
</tr>
</thead>
<tbody>
<tr>
<td>fscanf</td>
<td>ispunct</td>
<td>iswgraph</td>
<td>mblen</td>
<td>toupper</td>
</tr>
<tr>
<td>isalnum</td>
<td>isspace</td>
<td>iswlower</td>
<td>mbstowcs</td>
<td>tolower</td>
</tr>
<tr>
<td>isalpha</td>
<td>isupper</td>
<td>iswprint</td>
<td>mbtowc</td>
<td>toupper</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>iswcntrol</td>
<td>iswxdigit</td>
<td>strtod</td>
<td>wcxfm</td>
</tr>
<tr>
<td>islower</td>
<td>iswctype</td>
<td>isxdigit</td>
<td>strxfm</td>
<td>wcxtomb</td>
</tr>
</tbody>
</table>
23 Containers library

23.1 General

1 This Clause describes components that C++ programs may use to organize collections of information.

2 The following subclauses describe container requirements, and components for sequence containers and associative containers, as summarized in Table 95.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.2</td>
<td></td>
</tr>
<tr>
<td>23.3</td>
<td>&lt;array&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;deque&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;forward_list&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;list&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;vector&gt;</td>
</tr>
<tr>
<td>23.4</td>
<td>&lt;map&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;set&gt;</td>
</tr>
<tr>
<td>23.5</td>
<td>&lt;unordered_map&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;unordered_set&gt;</td>
</tr>
<tr>
<td>23.6</td>
<td>&lt;queue&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;stack&gt;</td>
</tr>
</tbody>
</table>

23.2 Container requirements

23.2.1 General container requirements

1 Containers are objects that store other objects. They control allocation and deallocation of these objects through constructors, destructors, insert and erase operations.

2 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]

3 For the components affected by this subclause that declare an allocator_type, objects stored in these components shall be constructed using the allocator_traits<allocator_type>::construct function and destroyed using the allocator_traits<allocator_type>::destroy function (20.7.8.2). 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]

4 In Tables 96, 97, and 98 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>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::value_type</td>
<td>T</td>
<td></td>
<td>Requires: T is Erasable from X (see 23.2.1, below)</td>
<td>compile time</td>
</tr>
<tr>
<td>X::reference</td>
<td>T&amp;</td>
<td></td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>X::const_reference</td>
<td>const T&amp;</td>
<td></td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>X::iterator</td>
<td>iterator type whose value type is T</td>
<td>any iterator category that meets the forward iterator requirements. convertible to X::const_iterator.</td>
<td>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>
</tbody>
</table>

| u;                  | |                       | post: u.empty() | constant |
| X()                 | |                       | post: X().empty() | constant |
| X(a)                | |                       | Requires: T is CopyInsertable into X (see below). post: a == X(a). | linear |
| X u(a)              | |                       | Requires: T is CopyInsertable into X (see below). post: u == a | linear |
| X u = a;            | |                       |                | linear |
| X u(rv)             | |                       | post: u shall be equal to the value that rv had before this construction | (Note B) |
| X u = rv            | |                       |                | linear |
| a = rv              | X&          | All existing elements of a are either move assigned to or destroyed | a shall be equal to the value that rv had before this assignment | linear |
Table 96 — 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>(&amp;a)-&gt;-X()</td>
<td>void</td>
<td>note: 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;</td>
<td></td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>const_iterator for constant a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a.end()</td>
<td>iterator;</td>
<td></td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>const_iterator for constant a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a.cbegin()</td>
<td>const_iterator</td>
<td>const_cast&lt;X const&amp;&gt;(a).begin();</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>a.cend()</td>
<td>const_iterator</td>
<td>const_cast&lt;X const&amp;&gt;(a).end();</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a == b</td>
<td>convertible to bool</td>
<td>== is an equivalence relation. Requires: T is EqualityComparable</td>
<td>Constant if a.size() != b.size(), linear otherwise</td>
<td></td>
</tr>
<tr>
<td>a != b</td>
<td>convertible to bool</td>
<td>Equivalent to: !(a == b)</td>
<td>linear</td>
<td></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>post: r == a.</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>a.size()</td>
<td>size_type</td>
<td>distance(a.begin(), a.end())</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>a.max_size()</td>
<td>size_type</td>
<td>distance(begin(), end()) for the largest possible container</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>a.empty()</td>
<td>convertible to bool</td>
<td>a.begin() == a.end()</td>
<td>constant</td>
<td></td>
</tr>
</tbody>
</table>

Notes: the algorithm equal() is defined in Clause 25. Those entries marked “(Note A)” or “(Note B)” have linear complexity for array and have constant complexity for all other standard containers.

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

§ 23.2.1
\[
i == j
\]
\[
i != j
\]
\[
i <= j
\]
\[
i >= j
\]
\[
i > j
\]
\[
i - j
\]

where \(i\) and \(j\) denote objects of a container’s \texttt{iterator} type, either or both may be replaced by an object of the container’s \texttt{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 17.6.3.5). Copy constructors for these container types obtain an allocator by calling \texttt{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 \texttt{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 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 \texttt{swap()}. Allocator replacement is performed by copy assignment, move assignment, or swapping of the allocator only if \texttt{allocator_traits<allocator_type>::propagate_on_container_copy_assignment::value}, \texttt{allocator_traits<allocator_type>::propagate_on_container_move_assignment::value}, or \texttt{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 \texttt{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 \(\texttt{a.swap(b)}\), for containers \(a\) and \(b\) of a standard container type other than \texttt{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 \texttt{Compare}, \texttt{Pred}, or \texttt{Hash} types belonging to \(a\) and \(b\) shall be swappable and shall be exchanged by calling \texttt{swap} as described in 17.6.3.2. If \texttt{allocator_traits<allocator_type>::propagate_on_container_swap::value} is true, then lvalues of type \texttt{allocator_type} shall be swappable and the allocators of \(a\) and \(b\) shall also be exchanged by calling \texttt{swap} as described in 17.6.3.2. 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 (24.2), the container is called \textit{reversible} and satisfies the additional requirements in Table 97.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\texttt{X::reverse_iterator})</td>
<td>iterator type whose value type is (T)</td>
<td>\texttt{reverse_iterator&lt;iterator&gt;}</td>
<td>compile time</td>
</tr>
<tr>
<td>(\texttt{X::const_reverse_iterator})</td>
<td>constant iterator type whose value type is (T)</td>
<td>\texttt{reverse_iterator&lt;const_reverse_iterator&gt;}</td>
<td>compile time</td>
</tr>
</tbody>
</table>
Table 97 — Reversible container requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.rbegin()</td>
<td>reverse_iterator; const_reverse_iterator for constant a</td>
<td>reverse_iterator(a.rbegin())</td>
<td>constant</td>
</tr>
<tr>
<td>a.rend()</td>
<td>reverse_iterator; const_reverse_iterator for constant a</td>
<td>reverse_iterator(a.rend())</td>
<td>constant</td>
</tr>
<tr>
<td>a.crbegin()</td>
<td>const_reverse_iterator</td>
<td>const_cast&lt;X const&amp;&gt;(a).rbegin()</td>
<td>constant</td>
</tr>
<tr>
<td>a.crend()</td>
<td>const_reverse_iterator</td>
<td>const_cast&lt;X const&amp;&gt;(a).rend()</td>
<td>constant</td>
</tr>
</tbody>
</table>

11 Unless otherwise specified (see 23.2.4.1, 23.2.5.1, 23.3.3.4, and 23.3.6.5) all container types defined in this Clause meet the following additional requirements:

(11.1) — if an exception is thrown by an `insert()` or `emplace()` function while inserting a single element, that function has no effects.

(11.2) — if an exception is thrown by a `push_back()`, `push_front()`, `emplace_back()`, or `emplace_front()` function, that function has no effects.

(11.3) — no `erase()`, `clear()`, `pop_back()` or `pop_front()` function throws an exception.

(11.4) — no copy constructor or assignment operator of a returned iterator throws an exception.

(11.5) — no `swap()` function throws an exception.

(11.6) — no `swap()` function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped. [Note: The `end()` iterator does not refer to any element, so it may be invalidated. — end note]

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 (24.2.7) and whose member types `iterator` and `const_iterator` are contiguous iterators (24.2.1).

14 Table 98 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 98 unless otherwise stated.

§ 23.2.1

749
Table 98 — Optional container operations

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt; b</td>
<td>convertible to bool</td>
<td>lexicographical_compare(a.begin(), a.end(), b.begin(), b.end())</td>
<td>pre: &lt; is defined for values of T. &lt; is a total ordering relationship.</td>
<td>linear</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>convertible to bool</td>
<td>b &lt; a</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>convertible to bool</td>
<td>!(a &gt; b)</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>a &gt;= b</td>
<td>convertible to bool</td>
<td>!(a &lt; b)</td>
<td>linear</td>
<td></td>
</tr>
</tbody>
</table>

Note: the algorithm lexicographical_compare() is defined in Clause 25.

All of the containers defined in this Clause and in (21.4) except array meet the additional requirements of an allocator-aware container, as described in Table 99.

Given a container type X having an allocator_type identical to A and a value_type identical to 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 std::allocator<T> — no allocator object needs to be created and user specializations of std::allocator<T> are not instantiated:

(15.1) — T is DefaultInsertable into X means that the following expression is well-formed:

\[
\text{allocator_traits<A>::construct}(m, p)
\]

(15.2) — An element of X is default-inserted if it is initialized by evaluation of the expression

\[
\text{allocator_traits<A>::construct}(m, p)
\]

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:

\[
\text{allocator_traits<A>::construct}(m, p, rv)
\]

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:

\[
\text{allocator_traits<A>::construct}(m, p, v)
\]

and its evaluation causes the following postcondition to hold: The value of v is unchanged and is equivalent to \*p.

(15.5) — T is EmplaceConstructible into X from args, for zero or more arguments args, means that the following expression is well-formed:
allocators\texttraits\textless A\textgreater \::\textbf{construct}(m, p, \text{args})

(15.6)  
— T is Erasable from X means that the following expression is well-formed:

\textbf{allocators\texttraits\textless A\textgreater \::\textbf{destroy}(m, p)

[Note: A container calls \textbf{allocators\texttraits\textless A\textgreater \::\textbf{construct}(m, p, \text{args})} to construct an element at \( p \) using \text{args}. The default \textbf{construct} in \text{std\textcolon\textaposternot\textfamilydefault\selectfont{:}\textbf{allocator}} will call \textbf{\texttt{::new}}((\texttt{void}*)p) \( \text{T} \)(\text{args}), but specialized allocators may choose a different definition. — end note]

In Table 99, X denotes an allocator-aware container class with a \textbf{value_type} of \( \text{T} \) 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, \( \text{rv} \) denotes a non-const rvalue of type X, and m is a value of type A.

Table 99 — Allocator-aware container requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{allocator\textunderscore type} A</td>
<td></td>
<td>\textit{Requires:} allocator\textunderscore type::value_type is the same as X::value_type.</td>
<td>compile time</td>
</tr>
<tr>
<td>get\textunderscore allocator()</td>
<td>A</td>
<td>\textit{Requires:} A is DefaultConstructible. post: u.empty() returns true, u.get_allocator() == A()</td>
<td>constant</td>
</tr>
<tr>
<td>X() u;</td>
<td>X</td>
<td>post: u.empty() returns true, u.get_allocator() == A()</td>
<td>constant</td>
</tr>
<tr>
<td>X(m) u(m);</td>
<td>X</td>
<td>post: u.empty() returns true, u.get_allocator() == m</td>
<td>constant</td>
</tr>
<tr>
<td>X(t, m) X u(t, m);</td>
<td>X</td>
<td>\textit{Requires:} ( \text{T} ) is CopyInsertable into X. post: u == t, u.get_allocator() == m</td>
<td>linear</td>
</tr>
<tr>
<td>X(rv) X u(rv)</td>
<td>X</td>
<td>\textit{Requires:} move construction of A shall not exit via an exception. post: u shall have the same elements as ( \text{rv} ) had before this construction; the value of u.get_allocator() shall be the same as the value of rv.get_allocator() before this construction.</td>
<td>constant</td>
</tr>
<tr>
<td>X(rv, m) X u(rv, m);</td>
<td>X</td>
<td>\textit{Requires:} ( \text{T} ) is MoveInsertable into X. post: u shall have the same elements, or copies of the elements, that ( \text{rv} ) had before this construction, u.get_allocator() == m</td>
<td>constant if m == rv.get\textunderscore allocator(), otherwise linear</td>
</tr>
</tbody>
</table>
Table 99 — Allocator-aware container requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = t</td>
<td>X&amp;</td>
<td>Requires: T is CopyInsertable into X and CopyAssignable. post: a == t</td>
<td>linear</td>
</tr>
<tr>
<td>a = rv</td>
<td>X&amp;</td>
<td>Requires: If allocator_traits&lt;allocator_type&gt;::propagate_on_container_=move_assignment::value is false, T is MoveInsertable into X and MoveAssignable. All existing elements of a are either move assigned to or destroyed. post: a shall be equal to the value that rv had before this assignment.</td>
<td>linear</td>
</tr>
<tr>
<td>a.swap(b)</td>
<td>void</td>
<td>exchanges the contents of a and b</td>
<td>constant</td>
</tr>
</tbody>
</table>

23.2.2 Container data races

1 For purposes of avoiding data races (17.6.5.9), 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 (17.6.5.9), 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]

23.2.3 Sequence containers

1 A sequence container organizes a finite set of objects, all of the same type, into a strictly linear arrangement. The library provides four basic kinds of sequence containers: vector, forward_list, list, and deque. In addition, array is provided as a sequence container which provides limited sequence operations because it has a fixed number of elements. The library also provides container adaptors that make it easy to construct abstract data types, such as stacks or queues, out of the basic sequence container kinds (or out of other kinds of sequence containers that the user might define).

2 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 100 and 101, X denotes a sequence container class, a denotes a value of X containing elements of type T, A denotes X::allocator_type if the qualified-id X::allocator_type is valid and denotes a type (14.8.2) and std::allocator<T> if it doesn’t, i and j denote iterators satisfying input iterator requirements and

§ 23.2.3

752
refer to elements implicitly convertible to `value_type`, 
[i, j) denotes a valid range, il designates an 
object of type `initializer_list<value_type>`, n denotes a value of `X::size_type`, p denotes a valid 
const iterator to a, q designates a valid dereferenceable const iterator to a. [q1, q2) denotes a valid range of 
const iterators in a, t denotes an lvalue or a const rvalue of `X::value_type`, and rv denotes a non-const 
rvalue of `X::value_type`. Args denotes a template parameter pack; args denotes a function parameter 
pack with the pattern `Args&&`.

4 The complexities of the expressions are sequence dependent.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
</tr>
</thead>
</table>
| `X(n, t)` | `X a(n, t)` | Requires: T shall be `CopyInsertable` into X.  
post: `distance(begin(), end()) == n`  
Constructs a sequence container with n copies of t |
| `X(i, j)` | `X a(i, j)` | Requires: T shall be `EmplaceConstructible` 
into X from *i. For vector, if the iterator 
does not meet the forward iterator 
requirements (24.2.5), T shall also be 
`MoveInsertable` into X. Each iterator in the 
range [i,j) shall be dereferenced exactly once.  
post: `distance(begin(), end()) == distance(i, j)`  
Constructs a sequence container equal to the 
range [i, j) |
| `X(il);` | X& | Equivalent to `X(il.begin(), il.end())`  
| `a = il;` | X& | 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. |
| `a.emplace(p, args);` | iterator | 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 
`std::forward<Args>(args)...` before p. |
| `a.insert(p, t)` | iterator | Requires: T shall be `CopyInsertable` into X.  
For vector and deque, T shall also be  
`CopyAssignable`.  
Effects: Inserts a copy of t before p. |
| `a.insert(p, rv)` | iterator | Requires: T shall be `MoveInsertable` into X.  
For vector and deque, T shall also be  
`MoveAssignable`.  
Effects: Inserts a copy of rv before p. |
### Table 100 — Sequence container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(\text{.}\text{insert}(p, n, t))</td>
<td>iterator</td>
<td>Requires: (T) shall be \texttt{CopyInsertable} into (X) and \texttt{CopyAssignable}. Inserts (n) copies of (t) before (p).</td>
</tr>
<tr>
<td>a(\text{.}\text{insert}(p, i, j))</td>
<td>iterator</td>
<td>Requires: (T) shall be \texttt{EmplaceConstructible} into (X) from (*i). For \texttt{vector} and \texttt{deque}, (T) shall also be \texttt{MoveInsertable} into (X), \texttt{MoveConstructible}, \texttt{MoveAssignable}, and swappable (17.6.3.2). Each iterator in the range ([i, j)) shall be dereferenced exactly once. (\text{pre}: i) and (j) are not iterators into (a). Inserts copies of elements in ([i, j)) before (p).</td>
</tr>
<tr>
<td>a(\text{.}\text{insert}(p, \text{il}));</td>
<td>iterator</td>
<td>Requires: (T) shall be \texttt{CopyInsertable} into (X) and \texttt{CopyAssignable}. Inserts (n) copies of (t) before (p).</td>
</tr>
<tr>
<td>a(\text{.}\text{erase}(q))</td>
<td>iterator</td>
<td>Requires: For \texttt{vector} and \texttt{deque}, (T) shall be \texttt{MoveAssignable}. Effects: Erases the element pointed to by (q).</td>
</tr>
<tr>
<td>a(\text{.}\text{erase}(q1, q2))</td>
<td>iterator</td>
<td>Requires: For \texttt{vector} and \texttt{deque}, (T) shall be \texttt{MoveAssignable}. Effects: Erases the elements in the range ([q1, q2)).</td>
</tr>
<tr>
<td>a(\text{.}\text{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. (\text{post}: a\text{.}\text{empty}()) returns \texttt{true}. (\text{Complexity}: \text{Linear}).</td>
</tr>
<tr>
<td>a(\text{.}\text{assign}(i, j))</td>
<td>void</td>
<td>Requires: (T) shall be \texttt{EmplaceConstructible} into (X) from (*i) and assignable from (*i). For \texttt{vector}, if the iterator does not meet the forward iterator requirements (24.2.5), (T) shall also be \texttt{MoveInsertable} into (X). Each iterator in the range ([i, j)) shall be dereferenced exactly once. (\text{pre}: i, j) are not iterators into (a). Replaces elements in (a) with a copy of ([i, j)).</td>
</tr>
<tr>
<td>a(\text{.}\text{assign}(il))</td>
<td>void</td>
<td>(\text{Requires: } T\text{ shall be } \texttt{CopyInsertable} \text{ into } X\text{ and } \texttt{CopyAssignable}). (\text{pre}: t) is not a reference into (a). Replaces elements in (a) with (n) copies of (t).</td>
</tr>
</tbody>
</table>

5. \texttt{iterator} and \texttt{const\_iterator} types for sequence containers shall be at least of the forward iterator category.

6. The iterator returned from \(a\text{.}\text{insert}(p, t)\) points to the copy of \(t\) inserted into \(a\).

7. The iterator returned from \(a\text{.}\text{insert}(p, \text{rv})\) points to the copy of \(\text{rv}\) inserted into \(a\).
The iterator returned from `a.insert(p, n, t)` points to the copy of the first element inserted into `a`, or `p` if `n == 0`.

The iterator returned from `a.insert(p, i, j)` points to the copy of the first element inserted into `a`, or `p` if `i == j`.

The iterator returned from `a.insert(p, il)` points to the copy of the first element inserted into `a`, or `p` if `il` is empty.

The iterator returned from `aemplace(p, args)` points to the new element constructed from `args` into `a`.

The iterator returned from `a.erase(q)` points to the element immediately following `q` prior to the element being erased. If no such element exists, `a.end()` is returned.

The iterator returned by `a.erase(q1,q2)` points to the element pointed to by `q2` prior to any elements being erased. If no such element exists, `a.end()` is returned.

For every sequence container defined in this Clause and in Clause 21:

(14.1) — 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.

(14.2) — If the member functions of the forms:

```cpp
template <class InputIterator> // such as insert()
rt fx1(const_iterator p, InputIterator first, InputIterator last);
```

```cpp
template <class InputIterator> // such as append(), assign()
rt fx2(InputIterator first, InputIterator last);
```

```cpp
template <class InputIterator> // such as replace()
rt fx3(const_iterator i1, const_iterator i2, InputIterator first, InputIterator last);
```

are called with a type `InputIterator` that does not qualify as an input iterator, then these functions shall not participate in overload resolution.

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.

Table 101 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>a.front()</td>
<td>reference; const_reference</td>
<td>*a.begin()</td>
<td>basic_, string, array, deque, forward_, list, list, vector</td>
</tr>
<tr>
<td></td>
<td>for constant a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 101 — 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.back()</td>
<td>reference; const_reference</td>
<td>{ auto tmp = a.end(); --tmp; return *tmp; }</td>
<td>basic_string, array, deque, list, vector</td>
</tr>
<tr>
<td>a.emplace_front(args)</td>
<td>void</td>
<td>Prepends an object of type T constructed with std::forward&lt;Args&gt;(args)....</td>
<td>deque, forward_list, list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requires: T shall be EmplaceConstructible into X from args.</td>
<td></td>
</tr>
<tr>
<td>a.emplace_back(args)</td>
<td>void</td>
<td>Appends an object of type T constructed with std::forward&lt;Args&gt;(args)....</td>
<td>deque, list, vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requires: T shall be EmplaceConstructible into X from args. For vector, T shall also be MoveInsertable into X.</td>
<td></td>
</tr>
<tr>
<td>a.push_front(t)</td>
<td>void</td>
<td>Prepends a copy of t. Requires: T shall be CopyInsertable into X.</td>
<td>deque, forward_list, list, list</td>
</tr>
<tr>
<td>a.push_front(rv)</td>
<td>void</td>
<td>Prepends a copy of rv. Requires: T shall be MoveInsertable into X.</td>
<td>deque, forward_list, list, list</td>
</tr>
<tr>
<td>a.push_back(t)</td>
<td>void</td>
<td>Appends a copy of t. Requires: T shall be 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 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: a.empty() shall be false.</td>
<td>deque, forward_list, list, list</td>
</tr>
<tr>
<td>a.pop_back()</td>
<td>void</td>
<td>Destroys the last element. Requires: 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>
</tbody>
</table>

§ 23.2.3  756
The member function at() provides bounds-checked access to container elements. at() throws out_of_range if n >= a.size().

### 23.2.4 Associative containers

Associative containers provide fast retrieval of data based on keys. The library provides four basic kinds of associative containers: set, multiset, map and multimap.

Each associative container is parameterized on Key and an ordering relation Compare that induces a strict weak ordering (25.4) on elements of Key. In addition, map and multimap associate an arbitrary mapped type T with the Key. The object of type Compare is called the comparison object of a container.

The phrase “equivalence of keys” means the equivalence relation imposed by the comparison and operator== on keys. That is, two keys k1 and k2 are considered to be equivalent if for the comparison object comp, comp(k1, k2) == false && comp(k2, k1) == false. For any two keys k1 and k2 in the same container, calling comp(k1, k2) shall always return the same value.

An associative container supports unique keys if it may contain at most one element for each key. Otherwise, it supports equivalent keys. The set and map classes support unique keys; the multiset and multimap classes support equivalent keys. For multiset and multimap, insert, emplace, and erase preserve the relative ordering of equivalent elements.

For set and multiset the value type is the same as the key type. For map and multimap it is equal to pair<const Key, T>.

iterator of an associative container is of the bidirectional iterator category. For associative containers where the value type is the same as the key type, both iterator and const_iterator are constant iterators. It is unspecified whether or not iterator and const_iterator are the same type. [Note: iterator and const_iterator have identical semantics in this case, and iterator is convertible to const_iterator. Users can avoid violating the One Definition Rule by always using const_iterator in their function parameter lists. — end note]

The associative containers meet all the requirements of Allocator-aware containers (23.2.1), except that for map and multimap, the requirements placed on value_type in Table 96 apply instead to key_type and mapped_type. [Note: For example, in some cases key_type and mapped_type are required to be CopyAssignable even though the associated value_type, pair<const key_type, mapped_type>, is not CopyAssignable. — end note]

In Table 102, X denotes an associative container class, a denotes a value of X, a_uniq denotes a value of X when X supports unique keys, a_eq denotes a value of X when X supports multiple keys, a_tran denotes a value of X when the qualified-id X::key_compare::is_transparent is valid and denotes a type (14.8.2), i and j satisfy input iterator requirements and refer to elements implicitly convertible to value_type, [i,j) denotes a valid range, p denotes a valid const iterator to a, q denotes a valid dereferenceable const iterator to a, [qi, q2) denotes a valid range of const iterators in a, il designates an object of type initializer_list<T::value_type>, t denotes a value of X::value_type, k denotes a value of X::key_type and c denotes a value of type X::key_compare; kl is a value such that a is partitioned (25.4) 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 std::allocator<X::value_type> otherwise, and $m$ denotes an allocator of a type convertible to $A$.

Table 102 — Associative container requirements (in addition to container)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X::key_type$</td>
<td>Key</td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>mapped_type (map and multimap only)</td>
<td>$T$</td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>$X::value_type$ (set and multiset only)</td>
<td>Key</td>
<td>Requires: value_type is Erasable from $X$</td>
<td>compile time</td>
</tr>
<tr>
<td>$X::value_type$ (map and multimap only)</td>
<td>pair&lt;const Key, $T$&gt;</td>
<td>Requires: value_type is Erasable from $X$</td>
<td>compile time</td>
</tr>
<tr>
<td>$X::key_compare$</td>
<td>Compare</td>
<td>defaults to less$&lt;$key_type$&gt;$</td>
<td>compile time</td>
</tr>
<tr>
<td>$X::value_compare$</td>
<td>a binary predicate type</td>
<td>is the same as key_compare for set and multiset; is an ordering relation on pairs induced by the first component (i.e., Key) for map and multimap.</td>
<td>compile time</td>
</tr>
<tr>
<td>$X(c)$</td>
<td></td>
<td>Requires: key_compare is CopyConstructible. Effects: Constructs an empty container. Uses a copy of $c$ as a comparison object.</td>
<td>constant</td>
</tr>
<tr>
<td>$X a(c);$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X()$</td>
<td></td>
<td>Requires: key_compare is DefaultConstructible. Effects: Constructs an empty container. Uses Compare() as a comparison object</td>
<td>constant</td>
</tr>
</tbody>
</table>
Table 102 — 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>X a(i,j,c);</td>
<td>Requires: key_compare is CopyConstructible. 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 (N has the value distance(i, j)); linear if [i, j) is sorted with value_comp()</td>
</tr>
<tr>
<td>X(i,j)</td>
<td>X a(i,j);</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(il);</td>
<td>Same as X(il.begin(), il.end()).</td>
<td>Same as X(il.begin(), il.end()).</td>
<td></td>
</tr>
<tr>
<td>X(il,c);</td>
<td>Same as X(il.begin(), il.end(), c).</td>
<td>Same as X(il.begin(), il.end(), c).</td>
<td></td>
</tr>
<tr>
<td>a = il X&amp;</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>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>
<td></td>
</tr>
<tr>
<td>a.key_-comp()</td>
<td>X::key_-compare</td>
<td>returns the comparison object out of which a was constructed.</td>
<td>constant</td>
</tr>
<tr>
<td>a.value_-comp()</td>
<td>X::value_-compare</td>
<td>returns an object of value_compare constructed out of the comparison object</td>
<td>constant</td>
</tr>
</tbody>
</table>
Table 102 — Associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_uniq. emplace(args)</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>
<tr>
<td>a_eq. emplace(args)</td>
<td>iterator</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)... and returns the iterator pointing to the newly inserted element. If a range containing elements equivalent to t exists in a_eq, t is inserted at the end of that range.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>a.emplace_hint(p, args)</td>
<td>iterator</td>
<td>equivalent to a.emplace(std::forward&lt;Args&gt;(args)...). Return value is an iterator pointing to the element with the key equivalent to the newly inserted element. The element is inserted as close as possible to the position just prior to p.</td>
<td>logarithmic in general, but amortized constant if the element is inserted right before p</td>
</tr>
</tbody>
</table>
Table 102 — Associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_uniq. insert(t)</td>
<td>pair&lt;iterator, bool&gt;</td>
<td>Requires: If ( t ) is a non-const value expression, ( \text{value}<em>\text{type} ) shall be \text{MoveInsertable} into ( X ); otherwise, ( \text{value}</em>\text{type} ) shall be \text{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 \text{bool} component of the returned pair is true if and only if the insertion takes place, and the \text{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 value expression, ( \text{value}<em>\text{type} ) shall be \text{MoveInsertable} into ( X ); otherwise, ( \text{value}</em>\text{type} ) shall be \text{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>
<tr>
<td>a.insert(p, t)</td>
<td>iterator</td>
<td>Requires: If ( t ) is a non-const value expression, ( \text{value}<em>\text{type} ) shall be \text{MoveInsertable} into ( X ); otherwise, ( \text{value}</em>\text{type} ) shall be \text{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>
</tbody>
</table>
Table 102 — 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(i, j)</td>
<td>void</td>
<td>Requires: value_type shall be EmplaceConstructible into X from *i. pre: 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>$N \log(a.size() + N)$ (N has the value $\text{distance}(i, j)$)</td>
</tr>
<tr>
<td>a.insert(il)</td>
<td>void</td>
<td>Equivalent to a.insert(il.begin(), il.end()).</td>
<td></td>
</tr>
<tr>
<td>a.erase(k)</td>
<td>size_type</td>
<td>erases all elements in the container with key equivalent to k. returns the number of erased elements.</td>
<td>$\log(a.size()) + a.count(k)$</td>
</tr>
<tr>
<td>a.erase(q)</td>
<td>iterator</td>
<td>erases the element pointed to by q. Returns an iterator pointing to the element immediately following q prior to the element being erased. If no such element exists, returns a.end().</td>
<td>amortized constant</td>
</tr>
<tr>
<td>a.erase(q1, q2)</td>
<td>iterator</td>
<td>erases all the elements in the range [q1,q2). Returns an iterator pointing to the element pointed to by q2 prior to any elements being erased. If no such element exists, a.end() is returned.</td>
<td>$\log(a.size()) + N$ where N has the value $\text{distance}(q1, q2)$.</td>
</tr>
<tr>
<td>a.clear()</td>
<td>void</td>
<td>a.erase(a.begin(),a.end()) linear in a.size(). post: a.empty() returns true</td>
<td></td>
</tr>
<tr>
<td>a.find(k)</td>
<td>iterator; const_iterator for constant a.</td>
<td>returns an iterator pointing to an element with the key equivalent to k, or a.end() if such an element is not found</td>
<td>logarithmic</td>
</tr>
<tr>
<td>a_tran. find(ke)</td>
<td>iterator; const_iterator for constant a_tran.</td>
<td>returns an iterator pointing to an element with key r such that !c(r, ke) &amp;&amp; !c(ke, r), or a_tran.end() if such an element is not found</td>
<td>logarithmic</td>
</tr>
</tbody>
</table>
Table 102 — 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.count(k)</code></td>
<td><code>size_type</code></td>
<td>returns the number of elements with key equivalent to <code>k</code></td>
<td>( \log(a.size()) + a.count(k) )</td>
</tr>
<tr>
<td><code>a_tran.count(ke)</code></td>
<td><code>size_type</code></td>
<td>returns the number of elements with key <code>r</code> such that ( !c(r, ke) &amp;&amp; !c(ke, r) )</td>
<td>( \log(a_tran.size()) + a_tran.count(ke) )</td>
</tr>
<tr>
<td><code>a.lower_bound(k)</code></td>
<td><code>iterator; const_iterator</code> for constant <code>a</code></td>
<td>returns an iterator pointing to the first element with key not less than <code>k</code>, or <code>a.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><code>iterator; const_iterator</code> for constant <code>a_tran</code></td>
<td>returns an iterator pointing to the first element with key such that ( !c(r, kl) ), or <code>a_tran.end()</code> if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>a.upper_bound(k)</code></td>
<td><code>iterator; const_iterator</code> for constant <code>a</code></td>
<td>returns an iterator pointing to the first element with key greater than <code>k</code>, or <code>a.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><code>iterator; const_iterator</code> for constant <code>a_tran</code></td>
<td>returns an iterator pointing to the first element with key such that ( c(ku, r) ), or <code>a_tran.end()</code> if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>a.equal_range(k)</code></td>
<td><code>pair&lt;iterator, iterator&gt;; pair&lt;const_iterator, const_iterator&gt;</code> for constant <code>a</code></td>
<td>equivalent to <code>make_pair(a.lower_bound(k), a.upper_bound(k))</code>.</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;</code> for constant <code>a_tran</code></td>
<td>equivalent to <code>make_pair(a_tran.lower_bound(ke), a_tran.upper_bound(ke))</code>.</td>
<td>logarithmic</td>
</tr>
</tbody>
</table>

9 The `insert` and `emplace` members shall not affect the validity of iterators and references to the container, and the `erase` members shall invalidate only iterators and references to the erased elements.

10 The 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,
value_comp(*j, *i) == false

For associative containers with unique keys the stronger condition holds,

value_comp(*i, *j) != 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 (14.8.2).

### 23.2.4.1 Exception safety guarantees

For associative containers, no clear() function throws an exception. erase(k) does not throw an exception unless that exception is thrown by the container’s Compare object (if any).

For associative containers, if an exception is thrown by any operation from within an insert or emplace function inserting a single element, the insertion has no effect.

For associative containers, no swap function throws an exception unless that exception is thrown by the swap of the container’s Compare object (if any).

### 23.2.5 Unordered associative containers

Unordered associative containers provide an ability for fast retrieval of data based on keys. The worst-case complexity for most operations is linear, but the average case is much faster. The library provides four unordered associative containers: unordered_set, unordered_map, unordered_multiset, and unordered_multimap.

Unordered associative containers conform to the requirements for Containers (23.2), except that the expressions a == b and a != b have different semantics than for the other container types.

Each unordered associative container is parameterized by Key, by a function object type Hash that meets the Hash requirements (17.6.3.4) and acts as a hash function for argument values of type Key, and by a binary predicate Pred that induces an equivalence relation on values of type Key. Additionally, unordered_map and unordered_multimap associate an arbitrary mapped type T with the Key.

The container’s object of type Hash — denoted by hash — is called the hash function of the container. The container’s object of type Pred — denoted by pred — is called the key equality predicate of the container.

Two values k1 and k2 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 k2 in the same container, calling pred(k1, k2) shall always return the same value. For any key k in a container, calling hash(k) shall always return the same value.

An unordered associative container supports unique keys if it may contain at most one element for each key. Otherwise, it supports equivalent keys. unordered_set and unordered_map support unique keys. unordered_multiset and unordered_multimap support equivalent keys. In containers that support equivalent keys, elements with equivalent keys are adjacent to each other in the iteration order of the container. Thus, although the absolute order of elements in an unordered container is not specified, its elements are grouped into equivalent-key groups such that all elements of each group have equivalent keys. Mutating
operations on unordered containers shall preserve the relative order of elements within each equivalent-key group unless otherwise specified.

For **unordered_set** and **unordered_multiset** the value type is the same as the key type. For **unordered_map** and **unordered_multimap** it is `std::pair<const Key, T>`.

8 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]*

9 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.

10 The unordered associative containers meet all the requirements of Allocator-aware containers (23.2.1), except that for **unordered_map** and **unordered_multimap**, the requirements placed on **value_type** in Table 96 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]*

11 In Table 103: **X** is an unordered associative container class, **a** is an object of type **X**, **b** is a possibly const object of type **X**, **a_uniq** is an object of type **X** when **X** supports unique keys, **a_eq** is an object of type **X** when **X** supports equivalent keys, **i** and **j** are input iterators that refer to **value_type**, **[i, j)** is a valid range, **p** and **q2** are valid const iterators to **a**, **q** and **q1** are valid dereferenceable const iterators to **a**, **[q1, q2)** is a valid range in **a**, **il** designates an object of type **initializer_list<value_type>**, **t** is a value of type **X::value_type**, **k** is a value of type **key_type**, **hf** is a possibly const value of type **hashed**, **eq** is a possibly const value of type **key_equal**, **n** is a value of type **size_type**, and **z** is a value of type **float**.

<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><strong>X::key_type</strong></td>
<td>Key</td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td><strong>X::mapped_type</strong> (unordered_map and unordered_multimap only)</td>
<td>T</td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td><strong>X::value_type</strong> (unordered_set and unordered_multiset only)</td>
<td>Key</td>
<td>Requires: value_type is Erasable from <strong>X</strong></td>
<td>compile time</td>
</tr>
<tr>
<td><strong>X::value_type</strong> (unordered_map and unordered_multimap only)</td>
<td><code>pair&lt;const Key, T&gt;</code></td>
<td>Requires: value_type is Erasable from <strong>X</strong></td>
<td>compile time</td>
</tr>
<tr>
<td><strong>X::hasher</strong></td>
<td>Hash</td>
<td>Hash shall be a unary function object type such that the expression <code>hf(k)</code> has type <code>std::size_t</code></td>
<td>compile time</td>
</tr>
</tbody>
</table>
Table 103 — Unordered associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::key_equal</td>
<td>Pred</td>
<td>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(n, hf, eq)</td>
<td>X</td>
<td>Requires: hasher and key_equal are CopyConstructible. Effects: Constructs an empty container with at least n buckets, using hf as the hash function and eq as the key equality predicate.</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>X a(n, hf, eq)</td>
<td>X</td>
<td>Requires: hasher and key_equal are CopyConstructible and key_equal is DefaultConstructible. Effects: Constructs an empty container with at least n buckets, using hf as the hash function and key_equal() as the key equality predicate.</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>X(n, hf)</td>
<td>X</td>
<td>Requires: hasher is CopyConstructible and key_equal is DefaultConstructible. Effects: Constructs an empty container with at least n buckets, using hf as the hash function and key_equal() as the key equality predicate.</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>X(n)</td>
<td>X</td>
<td>Requires: hasher and key_equal are DefaultConstructible. Effects: Constructs an empty container with at least n buckets, using hasher() as the hash function and key_equal() as the key equality predicate.</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Table 103 — Unordered associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X())</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\ (i, j, n, hf, eq))</td>
<td>(X)</td>
<td>Requires: hasher and key_equal are CopyConstructible. 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))</td>
<td>(X)</td>
<td>Requires: hasher is CopyConstructible and 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>
</tbody>
</table>
Table 103 — 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, n)</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 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 ( \Theta(N) ) (N is distance(i, j)), worst case ( \Theta(N^2) )</td>
</tr>
<tr>
<td><code>X a(i, j, n)</code></td>
<td><code>X</code></td>
<td>Same as <code>X(i, j, n)</code>.</td>
<td></td>
</tr>
<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 ( \Theta(N) ) (N is distance(i, j)), worst case ( \Theta(N^2) )</td>
</tr>
<tr>
<td><code>X a(i, j)</code></td>
<td><code>X</code></td>
<td>Same as <code>X(i, j)</code>.</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 96, 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>Same as <code>X(b)</code>.</td>
<td></td>
</tr>
</tbody>
</table>

§ 23.2.5
Table 103 — 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 = b X&amp;</td>
<td>Copy assignment operator. In addition to the requirements of Table 96, copies the hash function, predicate, and maximum load factor.</td>
<td>Average case linear in b.size(), worst case quadratic.</td>
<td></td>
</tr>
<tr>
<td>a = il X&amp;</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 a = X(il).</td>
<td></td>
</tr>
<tr>
<td>b.hash_function() hasher</td>
<td>Returns b’s hash function.</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>b.key_eq() key_equal</td>
<td>Returns b’s key equality predicate.</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>a_uniq. emplace(args) 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>Average case (\mathcal{O}(1)), worst case (\mathcal{O}(a_uniq.size())).</td>
<td></td>
</tr>
<tr>
<td>a_eq.emplace(args) iterator</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)... and returns the iterator pointing to the newly inserted element.</td>
<td>Average case (\mathcal{O}(1)), worst case (\mathcal{O}(a_eq.size())).</td>
<td></td>
</tr>
</tbody>
</table>
Table 103 — 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.emplace_hint(p, args)</td>
<td>iterator</td>
<td>Requires: value_type shall be EmplaceConstructible into X from args. Effects: Equivalent to a.emplace(std::forward&lt;Args&gt;(args)...). Return value is an iterator pointing to the element with the key equivalent to the newly inserted element. The const_iterator p is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.</td>
<td>Average case $O(1)$, worst case $O(a.size())$.</td>
</tr>
<tr>
<td>a_uniq.insert(t)</td>
<td>pair&lt;iterator, bool&gt;</td>
<td>Requires: If t is a non-const value 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 indicates whether the insertion takes place, and the iterator component points to the element with key equivalent to the key of t.</td>
<td>Average case $O(1)$, worst case $O(a_uniq.size())$.</td>
</tr>
<tr>
<td>a_eq.insert(t)</td>
<td>iterator</td>
<td>Requires: If t is a non-const value expression, value_type shall be MoveInsertable into X; otherwise, value_type shall be CopyInsertable 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>
</tbody>
</table>
Table 103 — 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.insert(q, t)</code></td>
<td>iterator</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: Equivalent to <code>a.insert(t)</code>. Return value is an iterator pointing to the element with the key equivalent to that of <code>t</code>. The iterator <code>q</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.insert(i, j)</code></td>
<td>void</td>
<td>Requires: <code>value_type</code> shall be <code>EmplaceConstructible</code> into <code>X</code> from <code>*i</code>. Pre: <code>i</code> and <code>j</code> are not iterators in <code>a</code>. Equivalent to <code>a.insert(t)</code> for each element in <code>[i,j)</code>.</td>
<td>Average case $O(N)$, where $N$ is <code>distance(i, j)</code>. Worst case $O(N \times a.size()) + N)$.</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>
<tr>
<td><code>a.erase(k)</code></td>
<td>size_type</td>
<td>Erases all elements with key equivalent to <code>k</code>. Returns the number of elements erased.</td>
<td>Average case $O(a.count(k))$. Worst case $O(a.size())$.</td>
</tr>
<tr>
<td><code>a.erase(q)</code></td>
<td>iterator</td>
<td>Erases the element pointed to by <code>q</code>. Return value is the iterator immediately following <code>q</code> prior to the erasure.</td>
<td>Average case $O(1)$, worst case $O(a.size())$.</td>
</tr>
<tr>
<td><code>a.erase(q1, q2)</code></td>
<td>iterator</td>
<td>Erases all elements in the range <code>[q1, q2)</code>. Return value is the iterator immediately following the erased elements prior to the erasure.</td>
<td>Average case linear in <code>distance(q1, q2)</code>, worst case $O(a.size())$.</td>
</tr>
<tr>
<td><code>a.clear()</code></td>
<td>void</td>
<td>Erases all elements in the container. Post: <code>a.empty()</code> returns <code>true</code></td>
<td>Linear.</td>
</tr>
</tbody>
</table>

<p>| b.find(k) | iterator; const_iterator for const b. | Returns an iterator pointing to an element with key equivalent to <code>k</code>, or <code>b.end()</code> if no such element exists. | Average case $O(1)$, worst case $O(b.size())$. |</p>
<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>b.count(k)</td>
<td>size_type</td>
<td>Returns the number of elements with key equivalent to k.</td>
<td>Average case $O(b.count(k))$, worst case $O(b.size())$.</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 <code>make_pair(b.end(), b.end())</code> if no such elements exist.</td>
<td>Average case $O(b.count(k))$. Worst case $O(b.size())$.</td>
</tr>
<tr>
<td>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>Pre: $b.bucket_count() &gt; 0$. Returns the index of the bucket in which elements with keys equivalent to k would be found, if any such element existed. Post: 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>Pre: n shall be in the range $[0, b.bucket_count())$. Returns the number of elements in the n-th bucket.</td>
<td>$O(b.bucket_size(n))$</td>
</tr>
<tr>
<td>b.begin(n)</td>
<td>local_iterator; const_local_iterator for const b.</td>
<td>Pre: 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>Pre: 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>Pre: n shall be in the range $[0, b.bucket_count())$. Note: [b.cbegin(n), b.cend(n)] is a valid range containing all of the elements in the n-th bucket.</td>
<td>Constant</td>
</tr>
<tr>
<td>b.cend(n)</td>
<td>const_local_iterator</td>
<td>Pre: n shall be in the range $[0, b.bucket_count())$.</td>
<td>Constant</td>
</tr>
</tbody>
</table>
Table 103 — 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.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>Pre: 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>

Two unordered containers `a` and `b` compare equal if `a.size() == b.size()` and, for every equivalent-key group \([Ea1,Ea2]\) obtained from `a.equal_range(Ea1)`, there exists an equivalent-key group \([Eb1,Eb2]\) obtained from `b.equal_range(Ea1)`, such that `is_permutation(Ea1, Ea2, Eb1, Eb2)` returns `true`. For unordered_set and unordered_map, the complexity of `operator==` (i.e., the number of calls to the `==` operator of the `value_type`, to the predicate returned by `key_equal()`, and to the hasher returned by `hash_function()`) is proportional to \(N\) in the average case and to \(N^2\) in the worst case, where \(N\) is `a.size()`. For unordered_multiset and unordered_multimap, the complexity of `operator==` is proportional to \(\sum E_i^2\) in the average case and to \(N^2\) in the worst case, where \(N\) is `a.size()`, and \(E_i\) is the size of the \(i^{th}\) equivalent-key group in `a`. However, if the respective elements of each corresponding pair of equivalent-key groups \(Ea_i\) and \(Eb_i\) are arranged in the same order (as is commonly the case, e.g., if `a` and `b` are unmodified copies of the same container), then the average-case complexity for unordered_multiset and unordered_multimap becomes proportional to \(N\) (but worst-case complexity remains \(O(N^2)\), e.g., for a pathologically bad hash function). The behavior of a program that uses `operator==` or `operator!=` on unordered containers is undefined unless the `Hash` and `Pred` function objects respectively have the same behavior for both containers and the equality comparison operator for `Key` is a refinement\(^{263}\) of the partition into equivalent-key groups produced by `Pred`.

---

\(^{12}\) Equality comparison is a refinement of partitioning if no two objects that compare equal fall into different partitions.

\(^{263}\) Equality comparison is a refinement of partitioning if no two objects that compare equal fall into different partitions.

---

\(^{13}\) 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 const iterators.

\(^{263}\) Equality comparison is a refinement of partitioning if no two objects that compare equal fall into different partitions.
The `insert` and `emplace` members shall not affect the validity of references to container elements, but may invalidate all iterators to the container. The `erase` members shall invalidate only iterators and references to the erased elements, and preserve the relative order of the elements that are not erased.

The `insert` and `emplace` members shall not affect the validity of 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.

### 23.2.5.1 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` or `Comparison` function, the `rehash()` function has no effect.

### 23.3 Sequence containers

#### 23.3.1 In general

The headers `<array>`, `<deque>`, `<forward_list>`, `<list>`, and `<vector>` define template classes that meet the requirements for sequence containers.

The headers `<queue>` and `<stack>` define container adaptors (23.6) that also meet the requirements for sequence containers.

**Header `<array>` synopsis**

```
#include <initializer_list>

namespace std {
    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;

Header <deque> synopsis

#include <initializer_list>

namespace std {
    template <class T, class Allocator = allocator<T>> class deque;
    template <class T, class Allocator>
    bool operator==(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
    template <class T, class Allocator>
    bool operator<(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
    template <class T, class Allocator>
    bool operator!=(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
    template <class T, class Allocator>
    bool operator>(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
    template <class T, class Allocator>
    bool operator>=(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
    template <class T, class Allocator>
    bool operator<=(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
    template <class T, class Allocator>
    void swap(deque<T, Allocator>& x, deque<T, Allocator>& y)
        noexcept(noexcept(x.swap(y)));
}

Header <forward_list> synopsis

#include <initializer_list>

namespace std {
    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)));
}

Header <list> synopsis

#include <initializer_list>

§ 23.3.1
namespace std {
  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)));
}

Header <vector> synopsis

#include <initializer_list>

namespace std {
  template <class T, class Allocator = allocator<T> > class vector;
  template <class T, class Allocator>
  bool operator==(const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <class T, class Allocator>
  bool operator< (const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <class T, class Allocator>
  bool operator!=(const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <class T, class Allocator>
  bool operator> (const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <class T, class Allocator>
  bool operator>=(const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <class T, class Allocator>
  bool operator<=(const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <class T, class Allocator>
  void swap(vector<T,Allocator>& x, vector<T,Allocator>& y)
    noexcept(noexcept(x.swap(y)));

  template <class Allocator> class vector<bool,Allocator>;
  // hash support
  template <class T> struct hash;
  template <class Allocator> struct hash<vector<bool, Allocator> >;
}

23.3.2 Class template array

23.3.2.1 Class template array overview

The header <array> defines a class template for storing fixed-size sequences of objects. An array is a contiguous container (23.2.1). An instance of array<T, N> stores N elements of type T, so that size() == N is an invariant.
An array is an aggregate (8.5.1) that can be initialized with the syntax

```cpp
array<T, N> a = { initializer-list };```

where initializer-list is a comma-separated list of up to N elements whose types are convertible to T.

An array satisfies all of the requirements of a container and of a reversible container (23.2), 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 (23.2.3). 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:
        typedef T& reference;
        typedef const T& const_reference;
        typedef implementation-defined iterator;
        typedef implementation-defined const_iterator;
        typedef size_t size_type;
        typedef ptrdiff_t difference_type;
        typedef T value_type;
        typedef T* pointer;
        typedef const T* const_pointer;
        typedef std::reverse_iterator<iterator> reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

        T elems[N]; // exposition only

        // no explicit construct/copy/destroy for aggregate type

        void fill(const T& u);
        void swap(array&) noexcept(noexcept(swap(declval<T&>(), declval<T&>()));

        // 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:
        constexpr size_type size() const noexcept;
        constexpr size_type max_size() const noexcept;
        constexpr bool empty() const noexcept;

        // element access:
    }
}
```

§ 23.3.2.1
reference operator[](size_type n);
constexpr const_reference operator[](size_type n) const;
reference at(size_type n);
constexpr const_reference at(size_type n) const;
reference front();
constexpr const_reference front() const;
reference back();
constexpr const_reference back() const;

T * data() noexcept;
const T * data() const noexcept;
};

[Note: The member variable_elems_is shown for exposition only, to emphasize that_array_is a class aggregate. The name_elems_is not part of_array’s interface. — end note]

23.3.2.2 array constructors, copy, and assignment [array.cons]
The conditions for an aggregate (8.5.1) shall be met. Class_array_relied on the implicitly-declared special
member functions (12.1, 12.4, and 12.8) to conform to the container requirements table in 23.2. 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.

23.3.2.3 array specialized algorithms [array.special]
template <class T, size_t N> void swap(array<T,N>& x, array<T,N>& y) noexcept(noexcept(x.swap(y)));
  Effects:
    x.swap(y);
  Complexity: Linear in N.

23.3.2.4 array::size [array.size]
template <class T, size_t N> constexpr size_type array<T,N>::size() const noexcept;
  Returns: N

23.3.2.5 array::data [array.data]
T* data() noexcept;
const T* data() const noexcept;
  Returns: elems.

23.3.2.6 array::fill [array.fill]
void fill(const T& u);
  Effects: fill_n(begin(), N, u)

23.3.2.7 array::swap [array.swap]
void swap(array& y) noexcept(noexcept(swap(declval<T&>(), declval<T&>())))
  Effects: swap_ranges(begin(), end(), y.begin())
  Throws: Nothing unless one of the element-wise swap calls throws an exception.
  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.
23.3.2.8 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 noexcept-specification which is equivalent to noexcept(true).

23.3.2.9 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.

2 Value: The type T.

template <size_t I, class T, size_t N>
constexpr T& get(array<T, N>& a) noexcept;

3 Requires: I < N. The program is ill-formed if I is out of bounds.

4 Returns: A reference to the Ith element of a, where indexing is zero-based.

template <size_t I, class T, size_t N>
constexpr T&& get(array<T, N>&& a) noexcept;

5 Effects: Equivalent to return std::move(get<I>(a));

template <size_t I, class T, size_t N>
constexpr const T& get(const array<T, N>& a) noexcept;

6 Requires: I < N. The program is ill-formed if I is out of bounds.

7 Returns: A const reference to the Ith element of a, where indexing is zero-based.

23.3.3 Class template deque

23.3.3.1 Class template deque overview

A deque is a sequence container that, like a vector (23.3.6), supports random access iterators. 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. As with vectors, storage management is handled automatically.

1 A deque is a sequence container that, like a vector (23.3.6), supports random access iterators. 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. As with vectors, storage management is handled automatically.

2 A deque satisfies all of the requirements of a container, of a reversible container (given in tables in 23.2), of a sequence container, including the optional sequence container requirements (23.2.3), and of an allocator-aware container (Table 99). 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:
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef implementation-defined iterator; // See 23.2

  § 23.3.3.1
typedef implementation-defined const_iterator;  // See 23.2
typedef implementation-defined size_type;   // See 23.2
typedef implementation-defined difference_type; // See 23.2
typedef T value_type;
typedef Allocator allocator_type;
typedef typename allocator_traits<Allocator>::pointer pointer;
typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

// 23.3.3.2, construct/copy/destroy:
deque() : deque(Allocator()) { }
explicit deque(const Allocator&);
explicit deque(size_type n, const Allocator& = Allocator());
deque(size_type n, const T& value, const Allocator& = Allocator());
template <class InputIterator>
deque(InputIterator first, InputIterator last, const Allocator& = Allocator());
deque(const deque& x);
deque(deque&&);
deque(const deque&, const Allocator&);
deque(deque&&, const Allocator&);
deque(initializer_list<T>, const Allocator& = Allocator());
~deque();
deque& operator=(const deque& x);
deque& operator=(deque&& x);
    noexcept(allocator_traits<Allocator>::is_always_equal::value);
deque& operator=(initializer_list<T>);
template <class InputIterator>
void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& t);
void assign(initializer_list<T>);
allocator_type get_allocator() const noexcept;

// iterators:
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// 23.3.3.3, capacity:
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();

§ 23.3.3.1
bool empty() const noexcept;

// element access:
reference operator[](size_type n);
const_reference operator[](size_type n) const;
reference at(size_type n);
const_reference at(size_type n) const;
reference front();
const_reference front() const;
reference back();
const_reference back() const;

// 23.3.3.4, modifiers:
template <class... Args> void emplace_front(Args&&... args);
template <class... Args> void 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 T, class Allocator>
bool operator==(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
template <class T, class Allocator>
bool operator< (const deque<T,Allocator>& x, const deque<T,Allocator>& y);
template <class T, class Allocator>
bool operator!=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
template <class T, class Allocator>
bool operator> (const deque<T,Allocator>& x, const deque<T,Allocator>& y);
template <class T, class Allocator>
bool operator>=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
template <class T, class Allocator>
bool operator<=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);

// specialized algorithms:
template <class T, class Allocator>
void swap(deque<T,Allocator>& x, deque<T,Allocator>& y)
```c
    noexcept(noexcept(x.swap(y)));
}
```

### 23.3.3.2 deque constructors, copy, and assignment

[deque.cons]

```c
explicit deque(const Allocator&);
```

1. **Effects:** Constructs an empty `deque`, using the specified allocator.
2. **Complexity:** Constant.

```c
explicit deque(size_type n, const Allocator& = Allocator());
```

3. **Effects:** Constructs a `deque` with `n` default-inserted elements using the specified allocator.
4. **Requires:** `T` shall be `DefaultInsertable` into `*this`.
5. **Complexity:** Linear in `n`.

```c
dele(size_type n, const T& value,
    const Allocator& = Allocator());
```

6. **Effects:** Constructs a `deque` with `n` copies of `value`, using the specified allocator.
7. **Requires:** `T` shall be `CopyInsertable` into `*this`.
8. **Complexity:** Linear in `n`.

```c
    template <class InputIterator>
    deque(InputIterator first, InputIterator last,
        const Allocator& = Allocator());
```

9. **Effects:** Constructs a `deque` equal to the range `[first, last)`, using the specified allocator.
10. **Complexity:** Linear in `distance(first, last)`.

### 23.3.3.3 deque capacity

[deque.capacity]

```c
    void resize(size_type sz);
```

1. **Effects:** If `sz <= size()`, equivalent to calling `pop_back(size()) - sz` times. If `size() < sz`, appends `sz - size()` default-inserted elements to the sequence.
2. **Requires:** `T` shall be `MoveInsertable` and `DefaultInsertable` into `*this`.

```c
    void resize(size_type sz, const T& c);
```

3. **Effects:** If `sz <= size()`, equivalent to calling `pop_back(size()) - sz` times. If `size() < sz`, appends `sz - size()` copies of `c` to the sequence.
4. **Requires:** `T` shall be `CopyInsertable` into `*this`.

```c
    void shrink_to_fit();
```

5. **Requires:** `T` shall be `MoveInsertable` into `*this`.
6. **Complexity:** Linear in the size of the sequence.
7. **Remarks:** `shrink_to_fit` is a non-binding request to reduce memory use but does not change the size of the sequence. [Note: The request is non-binding to allow latitude for implementation-specific optimizations. — end note]
23.3.3.4 deque 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> void emplace_front(Args&&... args);
template <class... Args> void emplace_back(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);

1 Effects: An insertion in the middle of the deque invalidates all the iterators and references to elements of the deque. An insertion at either end of the deque invalidates all the iterators to the deque, but has no effect on the validity of references to elements of the deque.

2 Remarks: If an exception is thrown other than by the copy constructor, move constructor, assignment operator, or move assignment operator of T there are no effects. If an exception is thrown while inserting a single element at either end, there are no effects. Otherwise, if an exception is thrown by the move constructor of a non-CopyInsertable T, the effects are unspecified.

3 Complexity: The complexity is linear in the number of elements inserted plus the lesser of the distances to the beginning and end of the deque. Inserting a single element 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);

4 Effects: An erase operation that erases the last element of a deque invalidates only the past-the-end iterator and all iterators and references to the erased elements. An erase operation that erases the first element of a deque but not the last element invalidates only 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.

5 Complexity: The number of calls to the destructor is the same as the number of elements erased, but the number of calls to the assignment operator is no more than the lesser of the number of elements before the erased elements and the number of elements after the erased elements.

6 Throws: Nothing unless an exception is thrown by the copy constructor, move constructor, assignment operator, or move assignment operator of T.

23.3.3.5 deque specialized algorithms

template <class T, class Allocator>
    void swap(deque<T,Allocator>& x, deque<T,Allocator>& y)
        noexcept(noexcept(x.swap(y)));

1 Effects:
    x.swap(y);
23.3.4 Class template `forward_list`  

23.3.4.1 Class template `forward_list` overview

1 A `forward_list` is a container that supports forward iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Fast random access to list elements is not supported. [Note: It is intended that `forward_list` have zero space or time overhead relative to a hand-written C-style singly linked list. Features that would conflict with that goal have been omitted. —end note]

2 A `forward_list` satisfies all of the requirements of a container (Table 96), except that the `size()` member function is not provided and `operator==` has linear complexity. A `forward_list` also satisfies all of the requirements for an allocator-aware container (Table 99). In addition, a `forward_list` provides the `assign` member functions (Table 100) and several of the optional container requirements (Table 101). Descriptions are provided here only for operations on `forward_list` that are not described in that table or for operations where there is additional semantic information.

3 [Note: Modifying any list requires access to the element preceding the first element of interest, but in a `forward_list` there is no constant-time way to access a preceding element. For this reason, ranges that are modified, such as those supplied to `erase` and `splice`, must be open at the beginning. —end note]

```cpp
namespace std {
    template <class T, class Allocator = allocator<T> >
    class forward_list {
    public:
        // types:
        typedef value_type& reference;
        typedef const value_type& const_reference;
        typedef implementation-defined iterator;    // See 23.2
        typedef implementation-defined const_iterator; // See 23.2
        typedef implementation-defined size_type;     // See 23.2
        typedef implementation-defined difference_type; // See 23.2
        typedef T value_type;
        typedef Allocator allocator_type;
        typedef typename allocator_traits<Allocator>::pointer pointer;
        typedef typename allocator_traits<Allocator>::const_pointer const_pointer;

        // 23.3.4.2, construct/copy/destroy:
        forward_list() : forward_list(Allocator()) { }
        explicit forward_list(const Allocator&);
        explicit forward_list(size_type n, const Allocator& = Allocator());
        forward_list(size_type n, const T& value,
                     const Allocator& = Allocator());
        template <class InputIterator>
        forward_list(InputIterator first, InputIterator last,
                     const Allocator& = Allocator());
        forward_list(const forward_list& x);
        forward_list(forward_list&& x);
        forward_list(const forward_list& x, const Allocator&);
        forward_list(forward_list&& x, const Allocator&);
        forward_list(initializer_list<T>, const Allocator& = Allocator());
        ~forward_list();
        forward_list& operator=(const forward_list& x);
        forward_list& operator=(forward_list&& x)
            noexcept(allocator_traits<Allocator>::is_always_equal::value);
        forward_list& operator=(initializer_list<T>);
        template <class InputIterator>
```

§ 23.3.4.1
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;

// 23.3.4.3, 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:
bool empty() const noexcept;
size_type max_size() const noexcept;

// 23.3.4.4, element access:
reference front();
const_reference front() const;

// 23.3.4.5, modifiers:
template <class... Args> void emplace_front(Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void pop_front();

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);

// 23.3.4.6, 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;
};

// Comparison operators
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);

// 23.3.4.7, specialized algorithms:
template <class T, class Allocator>
void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>& y) noexcept(noexcept(x.swap(y)));
}

23.3.4.2 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 \( n \).

```cpp
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.

7 *Requires:* `T` shall be `CopyInsertable` into `*this`.

8 **Complexity:** Linear in \( n \).

```cpp
template <class InputIterator>
forward_list(InputIterator first, InputIterator last, const Allocator& = Allocator());
```

9 *Effects:* Constructs a `forward_list` object equal to the range \([first, last)\).

10 **Complexity:** Linear in \( \text{distance}(first, last) \).

### 23.3.4.3 forward_list iterators

```cpp
iterator before_begin() noexcept;
const_iterator before_begin() const noexcept;
const_iterator cbefore_begin() const noexcept;
```

1 *Returns:* A non-dereferenceable iterator that, when incremented, is equal to the iterator returned by `begin()`.

2 *Effects:* `cbefore_begin()` is equivalent to `const_cast<forward_list const&>(*this).before_begin()`.

3 *Remarks:* `before_begin() == end()` shall equal `false`.

### 23.3.4.4 forward_list element access

```cpp
reference front();
const_reference front() const;
```

1 *Returns:* `*begin()`.

### 23.3.4.5 forward_list modifiers

1 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> void emplace_front(Args&&... args);
```

2 *Effects:* Inserts an object of type `value_type` constructed with `value_type(std::forward<Args>(args)...)` at the beginning of the list.

```cpp
void push_front(const T& x);
void push_front(T&& x);
```

3 *Effects:* Inserts a copy of `x` at the beginning of the list.

```cpp
void pop_front();
```

4 *Effects:* `erase_after(before_begin())`

```cpp
iterator insert_after(const_iterator position, const T& x);
iterator insert_after(const_iterator position, T&& x);
```

§ 23.3.4.5
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);

29 Effects: If sz < distance(begin(), end()), erases the last distance(begin(), end()) - sz elements from the list. Otherwise, inserts sz - distance(begin(), end()) elements at the end of the list such that each new element, e, is initialized by a method equivalent to calling allocator_traits<allocator_type>::construct(get_allocator(), std::addressof(e), c).

Requires: T shall be CopyInsertable into *this.

void clear() noexcept;

31 Effects: Erases all elements in the range [begin(),end()).

Remarks: Does not invalidate past-the-end iterators.

23.3.4.6 forward_list operations [forwardlist.ops]

void splice_after(const_iterator position, forward_list& x);
void splice_after(const_iterator position, forward_list&& x);

1 Requires: position is before_begin() or is a dereferenceable iterator in the range [begin(),end()). get_allocator() == x.get_allocator(). &x != this.

2 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: $O(\text{distance}(x\cdot\text{begin}(), x\cdot\text{end}()))$

void splice_after(const_iterator position, forward_list& x, const_iterator i);
void splice_after(const_iterator position, forward_list&& x, const_iterator i);

5 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().

6 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);

9 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().

10 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: $O(\text{distance}(\text{first}, \text{last}))$
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()), \(\text{pred}(\*i)\) 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 (17.6.5.7).

Complexity: Exactly \(\text{distance}(begin(), end())\) 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 \([\text{first} + 1, \text{last})\) for which \(\*i == \*(i-1)\) (for the version with no arguments) or \(\text{pred}(\*i, \*(i - 1))\) (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 \([\text{first}, \text{last})\) is not empty, exactly \((\text{last} - \text{first}) - 1\) applications of the corresponding predicate, otherwise no applications of the predicate.

void merge(forward_list& x);
void merge(forward_list&& x);
    template <class Compare> void merge(forward_list& x, Compare comp);
    template <class Compare> void merge(forward_list&& x, Compare comp);

Requires: \(\text{comp}\) defines a strict weak ordering (25.4), and \(\text{*this}\) and \(x\) are both sorted according to this ordering. get_allocator() == x.get_allocator().

Effects: Merges the two sorted ranges \([\text{begin()}, \text{end()}\) and \([x.\text{begin()}, x.\text{end()}\). \(x\) is empty after the merge. If an exception is thrown other than by a comparison there are no effects. Pointers and references to the moved elements of \(x\) now refer to those same elements but as members of \(\text{*this}\). Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into \(\text{*this}\), not into \(x\).

Remarks: Stable (17.6.5.7). The behavior is undefined if this->get_allocator() != x.get_allocator().

Complexity: At most \(\text{distance}(begin(), end()) + \text{distance}(x.\text{begin()}, x.\text{end()}) - 1\) comparisons.

void sort();
    template <class Compare> void sort(Compare comp);

Requires: operator\(<\) (for the version with no arguments) or \(\text{comp}\) (for the version with a comparison argument) defines a strict weak ordering (25.4).

Effects: Sorts the list according to the operator\(<\) or the \(\text{comp}\) function object. If an exception is thrown the order of the elements in \(\text{*this}\) is unspecified. Does not affect the validity of iterators and references.

Remarks: Stable (17.6.5.7).

Complexity: Approximately \(N \log N\) comparisons, where \(N\) is \(\text{distance}(begin(), end())\).

void reverse() noexcept;

Effects: Reverses the order of the elements in the list. Does not affect the validity of iterators and references.

Complexity: Linear time.
23.3.4.7 forward_list specialized algorithms

template <class T, class Allocator>
void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>& y)
    noexcept(noexcept(x.swap(y)));  

1 Effects: x.swap(y)

23.3.5 Class template list

23.3.5.1 Class template list overview

A list is a sequence container that supports bidirectional iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Unlike vectors (23.3.6) and deques (23.3.3), fast random access to list elements is not supported, but many algorithms only need sequential access anyway.

A list satisfies all of the requirements of a container, of a reversible container (given in two tables in 23.2), of a sequence container, including most of the optional sequence container requirements (23.2.3), and of an allocator-aware container (Table 99). The exceptions are the operator[] and at member functions, which are not provided. Descriptions are provided here only for operations on list that are not described in one of these tables or for operations where there is additional semantic information.

namespace std {
    template <class T, class Allocator = allocator<T> >
    class list {
        public:
            // types:
            typedef value_type& reference;
            typedef const value_type& const_reference;
            typedef implementation-defined iterator;  // see 23.2
            typedef implementation-defined const_iterator;  // see 23.2
            typedef implementation-defined size_type;  // see 23.2
            typedef implementation-defined difference_type;  // see 23.2
            typedef T value_type;
            typedef Allocator allocator_type;
            typedef typename allocator_traits<Allocator>::pointer pointer;
            typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
            typedef std::reverse_iterator<iterator> reverse_iterator;
            typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

            // 23.3.5.2, 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);

264) These member functions are only provided by containers whose iterators are random access iterators.
noexcept(algorithm_traits<Observer>::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() const noexcept;
const_iterator begin() const noexcept;
iterator end() const noexcept;
const_iterator end() const noexcept;
reverse_iterator rbegin() const noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator rend() const 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;

// 23.3.5.3, capacity:
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;

// 23.3.5.4, modifiers:
template <class... Args> void emplace_front(Args&&... args);
void pop_front();
template <class... Args> void emplace_back(Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
void pop_back();
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
  iterator insert(const_iterator position, InputIterator first,
                  InputIterator last);
iterator insert(const_iterator position, initializer_list<T> il);
iterator erase(const_iterator position);
iterator erase(const_iterator position, const_iterator last);
void swap(list&)
    noexcept(algorithm_traits<Allocator>::is_always_equal::value);
void clear() noexcept;

// 23.3.5.5, 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 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);

// specialized algorithms:
template <class T, class Allocator>
    void swap(list<T,Allocator>& x, list<T,Allocator>& y)
    noexcept(noexcept(x.swap(y)));
}

23.3.5.2 list constructors, copy, and assignment

§ 23.3.5.2
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).

23.3.5.3 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:
       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.
23.3.5.4 list modifiers

```cpp
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
    iterator insert(const_iterator position, InputIterator first,
                    InputIterator last);
iterator insert(const_iterator position, initializer_list<T>);
template <class... Args> void emplace_front(Args&&... args);
template <class... Args> void emplace_back(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
```

1 Remarks: Does not affect the validity of iterators and references. If an exception is thrown there are no effects.

2 Complexity: Insertion of a single element into a list takes constant time and exactly one call to a constructor of T. Insertion of multiple elements into a list is linear in the number of elements inserted, and the number of calls to the copy constructor or move constructor of T is exactly equal to the number of elements inserted.

```cpp
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void pop_front();
void pop_back();
void clear() noexcept;
```

3 Effects: Invalidates only the iterators and references to the erased elements.

4 Throws: Nothing.

5 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.

23.3.5.5 list operations

Since lists allow fast insertion and erasing from the middle of a list, certain operations are provided specifically for them.²⁶⁵

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);
```

3 Requires: &x != this.

4 Effects: Inserts the contents of x before position and x becomes empty. Pointers and references to the moved elements of x now refer to those same elements but as members of *this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this, not into x.

²⁶⁵) As specified in 17.6.3.5, the requirements in this Clause apply only to lists whose allocators compare equal.
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);
```

Effects: Inserts an element pointed to by i from list x before position and removes the element from x. The result is unchanged if position == i or position == ++i. Pointers and references to *i continue to refer to this same element but as a member of *this. Iterators to *i (including i itself) continue to refer to the same element, but now behave as iterators into *this, not into x.

Requires: i is a valid dereferenceable iterator of 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);
```

Effects: Inserts elements in the range [first, last) before position and removes the elements from x.

Requires: [first, last) is a valid range in x. The result is undefined if position is an iterator in the range [first, last). Pointers and references to the moved elements of x now refer to those same elements but as members of *this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this, not into x.

Throws: Nothing.

Complexity: Constant time if &x == this; otherwise, linear time.

```cpp
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, 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 (17.6.5.7).

Complexity: Exactly size() applications of the corresponding predicate.

```cpp
void unique();
template <class BinaryPredicate> void unique(BinaryPredicate binary_pred);
```

Effects: Erases all but the first element from every consecutive group of equal elements referred to by the iterator i in the range [first + 1, last) for which *i == *(i-1) (for the version of unique with no arguments) or pred(*i, *(i - 1)) (for the version of unique with a predicate argument) holds. Invalidates only the iterators and references to the erased elements.

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);
template <class Compare> void merge(list& x, Compare comp);
template <class Compare> void merge(list&& x, Compare comp);

Requires: comp shall define a strict weak ordering (25.4), and both the list and the argument list shall be sorted according to this ordering.

Effects: If (&x == this) does nothing; otherwise, merges the two sorted ranges \([\text{begin}(), \text{end}())\) and \([x.\text{begin}(), x.\text{end}())\). The result is a range in which the elements will be sorted in non-decreasing order according to the ordering defined by comp; that is, for every iterator \(i\), in the range other than the first, the condition \(\text{comp}(*i, *(i - 1))\) will be false. Pointers and references to the moved elements of \(x\) now refer to those same elements but as members of \(*\text{this}\). Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into \(*\text{this}\), not into \(x\).

Remarks: Stable (17.6.5.7). If (&x != this) the range \([x.\text{begin}(), x.\text{end}())\) is empty after the merge. No elements are copied by this operation. The behavior is undefined if \(\text{this->get_allocator}() != x.\text{get_allocator}()\).

Complexity: At most \(\text{size}() + x.\text{size}() - 1\) applications of \(\text{comp}\) if (&x != this); otherwise, no applications of \(\text{comp}\) 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);

Requires: operator< (for the first version) or \(\text{comp}\) (for the second version) shall define a strict weak ordering (25.4).

Effects: Sorts the list according to the operator< or a \(\text{Compare}\) function object. Does not affect the validity of iterators and references.

Remarks: Stable (17.6.5.7).

Complexity: Approximately \(N \log(N)\) comparisons, where \(N == \text{size}()\).

23.3.5.6 list specialized algorithms

\[
\text{template <class T, class Allocator>} \begin{align*}
\text{void swap} & (\text{list}<T,\text{Allocator}>& x, \text{list}<T,\text{Allocator}>& y) \\
& \text{noexcept(noexcept(x.swap(y)))};
\end{align*}
\]

Effects:
\[
x.\text{swap}(y);
\]

23.3.6 Class template vector

23.3.6.1 Class template vector overview

A \texttt{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 Clause 23.2), of a sequence container, including most of the optional sequence container requirements (23.2.3), of an allocator-aware container (Table 99), and, for an element type other than bool, of a contiguous container (23.2.1). The exceptions are the push_front, pop_front, and emplace_front member functions, which are not provided. Descriptions are provided here only for operations on vector that are not described in one of these tables or for operations where there is additional semantic information.

namespace std {
    template <class T, class Allocator = allocator<T> >
    class vector {
    public:
        // types:
        typedef value_type& reference;
        typedef const value_type& const_reference;
        typedef implementation-defined iterator; // see 23.2
        typedef implementation-defined const_iterator; // see 23.2
        typedef implementation-defined size_type; // see 23.2
        typedef implementation-defined difference_type; // see 23.2
        typedef T value_type;
        typedef Allocator allocator_type;
        typedef typename allocator_traits<Allocator>::pointer pointer;
        typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
        typedef std::reverse_iterator<iterator> reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

        // 23.3.6.2, construct/copy/destroy:
        vector() noexcept : 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&, const Allocator&);
        vector(vector&&, const Allocator&);
        vector(initializer_list<T>, const Allocator& = Allocator());
        ~vector();
        vector& operator=(const vector& x);
        vector& operator=(vector&& x);
        noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
                  allocator_traits<Allocator>::is_always_equal::value);
        vector& operator=(initializer_list<T>);
        template <class InputIterator>
        void assign(InputIterator first, InputIterator last);
        void assign(size_type n, const T& u);
        void assign(initializer_list<T>);
        allocator_type get_allocator() const noexcept;

        // iterators:
        iterator begin() noexcept;
        const_iterator begin() const noexcept;
        iterator end() noexcept;
        const_iterator end() const noexcept;

        // 23.3.6.1
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;

// 23.3.6.3, capacity:
size_type size() const noexcept;
size_type max_size() const noexcept;
void resize(size_type sz);
void resize(size_type sz, const T& c);
size_type capacity() const noexcept;
bool empty() const noexcept;
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;

// 23.3.6.4, data access
T* data() noexcept;
const T* data() const noexcept;

// 23.3.6.5, modifiers:
template <class... Args> void 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&)
    noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
    allocator_traits<Allocator>::is_always_equal::value);
void clear() noexcept;
};
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);

// 23.3.6.6, specialized algorithms:
template <class T, class Allocator>
  void swap(vector<T,Allocator>& x, vector<T,Allocator>& y)
    noexcept(noexcept(x.swap(y)));

23.3.6.2 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.
23.3.6.3 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().

void shrink_to_fit();

Requires: T shall be MoveInsertable into *this.

Complexity: Linear in the size of the sequence.

Remarks: 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] If an exception is thrown other than by the move constructor of a non-CopyInsertable T there are no effects.

void swap(vector& x)

noexcept(algorithm_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(), equivalent to calling pop_back() size() - sz times. If size() < sz, 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(), equivalent to calling pop_back() size() - sz times. If size() < sz, 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.

reserve() uses Allocator::allocate() which may throw an appropriate exception.
23.3.6.4 vector data

T* data() noexcept;

const T* data() const noexcept;

1  Returns: A pointer such that [data(), data() + size()) is a valid range. For a non-empty vector, data() == &front().

2  Complexity: Constant time.

23.3.6.5 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);

iterator insert(const_iterator position, initializer_list<T>);

template <class... Args> void 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);

1  Remarks: Causes reallocation if the new size is greater than the old capacity. 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<T>::value 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.

2  Complexity: The complexity is linear in the number of elements inserted plus the distance to the end of the vector.

iterator erase(const_iterator position);

iterator erase(const_iterator first, const_iterator last);

3  Effects: Invalidates iterators and references at or after the point of the erase.

4  Complexity: The destructor of T is called the number of times equal to the number of the elements erased, but the move assignment operator of T is called the number of times equal to the number of elements in the vector after the erased elements.

5  Throws: Nothing unless an exception is thrown by the copy constructor, move constructor, assignment operator, or move assignment operator of T.

23.3.6.6 vector specialized algorithms

template <class T, class Allocator>

void swap(vector<T, Allocator>& x, vector<T, Allocator>& y)

noexcept(noexcept(x.swap(y)));

1  Effects:

x.swap(y);
23.3.7 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:
            typedef bool const_reference;
            typedef implementation-defined iterator;  // see 23.2
            typedef implementation-defined const_iterator; // see 23.2
            typedef implementation-defined size_type; // see 23.2
            typedef implementation-defined difference_type; // see 23.2
            typedef bool value_type;
            typedef Allocator allocator_type;
            typedef implementation-defined pointer;
            typedef implementation-defined const_pointer;
            typedef std::reverse_iterator<iterator> reverse_iterator;
            typedef std::reverse_iterator<const_iterator> const_reverse_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<bool,Allocator>& x);
            vector(vector<bool,Allocator>&& x);
            vector(const vector&, const Allocator&);
            vector(vector&&, const Allocator&);
            vector(initializer_list<bool>, const Allocator& = Allocator());
            ~vector();
            vector<bool,Allocator>& operator=(const vector<bool,Allocator>& x);
            vector<bool,Allocator>& operator=(vector<bool,Allocator>&& 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:
size_type size() const noexcept;
size_type max_size() const noexcept;
void resize(size_type sz, bool c = false);
size_type capacity() const noexcept;
bool empty() const noexcept;
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> void 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<bool, Allocator>&);
static void swap(reference x, reference y) noexcept;
void flip() noexcept;  // flips all bits
void clear() noexcept;
};
Unless described below, all operations have the same requirements and semantics as the primary `vector` template, except that operations dealing with the `bool` value type map to bit values in the container storage and `allocator_traits::construct` (20.7.8.2) is not used to construct these values.

There is no requirement that the data be stored as a contiguous allocation of `bool` values. A space-optimized representation of bits is recommended instead.

`reference` is a class that simulates the behavior of references of a single bit in `vector<bool>`. The conversion operator 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.

```cpp
void flip() noexcept;  
Effects: Replaces each element in the container with its complement.
```

```cpp
static void swap(reference x, reference y) noexcept;
Effects: exchanges the contents of x and y as if by
bool b = x;
   x = y;
   y = b;
```

```cpp
template <class Allocator> struct hash<vector<bool, Allocator>>;
The template specialization shall meet the requirements of class template `hash` (20.9.13).
```

23.4 Associative containers [associative]

23.4.1 In general [associative.general]
The header `<map>` defines the class templates `map` and `multimap`; the header `<set>` defines the class templates `set` and `multiset`.

23.4.2 Header `<map>` synopsis [associative.map.syn]

```cpp
#include <initializer_list>
namespace std {

   template <class Key, class T, class Compare = less<Key>,
            class Allocator = allocator<pair<const Key, T> > >
   class map;
   template <class Key, class T, class Compare, class Allocator>
   bool operator==(const map<Key,T,Compare,Allocator>& x,
                   const map<Key,T,Compare,Allocator>& y);
   template <class Key, class T, class Compare, class Allocator>
   bool operator< (const map<Key,T,Compare,Allocator>& x,
                  const map<Key,T,Compare,Allocator>& y);
   template <class Key, class T, class Compare, class Allocator>
   bool operator!=(const map<Key,T,Compare,Allocator>& x,
                  const map<Key,T,Compare,Allocator>& y);
   template <class Key, class T, class Compare, class Allocator>
   bool operator> (const map<Key,T,Compare,Allocator>& x,
                  const map<Key,T,Compare,Allocator>& y);
   template <class Key, class T, class Compare, class Allocator>
   bool operator>=(const map<Key,T,Compare,Allocator>& x,
                 const map<Key,T,Compare,Allocator>& y);
   template <class Key, class T, class Compare, class Allocator>
   bool operator<=(const map<Key,T,Compare,Allocator>& x,
                 const map<Key,T,Compare,Allocator>& y);
}
```

§ 23.4.2
bool operator<=(const map<Key,T,Compare,Allocator>& x, const map<Key,T,Compare,Allocator>& y);

namespace std {

template <class Key, class T, class Compare, class Allocator>
void swap(map<Key,T,Compare,Allocator>& x, map<Key,T,Compare,Allocator>& y)
noexcept(noexcept(x.swap(y)));

template <class Key, class T, class Compare = less<Key>, class Allocator = allocator<pair<const Key, T>>>
class set;

template <class Key, class T, class Compare, class Allocator>
void swap(set<Key,T,Compare,Allocator>& x, set<Key,T,Compare,Allocator>& y)
noexcept(noexcept(x.swap(y)));

23.4.3 Header <set> synopsis

#include <initializer_list>

namespace std {

template <class Key, class Compare = less<Key>, class Allocator = allocator<Key>>
class set;

template <class Key, class Compare, class Allocator>
bool operator==(const set<Key,Compare,Allocator>& x, const set<Key,Compare,Allocator>& y);

template <class Key, class Compare, class Allocator>
bool operator<=(const set<Key,Compare,Allocator>& x, const set<Key,Compare,Allocator>& y);

template <class Key, class Compare, class Allocator>
bool operator<(const set<Key,Compare,Allocator>& x, const set<Key,Compare,Allocator>& y);

template <class Key, class Compare, class Allocator>
bool operator!=(const set<Key,Compare,Allocator>& x, const set<Key,Compare,Allocator>& y);

template <class Key, class Compare, class Allocator>
bool operator>(const set<Key,Compare,Allocator>& x, const set<Key,Compare,Allocator>& y);

template <class Key, class Compare, class Allocator>
bool operator>=(const set<Key,Compare,Allocator>& x, const set<Key,Compare,Allocator>& y);

§ 23.4.3 806
bool operator>=(const set<Key, Compare, Allocator>& x, const set<Key, Compare, Allocator>& y);

template <class Key, class Compare, class Allocator>
bool operator<=(const set<Key, Compare, Allocator>& x, const set<Key, Compare, Allocator>& y);

template <class Key, class Compare, class Allocator>
void swap(set<Key, Compare, Allocator>& x, set<Key, Compare, Allocator>& y)
noexcept(noexcept(x.swap(y)));

template <class Key, class Compare = 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>
void swap(multiset<Key, Compare, Allocator>& x, multiset<Key, Compare, Allocator>& y)
noexcept(noexcept(x.swap(y)));}

23.4.4 Class template map

23.4.4.1 Class template map overview

A map is an associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of values of another type T based on the keys. The map class supports bidirectional iterators.

A map satisfies all of the requirements of a container, of a reversible container (23.2), of an associative container (23.2.4), and of an allocator-aware container (Table 99). A map also provides most operations described in (23.2.4) for unique keys. This means that a map supports the a_uniq operations in (23.2.4) but not the a_eq operations. For a map<Key,T> the key_type is Key and the value_type is pair<const Key,T>. Descriptions are provided here only for operations on map that are not described in one of those tables or for operations where there is additional semantic information.

namespace std {
  template <class Key, class T, class Compare = less<Key>,
  class Allocator = allocator<pair<const Key, T> > >
  class map {
  public:

typedef Key key_type;
typedef T mapped_type;
typedef pair<const Key, T> value_type;
typedef Compare key_compare;
typedef Allocator allocator_type;
typedef value_type& reference;
typedef const value_type& const_reference;
typedef implementation-defined iterator; // see 23.2
typedef implementation-defined const_iterator; // see 23.2
typedef implementation-defined size_type; // see 23.2
typedef implementation-defined difference_type; // see 23.2
typedef typename allocator_traits<Allocator>::pointer pointer;
typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

class value_compare {
friend class map;
protected:
  Compare comp;
  value_compare(Compare c) : comp(c) {}}
public:
  typedef bool result_type;
  typedef value_type first_argument_type;
  typedef value_type second_argument_type;
  bool operator()(const value_type& x, const value_type& y) const {
    return comp(x.first, y.first);
  }
};

// 23.4.4.2, 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(map&, const Allocator&);
map(map&, const Allocator&);
map(initializer_list<value_type>, 
  const Compare& = Compare(), const Allocator& = Allocator());
template <class InputIterator>
map(InputIterator first, InputIterator last, const Allocator& a) :
  map(first, last, Compare(), a) { }
map(initializer_list<value_type> il, const Allocator& a) :
  map(il, Compare(), a) { }
~map();
map& operator=(const map& x);
map& operator=(map& x)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &

§ 23.4.4.1
is_nothrow_move_assignable<Compare>::value);
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 rbegin() const noexcept;
const_reverse_iterator rend() const noexcept;

// capacity:
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// 23.4.4.3, element access:
T& operator[](const key_type& x);
T& operator[](key_type&& x);
T& at(const key_type& x);
const T& at(const key_type& x) const;

// 23.4.4.4, modifiers:
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
template <class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
pair<iterator, bool> insert(const value_type& x);
pair<iterator, bool> insert(value_type&& x);
template <class P> pair<iterator, bool> insert(P&& x);

void insert(InputIterator first, InputIterator last);  
void insert(initializer_list<value_type>);

template <class... Args>
pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template <class... Args>
pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template <class... Args>
iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);
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(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 &&
    noexcept(swap(declval<Compare&>(), declval<Compare&>())));
void clear() noexcept;

// 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;
};
bool operator!=(const map<Key,T,Compare,Allocator>& x,  
    const map<Key,T,Compare,Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator> (const map<Key,T,Compare,Allocator>& x,  
    const map<Key,T,Compare,Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator>=(const map<Key,T,Compare,Allocator>& x,  
    const map<Key,T,Compare,Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator<=(const map<Key,T,Compare,Allocator>& x,  
    const map<Key,T,Compare,Allocator>& y);

// 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)));
Complexity: Logarithmic.

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.

23.4.4.4 map modifiers

template <class P> pair<iterator, bool> insert(P&& x);
template <class P> iterator insert(const_iterator position, P&& x);
template <class InputIterator>
void insert(InputIterator first, InputIterator last);

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 std::is_constructible<value_type, P&&>::value is true.

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);
template <class... Args> iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);

Effects: If the key k already exists in the map, there is no effect. Otherwise, inserts an element into the map. In the first and third forms, the element is constructed from the arguments as value_type(k, std::forward<Args>(args)...). In the second and fourth forms, the element is constructed from the arguments as value_type(std::move(k), std::forward<Args>(args)...). In the first two overloads, the bool component of the returned pair is true if and only if the insertion took place. The returned iterator points to the element that was inserted or updated.

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> 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);

Effects: If the key k does not exist in the map, inserts an element into the map. In the first and third forms, the element is constructed from the arguments as value_type(k, std::forward<Args>(args)...). In the second and fourth forms, the element is constructed from the arguments as value_type(std::move(k), std::forward<Args>(args)...). If the key already exists, std::forward<M>(obj) is assigned to the mapped_type corresponding to the key. In the first two overloads, the bool component of the returned value is true if and only if the insertion took place. The returned iterator points to the element that was inserted or updated.

Complexity: The same as emplace and emplace_hint, respectively.

23.4.4.5 map 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)));
Effects:

\[ x . \text{swap}(y); \]

23.4.5 Class template multimap

23.4.5.1 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 (23.2), of an associative container (23.2.4), and of an allocator-aware container (Table 99). A `multimap` also provides most operations described in (23.2.4) for equal keys. This means that a `multimap` supports the `a_eq` operations 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:
            typedef Key key_type;
            typedef T mapped_type;
            typedef pair<const Key, T> value_type;
            typedef Compare key_compare;
            typedef Allocator allocator_type;
            typedef value_type& reference;
            typedef const value_type& const_reference;
            typedef implementation-defined iterator;
            typedef implementation-defined const_iterator;
            typedef implementation-defined size_type;  // see 23.2
            typedef implementation-defined difference_type;  // see 23.2
            typedef typename allocator_traits<Allocator>::pointer pointer;
            typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
            typedef std::reverse_iterator<iterator> reverse_iterator;
            typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

            class value_compare {
                friend class multimap;
                protected:
                    Compare comp;
                value_compare(Compare c) : comp(c) { }
            public:
                typedef bool result_type;
                typedef value_type first_argument_type;
                typedef value_type second_argument_type;
                bool operator()(const value_type& x, const value_type& y) const {
                    return comp(x.first, y.first);
                }
            };

            // construct/copy/destroy:
            multimap() : multimap(Compare()) { }
    };
```
explicit multimap(const Compare& comp,  
        const Allocator& = Allocator());

template <class InputIterator>
multimap(InputIterator first, InputIterator last,  
        const Compare& comp = Compare(),  
        const Allocator& = Allocator());
multimap(const multimap& x);
multimap(multimap&& x);
explicit multimap(const Allocator&);
multimap(const multimap&, const Allocator&);
multimap(multimap&, const Allocator&);
multimap(initializer_list<value_type>,  
        const Compare& = Compare(),  
        const Allocator& = Allocator());

// iterators:
iterator begin() const noexcept;
const_iterator begin() noexcept;
iterator end() const noexcept;
const_iterator end() noexcept;
reverse_iterator rbegin() const noexcept;
const_reverse_iterator rbegin() noexcept;
reverse_iterator rend() const 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:
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);
template <class P> iterator insert(P&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);

#define N4296

template <class P> iterator insert(const_iterator position, P&& x);
template <class InputIterator>
    void insert(InputIterator first, InputIterator last);

iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);

void swap(multimap&);

#endif

noexcept(allocator_traits<Allocator>::is_always_equal::value &
    noexcept(swap(declval<Compare&>(),declval<Compare&>())))

void clear() noexcept;

// 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;
};
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);

// specialized algorithms:
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)));
}

23.4.5.2 multimap constructors

explicit multimap(const Compare& comp,
    const Allocator& = Allocator());

1 Effects: Constructs an empty multimap using the specified comparison object and allocator.

2 Complexity: Constant.

template <class InputIterator>
multimap(InputIterator first, InputIterator last,
    const Compare& comp = Compare(),
    const Allocator& = Allocator());

3 Requires: If the iterator’s indirection operator returns an lvalue or a const rvalue pair<key_type,
mapped_type>, then both key_type and mapped_type shall be CopyInsertable into *this.

4 Effects: Constructs an empty multimap using the specified comparison object and allocator, and inserts elements from the range [first,last).

5 Complexity: Linear in \(N\) if the range [first,last) is already sorted using comp and otherwise \(N \log N\), where \(N\) is last - first.

23.4.5.3 multimap modifiers

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 std::is_constructible<value_type, P&&>::value is true.

23.4.5.4 multimap specialized algorithms

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)));

1 Effects:
23.4.6 Class template set

23.4.6.1 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 (23.2), of an associative container (23.2.4), and of an allocator-aware container (Table 99). A set also provides most operations described in (23.2.4) for unique keys. This means that a set supports the a_unique operations in (23.2.4) but not the a_eq operations. For a set<Key> both the key_type and value_type are Key. Descriptions are provided here only for operations on set that are not described in one of these tables and for operations where there is additional semantic information.

```cpp
namespace std {
    template <class Key, class Compare = less<Key>,
              class Allocator = allocator<Key> >
    class set {
    public:
        // types:
        typedef Key key_type;
        typedef Key value_type;
        typedef Compare key_compare;
        typedef Compare value_compare;
        typedef Allocator allocator_type;
        typedef value_type& reference;
        typedef const value_type& const_reference;
        typedef typename allocator_traits<Allocator>::pointer pointer;
        typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
        typedef std::reverse_iterator<iterator> reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

        // 23.4.6.2, construct/copy/destroy:
        set() : set(Compare()) { } // See 23.2
        explicit set(const Compare& comp, // See 23.2
                     const Allocator& = Allocator());
        template <class InputIterator>
        set(InputIterator first, InputIterator last, // See 23.2
             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<Compare>::value);
set& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators:
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;

// capacity:
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>);

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 &&
  noexcept swap(declval<Compare&>(), declval<Compare&>());
void clear() noexcept;

// 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);
template <class K> pair<const_iterator, const_iterator> equal_range(const K& x) const;

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);

// 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)));

23.4.6.2  set constructors, copy, and assignment  [set.cons]

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).

Requires: If the iterator’s indirection operator returns an lvalue or a non-const rvalue, then Key shall be CopyInsertable into *this.

Complexity: Linear in \( N \) if the range \([\text{first}, \text{last})\) is already sorted using \( \text{comp} \) and otherwise \( N \log N \), where \( N = \text{last} - \text{first} \).

23.4.6.3 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: 

x.swap(y);

23.4.7 Class template multiset

23.4.7.1 Class template multiset overview

A multiset is an associative container that supports equivalent keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of the keys themselves. The multiset class supports bidirectional iterators.

A multiset satisfies all of the requirements of a container, of a reversible container (23.2), of an associative container (23.2.4), and of an allocator-aware container (Table 99). multiset also provides most operations described in (23.2.4) for duplicate keys. This means that a multiset supports the \texttt{a\_eq} operations in (23.2.4) but not the \texttt{a\_uniq} operations. For a multiset<\texttt{Key}> both the \texttt{key\_type} and \texttt{value\_type} are \texttt{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:
            typedef Key key_type;
            typedef Key value_type;
            typedef Compare key_compare;
            typedef Compare value_compare;
            typedef Allocator allocator_type;
            typedef value_type& reference;
            typedef const value_type& const_reference;
            typedef implementation-defined iterator; // see 23.2
            typedef implementation-defined const_iterator; // see 23.2
            typedef implementation-defined size_type; // see 23.2
            typedef implementation-defined difference_type; // see 23.2
```
typedef typename allocator_traits<Allocator>::pointer pointer;
typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

// construct/copy/destroy:
multiset() : multiset(Compare()) { }
explicit multiset(const Compare& comp,
   const Allocator& = Allocator());
template <class InputIterator>
multiset(InputIterator first, InputIterator last,
   const Compare& comp = Compare(),
   const Allocator& = Allocator());
multiset(const multiset& x);
multiset(multiset& x);
explicit multiset(const Allocator&);
multiset(const multiset&, const Allocator&);
multiset(multiset&, const Allocator&);
multiset(initializer_list<value_type>-
   const Compare& = Compare(),
   const Allocator& = Allocator());
template <class InputIterator>
multiset(InputIterator first, InputIterator last, const Allocator& a)
   : multiset(first, last, Compare(), a) { }
multiset(initializer_list<value_type> il, const Allocator& a)
   : multiset(il, Compare(), a) { }
~multiset();
multiset& operator=(const multiset& x);
multiset& operator=(multiset&& x)
   noexcept(allocator_traits<Allocator>::is_always_equal::value &&
   is_nothrow_move_assignable<Compare>::value);
allocation_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:
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

§ 23.4.7.1
// 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>);

iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);
void swap(multiset&)
    noexcept(algorithm_traits<Allocate>::is_always_equal::value &&
             noexcept(swap(declval<Compare&>(), declval<Compare&>()));
void clear() noexcept;

// 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);
template <class K> pair<const_iterator, const_iterator> equal_range(const K& x) const;

// relational operators:
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);

// 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)));

23.4.7.2 multiset constructors
[multiset.cons]

explicit multiset(const Compare& comp,
    const Allocator& = Allocator());

1 Effects: Constructs an empty set 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 Requires: If the iterator’s indirection operator returns an lvalue or a const rvalue, then Key shall be
    CopyInsertable into *this.

4 Effects: Constructs an empty multiset using the specified comparison object and allocator, and inserts
    elements from the range [first,last).

5 Complexity: Linear in N if the range [first,last) is already sorted using comp and otherwise N log N,
    where N is last - first.

23.4.7.3 multiset specialized algorithms
[multiset.special]

template <class Key, class Compare, class Allocator>
void swap(multiset<Key,Compare,Allocator>& x,
    multiset<Key,Compare,Allocator>& y)
noexcept(noexcept(x.swap(y)));

1 Effects:

   x.swap(y);

23.5 Unordered associative containers
[unord]

23.5.1 In general
[unord.general]

1 The header <unordered_map> defines the class templates unordered_map and unordered_multimap; the
    header <unordered_set> defines the class templates unordered_set and unordered_multiset.

§ 23.5.1
23.5.2 Header <unordered_map> synopsis

#include <initializer_list>

namespace std {

// 23.5.4, class template unordered_map:
template <class Key,
    class T,
    class Hash = hash<Key>,
    class Pred = std::equal_to<Key>,
    class Alloc = std::allocator<std::pair<const Key, T> > >
class unordered_map;

// 23.5.5, class template unordered_multimap:
template <class Key,
    class T,
    class Hash = hash<Key>,
    class Pred = std::equal_to<Key>,
    class Alloc = std::allocator<std::pair<const Key, T> > >
class unordered_multimap;

template <class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
    unordered_map<Key, T, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y)));

template <class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
    unordered_multimap<Key, T, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y)));

template <class Key, class T, class 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 std

23.5.3 Header <unordered_set> synopsis

#include <initializer_list>

namespace std {

// 23.5.6, class template unordered_set:
template <class Key,
    class Hash = hash<Key>,
    class Pred = std::equal_to<Key>,
    class Alloc = std::allocator<std::set<Key> > >
class unordered_set;

} // namespace std

§ 23.5.3
class Alloc = std::allocator<Key> >
class unordered_set;

// 23.5.7, class template unordered_multiset:
template <class Key, class Hash = hash<Key>,
class Pred = std::equal_to<Key>,
class Alloc = std::allocator<Key> >
class unordered_multiset;

template <class Key, class Hash, class Pred, class Alloc>
void swap(unordered_set<Key, Hash, Pred, Alloc>& x,
          unordered_set<Key, Hash, Pred, Alloc>& y)
  noexcept(noexcept(x.swap(y)));

template <class Key, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Key, Hash, Pred, Alloc>& x,
          unordered_multiset<Key, Hash, Pred, Alloc>& y)
  noexcept(noexcept(x.swap(y)));

template <class 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);  

} // namespace std

23.5.4 Class template unordered_map

23.5.4.1 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 99). 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 std::pair<const Key, T>.

This section only describes operations on unordered_map that are not described in one of the requirement tables, or for which there is additional semantic information.

namespace std {
  template <class Key,
            class T,
            class Hash = hash<Key>,
            class Pred = std::equal_to<Key>,
            class Allocator = std::allocator<
              std::pair<const Key, T> > >

§ 23.5.4.1 825
class unordered_map
{
public:
    // types
    typedef Key key_type;
    typedef std::pair<const Key, T> value_type;
    typedef T mapped_type;
    typedef Hash hasher;
    typedef Pred key_equal;
    typedef Allocator allocator_type;
    typedef typename allocator_traits<Allocator>::pointer pointer;
    typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef implementation-defined size_type;
    typedef implementation-defined difference_type;
    typedef implementation-defined iterator;
    typedef implementation-defined const_iterator;
    typedef implementation-defined local_iterator;
    typedef implementation-defined const_local_iterator;

    // construct/destroy/copy
    unordered_map();
    explicit unordered_map(size_type n, const hasher& hf = hasher(),
        const key_equal& eql = key_equal(),
        const allocator_type& a = allocator_type());
    template <class InputIterator>
    unordered_map(InputIterator f, InputIterator l,
        size_type n = see below,
        const hasher& hf = hasher(),
        const key_equal& eql = key_equal(),
        const allocator_type& a = allocator_type());
    unordered_map(const unordered_map&);
    unordered_map(unordered_map&&);
    explicit unordered_map(const Allocator&);
    unordered_map(const unordered_map&, const Allocator&);
    unordered_map(unordered_map&&, const Allocator&);
    unordered_map(initializer_list<value_type> il,
        size_type n = see below,
        const hasher& hf = hasher(),
        const key_equal& eql = key_equal(),
        const allocator_type& a = allocator_type());
    unordered_map(size_type n, const allocator_type& a)
        : unordered_map(n, hasher(), key_equal(), a) { }
    unordered_map(size_type n, const 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<Hash>::value &&  
is_nothrow_move_assignable<Pred>::value);  
unordered_map& operator=(initializer_list<value_type>);  
allocator_type get_allocator() const noexcept;

// size and capacity  
bool empty() const noexcept;  
size_type size() const noexcept;  
size_type max_size() 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;

// 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>);  

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(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear() noexcept;

void swap(unordered_map&)
  noexcept((allocator_traits<Allocator>::is_always_equal::value &&
  noexcept(swap(declval<Hash&>(),declval<Hash&>()))) &&
  noexcept(swap(declval<Pred&>(),declval<Pred&>())));

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
std::pair<iterator, iterator> equal_range(const key_type& k);
std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

mapped_type& operator[](const key_type& k);
mapped_type& operator[](key_type&& k);
mapped_type& at(const key_type& k);
const mapped_type& at(const key_type& k) const;

// bucket interface
size_type bucket_count() const noexcept;
size_type max_bucket_count() const noexcept;
size_type bucket_size(size_type n) const;
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 Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
  unordered_map<Key, T, Hash, Pred, Alloc>& y)
  noexcept(noexcept(x.swap(y)));

template <class Key, class T, class Hash, class Pred, class Alloc>
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);
}

23.5.4.2 unordered_map constructors

unordered_map() : unordered_map(size_type(see below)) {} 
explicit unordered_map(size_type n,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());

   Effects: Constructs an empty unordered_map using the specified hash function, key equality function, and allocator, and using at least \( n \) buckets. For the default constructor, the number of buckets is implementation-defined. \( \text{max}_\text{load}_\text{factor}() \) returns 1.0.

   Complexity: Constant.

unordered_map(InputIterator f, InputIterator l,
    size_type n = see below,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
unordered_map(initializer_list<value_type> il,
    size_type n = see below,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());

   Effects: Constructs an empty unordered_map using the specified hash function, key equality function, 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. \( \text{max}_\text{load}_\text{factor}() \) returns 1.0.

   Complexity: Average case linear, worst case quadratic.

23.5.4.3 unordered_map element access

mapped_type& operator[](const key_type& k);
mapped_type& operator[](key_type&& k);

   Requires: \( \text{mapped}_\text{type} \) shall be DefaultInsertable into *this. For the first operator, key_type shall be CopyInsertable into *this. For the second operator, key_type shall be MoveConstructible.

   Effects: If the unordered_map does not already contain an element whose key is equivalent to \( k \), the first operator inserts the value value_type(k, mapped_type()) and the second operator inserts the value value_type(std::move(k), mapped_type()).

   Returns: A reference to x.second, where x is the (unique) element whose key is equivalent to \( k \).

   Complexity: Average case \( O(1) \), worst case \( O(\text{size}()) \).

mapped_type& at(const key_type& k);
const mapped_type& at(const key_type& k) const;

   Returns: A reference to x.second, where x is the (unique) element whose key is equivalent to \( k \).

   Throws: An exception object of type out_of_range if no such element is present.
23.5.4.4 unordered_map modifiers

```
template <class P>
pair<iterator, bool> insert(P&& obj);
```

1. **Effects:** Equivalent to return `emplace(std::forward<P>(obj))`.

2. **Remarks:** This signature shall not participate in overload resolution unless `std::is_constructible<value_type, P&&>::value` is true.

```
template <class P>
iterator insert(const_iterator hint, P&& obj);
```

3. **Effects:** Equivalent to return `emplace_hint(hint, std::forward<P>(obj))`.

4. **Remarks:** This signature shall not participate in overload resolution unless `std::is_constructible<value_type, P&&>::value` is true.

```
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);
```

5. **Effects:** If the key `k` already exists in the map, there is no effect. Otherwise, inserts an element into the map. In the first and third forms, the element is constructed from the arguments as `value_type(k, std::forward<Args>(args)...)`. In the second and fourth forms, the element is constructed from the arguments as `value_type(std::move(k), std::forward<Args>(args)...)`. In the first two overloads, the bool component of the returned pair is `true` if and only if the insertion took place. The returned iterator points to the element of the map whose key is equivalent to `k`.

6. **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> 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);
```

7. **Effects:** If the key `k` does not exist in the map, inserts an element into the map. In the first and third forms, the element is constructed from the arguments as `value_type(k, std::forward<Args>(args)...)`. In the second and fourth forms, the element is constructed from the arguments as `value_type(std::move(k), std::forward<Args>(args)...)`. If the key already exists, `std::forward<M>(obj)` is assigned to the `mapped_type` corresponding to the key. In the first two overloads, the bool component of the returned value is `true` if and only if the insertion took place. The returned iterator points to the element that was inserted or updated.

8. **Complexity:** The same as `emplace` and `emplace_hint`, respectively.

23.5.4.5 unordered_map swap

```
template <class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
          unordered_map<Key, T, Hash, Pred, Alloc>& y)
noexcept(noexcept(x.swap(y)));
```

1. **Effects:** `x.swap(y)`.
23.5.5 Class template unordered_multimap

23.5.5.1 Class template unordered_multimap overview

1 An unordered_multimap is an unordered associative container that supports equivalent keys (an instance of unordered_multimap may contain multiple copies of each key value) and that associates values of another type mapped_type with the keys. The unordered_multimap class supports forward iterators.

2 An unordered_multimap satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 99). 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 std::pair<const Key, T>.

3 This section only describes operations on unordered_multimap that are not described in one of the requirement tables, or for which there is additional semantic information.

namespace std {
    template <class Key,
              class T,
              class Hash = hash<Key>,
              class Pred = std::equal_to<Key>,
              class Allocator = std::allocator<std::pair<const Key, T> >
    >
    class unordered_multimap
    {
        public:
        // types
        typedef Key key_type;
        typedef std::pair<const Key, T> value_type;
        typedef T mapped_type;
        typedef Hash hasher;
        typedef Pred key_equal;
        typedef Allocator allocator_type;
        typedef typename allocator_traits<Allocator>::pointer pointer;
        typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
        typedef value_type& reference;
        typedef const value_type& const_reference;
        typedef implementation-defined size_type;
        typedef implementation-defined difference_type;
        typedef implementation-defined iterator;
        typedef implementation-defined const_iterator;
        typedef implementation-defined local_iterator;
        typedef implementation-defined const_local_iterator;

        // construct/destroy/copy
        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());
    } // class unordered_multimap
} // namespace std

§ 23.5.5.1
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, hf, 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(allocator_traits<Allocator>::is_always_equal::value &&
is_nothrow_move_assignable<Hash>::value &&
is_nothrow_move_assignable<Pred>::value);
unordered_multimap& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// size and capacity
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() 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;

// 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>);

iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear() noexcept;

void swap(unordered_multimap &)
    noexcept(algorithm_traits<Allocator>::is_always_equal::value &&
             noexcept(swap(declval<Hash &>(), declval<Hash &>()))) &&
             noexcept(swap(declval<Pred &>(), declval<Pred &>()));

// observers
    hasher hash_function() const;
    key_equal key_eq() const;

// lookup
    iterator find(const key_type& k);
    const_iterator find(const key_type& k) const;
    size_type count(const key_type& k) const;
    std::pair<iterator, iterator> equal_range(const key_type& k);
    std::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_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 Key, class T, class Hash, class Pred, class Alloc>
    void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
              unordered_multimap<Key, T, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y)));

template <class Key, class T, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_multimap<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_multimap<Key, T, Hash, Pred, Alloc>& b);

template <class Key, class T, class Hash, class Pred, class Alloc>
    void operator=(const unordered_multimap<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_multimap<Key, T, Hash, Pred, Alloc>& b);
bool operator!=(const unordered_multimap<Key, T, Hash, Pred, Alloc>& a, 
    const unordered_multimap<Key, T, Hash, Pred, Alloc>& b);
}

### 23.5.5.2 unordered_multimap constructors

 unordered_multimap() : unordered_multimap(size_type(see below)) { }
 explicit unordered_multimap(size_type n, 
    const hasher& hf = hasher(), 
    const key_equal& eql = key_equal(), 
    const allocator_type& a = allocator_type());

1. **Effects:** Constructs an empty unordered_multimap using the specified hash function, key equality function, 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.

2. **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(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());
```

3. **Effects:** Constructs an empty unordered_multimap using the specified hash function, key equality function, 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.

4. **Complexity:** Average case linear, worst case quadratic.

### 23.5.5.3 unordered_multimap modifiers

```
template <class P>
iterator insert(P&& obj);
```

1. **Effects:** Equivalent to return emplace(std::forward<P>(obj)).

2. **Remarks:** This signature shall not participate in overload resolution unless std::is_constructible<value_type, P&>::value is true.

```
template <class P>
iterator insert(const_iterator hint, P&& obj);
```

3. **Effects:** Equivalent to return emplace_hint(hint, std::forward<P>(obj)).

4. **Remarks:** This signature shall not participate in overload resolution unless std::is_constructible<value_type, P&>::value is true.

### 23.5.5.4 unordered_multimap 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)));```
Effects: \texttt{x.swap(y)}.

23.5.6 Class template \texttt{unordered_set} \hfill [unord.set]

23.5.6.1 Class template \texttt{unordered_set} overview \hfill [unord.set.overview]

An \texttt{unordered_set} is an unordered associative container that supports unique keys (an \texttt{unordered_set} contains at most one of each key value) and in which the elements' keys are the elements themselves. The \texttt{unordered_set} class supports forward iterators.

An \texttt{unordered_set} satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 99). It provides the operations described in the preceding requirements table for unique keys; that is, an \texttt{unordered_set} supports the \texttt{a uniq} operations in that table, not the \texttt{a eq} operations. For an \texttt{unordered_set<Key>} the \texttt{key type} and the value type are both \texttt{Key}. The \texttt{iterator} and \texttt{const_iterator} types are both const iterator types. It is unspecified whether they are the same type.

This section only describes operations on \texttt{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 = std::equal_to<Key>,  
            class Allocator = std::allocator<Key> >  
    class unordered_set  
    {  
        public:  
            // types  
            typedef Key key_type;  
            typedef Key value_type;  
            typedef Hash hasher;  
            typedef Pred key_equal;  
            typedef Allocator allocator_type;  
            typedef typename Allocator_traits<Allocator>::pointer pointer;  
            typedef typename Allocator_traits<Allocator>::const_pointer const_pointer;  
            typedef value_type& reference;  
            typedef const value_type& const_reference;  
            typedef implementation-defined size_type;  
            typedef implementation-defined difference_type;  

            typedef implementation-defined iterator;  
            typedef implementation-defined const_iterator;  
            typedef implementation-defined local_iterator;  
            typedef implementation-defined const_local_iterator;  

            // construct/destroy/copy  
            unordered_set();  
            explicit unordered_set(size_type n,  
                const hasher& hf = hasher(),  
                const key_equal& eq = 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& eq = key_equal(),  
                const allocator_type& a = allocator_type());  
}
```

§ 23.5.6.1
unordered_set(const unordered_set&);
unordered_set(unordered_set&);
explicit unordered_set(const Allocator&);
unordered_set(const unordered_set&, const Allocator&);
unordered_set(unordered_set&, const Allocator&);
unordered_set(initializer_list<value_type> il,
  size_type n = see below,
  const hasher& hf = hasher(),
  const key_equal& eql = key_equal(),
  const allocator_type& a = allocator_type());
unordered_set(size_type n, const Allocator& a) :
  unordered_set(n, key_equal(), a) {} 
unordered_set(size_type n, const hasher& hf, const Allocator& a) :
  unordered_set(n, hf, key_equal(), a) {} 
template <class InputIterator>
unordered_set(InputIterator f, InputIterator l, size_type n, const Allocator& a) :
  unordered_set(f, l, n, key_equal(), a) {} 

template <class InputIterator>
unordered_set(InputIterator f, InputIterator l, size_type n, const hasher& hf,
  const Allocator& a) :
  unordered_set(f, l, n, hf, key_equal(), a) {} 
unordered_set(initializer_list<value_type> il, size_type n, const Allocator& a) :
  unordered_set(il, n, hasher(), key_equal(), a) {} 
unordered_set(initializer_list<value_type> il, size_type n, const hasher& hf,
  const Allocator& a) :
  unordered_set(il, n, hf, key_equal(), a) {} 
~unordered_set();
unordered_set& operator=(const unordered_set&);
unordered_set& operator=(unordered_set&&)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
    is_nothrow_move_assignable<Hash>::value &&
    is_nothrow_move_assignable<Pred>::value);
unordered_set& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// size and capacity
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() 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;

// 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>);

iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear() noexcept;

void swap(unordered_set&);
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
          noexcept(swap(declval<Hash&>(),declval<Hash&>())) &&
          noexcept(swap(declval<Pred&>(),declval<Pred&>())));

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator   find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type   count(const key_type& k) const;
std::pair<iterator, iterator>   equal_range(const key_type& k);
std::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);
local_iterator end(size_type n);
const_local_iterator begin(size_type n) const;
const_local_iterator end(size_type n) const;
const_local_iterator begin(size_type n) const;
const_local_iterator end(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 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>
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);
23.5.6.2 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 function, and allocator, and using at least \( n \) buckets. For the default constructor, the number of buckets is implementation-defined. \( \text{max\_load\_factor()} \) returns 1.0.

Complexity: Constant.

template <class InputIterator>
unordered_set(InputIterator f, InputIterator l,
    size_type n = see below,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
unordered_set(initializer_list<value_type> il,
    size_type n = see below,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());

Effects: Constructs an empty unordered_set using the specified hash function, key equality function, 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. \( \text{max\_load\_factor()} \) returns 1.0.

Complexity: Average case linear, worst case quadratic.

23.5.6.3 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)));

Effects: \( x.\text{swap}(y) \).

23.5.7 Class template unordered_multiset

23.5.7.1 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 99). 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 const iterator types. It is unspecified whether they are the same type.
This section 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 = std::equal_to<Key>,  
              class Allocator = std::allocator<Key> >
    class unordered_multiset  
    {  
      public:  
        // types  
        typedef Key key_type;  
        typedef Key value_type;  
        typedef Hash hasher;  
        typedef Pred key_equal;  
        typedef Allocator allocator_type;  
        typedef typename allocator_traits<Allocator>::pointer pointer;  
        typedef typename allocator_traits<Allocator>::const_pointer const_pointer;  
        typedef value_type& reference;  
        typedef const value_type& const_reference;  
        typedef implementation-defined size_type;  
        typedef implementation-defined difference_type;  
        typedef implementation-defined iterator;  
        typedef implementation-defined const_iterator;  
        typedef implementation-defined local_iterator;  
        typedef implementation-defined const_local_iterator;  
        // construct/destroy/copy
        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>
```

§ 23.5.7.1
unordered_multiset(InputIterator f, InputIterator l, size_type n, const allocator_type& a) : unordered_multiset(f, l, n, hasher(), key_equal(), a) { }

template <class InputIterator>
unordered_multiset(InputIterator f, InputIterator l, size_type n, const hasher& hf, const allocator_type& a) : unordered_multiset(f, l, n, hf, key_equal(), a) { }

unordered_multiset(initializer_list<value_type> il, size_type n, const allocator_type& a) : unordered_multiset(il, n, hasher(), key_equal(), a) { }

unordered_multiset(initializer_list<value_type> il, size_type n, const hasher& hf, const allocator_type& a) : unordered_multiset(il, n, hf, key_equal(), a) { }

-unordered_multiset();

unordered_multiset& operator=(const unordered_multiset&);
unordered_multiset& operator=(unordered_multiset&&) noexcept(allocator_traits<Allocator>::is_always_equal::value &&
is_nothrow_move_assignable<Hash>::value &&
is_nothrow_move_assignable<Pred>::value);

unordered_multiset& operator=(initializer_list<value_type>);

allocator_type get_allocator() const noexcept;

// size and capacity
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() 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;

// 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>);

iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear() noexcept;

void swap(unordered_multiset&)
  noexcept(allocation_traits<Allocator>::is_always_equal::value &
  noexcept(swap(declval<Hash>(), declval<Hash>()) &&
  noexcept(swap(declval<Pred>(), declval<Pred>())));

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
std::pair<iterator, iterator> equal_range(const key_type& k);
std::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 Key, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Key, Hash, Pred, Alloc>& x,
          unordered_multiset<Key, Hash, Pred, Alloc>& y);

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);

### 23.5.7.2 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 function, 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 function, 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.

23.5.7.3 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: x.swap(y);

23.6 Container adaptors

23.6.1 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.

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).

23.6.2 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);
template <class T, class Container>
bool operator!=(const queue<T, Container>& x,const queue<T, Container>& y);

§ 23.6.2
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>
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)));

23.6.3 Class template queue

23.6.3.1 queue definition

Any sequence container supporting operations front(), back(), push_back() and pop_front() can be used to instantiate queue. In particular, list (23.3.5) and deque (23.3.3) can be used.

namespace std {
    template <class T, class Container = deque<T> >
    class queue {
    public:
        typedef typename Container::value_type value_type;
        typedef typename Container::reference reference;
        typedef typename Container::const_reference const_reference;
        typedef typename Container::size_type size_type;
        typedef Container container_type;
        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(Container&&, const Alloc&);
        template <class Alloc> queue(const queue&, const Alloc&);
        template <class Alloc> queue(queue&&, const Alloc&);
        bool empty() const { return c.empty(); }
        size_type size() const { return c.size(); }
        reference front() { return c.front(); }
        const_reference front() const { return c.front(); }
        reference back() { return c.back(); }
        const_reference back() const { return c.back(); }
        void push(const value_type& x) { c.push_back(x); }
        void push(value_type&& x) { c.push_back(std::move(x)); }
        template <class... Args> void emplace(Args&&... args)
        { c.emplace_back(std::forward<Args>(args)...); }
        void pop() { c.pop_front(); }
        void swap(queue& q) noexcept(noexcept.swap(c, q.c))
        { using std::swap; swap(c, q.c); }
    };

§ 23.6.3.1
template <class T, class Container>
    bool operator==(const queue<T, Container>& x, const queue<T, Container>& y);

template <class T, class Container>
    bool operator<(const queue<T, Container>& x, const queue<T, Container>& y);

template <class T, class Container>
    bool operator!=(const queue<T, Container>& x, const queue<T, Container>& y);

template <class T, class Container>
    bool operator>(const queue<T, Container>& x, const queue<T, Container>& y);

template <class T, class Container>
    bool operator>=(const queue<T, Container>& x, const queue<T, Container>& y);

template <class T, class Container>
    bool operator<=(const queue<T, Container>& x, const queue<T, Container>& y);

template <class T, class Container>
    void swap(queue<T, Container>& x, queue<T, Container>& y) noexcept(noexcept(x.swap(y)));
23.6.3.4 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.
```

23.6.3.5 queue specialized algorithms

```cpp
template <class T, class Container>
void swap(queue<T, Container>& x, queue<T, Container>& y) noexcept(noexcept(x.swap(y)));

Effects: x.swap(y).
```

23.6.4 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` (23.3.6) and `deque` (23.3.3) can be used. Instantiating `priority_queue` also involves supplying a function or function object for making priority comparisons; the library assumes that the function or function object defines a strict weak ordering (25.4).

```cpp
namespace std {
    template <class T, class Container = vector<T>,
              class Compare = less<typename Container::value_type> >
    class priority_queue {
        public:
            typedef typename Container::value_type value_type;
            typedef typename Container::reference reference;

```
typedef typename Container::const_reference const_reference;
typedef typename Container::size_type size_type;
typedef Container container_type;

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&);
    template <class InputIterator>
        priority_queue(InputIterator first, InputIterator last, const Compare& x = Compare(), Container&& = Container());
    template <class Alloc>
        explicit priority_queue(const Alloc&);
    template <class Alloc>
        priority_queue(const Compare&, const Alloc&);
    template <class Alloc>
        priority_queue(const Compare&, const Container&, const Alloc&);
    template <class Alloc>
        priority_queue(const Compare&, Container&&, const Alloc&);
    template <class Alloc>
        priority_queue(const priority_queue&, const Alloc&);
    template <class Alloc>
        priority_queue(priority_queue&&, const Alloc&);

    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(noexcept(swap(c, q.c)) && noexcept(swap(comp, q.comp)))
        { using std::swap; swap(c, q.c); swap(comp, q.comp); }
};

// 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 { };

23.6.4.1 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 (25.4).
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).

```cpp
template <class InputIterator>
priority_queue(InputIterator first, InputIterator last,
               const Compare& x,
               const Container& y);
```

```cpp
template <class InputIterator>
priority_queue(InputIterator first, InputIterator last,
               const Compare& x = Compare(),
               Container& y = Container());
```

**Requires:** `x` shall define a strict weak ordering (25.4).

**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).

### 23.6.4.2 priority_queue constructors with allocators

[ priqueue.cons.alloc ]

If `uses Allocator<container_type, Alloc>::value` is `false` the constructors in this subclause shall not participate in overload resolution.

```cpp
template <class Alloc>
explicit priority_queue(const Alloc& a);
```

**Effects:** Initializes `c` with `a` and value-initializes `comp`.

```cpp
template <class Alloc>
priority_queue(const Compare& compare, const Alloc& a);
```

**Effects:** Initializes `c` with `a` and initializes `comp` with `compare`.

```cpp
template <class Alloc>
priority_queue(const Compare& compare, const Container& cont, const Alloc& a);
```

**Effects:** Initializes `c` with `cont` as the first argument and `a` as the second argument, and initializes `comp` with `compare`.

```cpp
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`.

```cpp
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`.

```cpp
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)`.
23.6.4.3 priority_queue members

void push(const value_type& x);

Effects:
c.push_back(x);
push_heap(c.begin(), c.end(), comp);

void push(value_type&& x);

Effects:
c.push_back(std::move(x));
push_heap(c.begin(), c.end(), comp);

template <class... Args> void emplace(Args&&... args)

Effects:
c.emplace_back(std::forward<Args>(args)...);
push_heap(c.begin(), c.end(), comp);

void pop();

Effects:
pop_heap(c.begin(), c.end(), comp);
c.pop_back();

23.6.4.4 priority_queue specialized algorithms

template <class T, class Container, Compare>
void swap(priority_queue<T, Container, Compare>& x,
          priority_queue<T, Container, Compare>& y) noexcept(noexcept(x.swap(y)));

Effects: x.swap(y).

23.6.5 Class template stack

Any sequence container supporting operations back(), push_back() and pop_back() can be used to instantiate stack. In particular, vector (23.3.6), list (23.3.5) and deque (23.3.3) can be used.

23.6.5.1 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);

§ 23.6.5.1
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)));

23.6.5.2 stack definition

namespace std {
    template <class T, class Container = deque<T> >
    class stack {
    public:
        typedef typename Container::value_type value_type;
        typedef typename Container::reference reference;
        typedef typename Container::const_reference const_reference;
        typedef typename Container::size_type size_type;
        typedef Container container_type;
    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&);
        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> void emplace(Args&&... args)
        { c.emplace_back(std::forward<Args>(args)...); }
        void pop() { c.pop_back(); }
        void swap(stack& s) noexcept(noexcept(swap(c, s.c)))
        { using std::swap; swap(c, s.c); }
    };

    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)));

§ 23.6.5.2
template <class T, class Container, class Alloc>
struct uses_allocator<stack<T, Container>, Alloc>
  : uses_allocator<Container, Alloc>::type { }

23.6.5.3 stack constructors

explicit stack(const Container& cont);

Effects: Initializes c with cont.

explicit stack(Container&& cont = Container());

Effects: Initializes c with std::move(cont).

23.6.5.4 stack constructors with allocators

If uses_allocator<container_type, Alloc>::value is false the constructors in
this subclause shall not participate in overload resolution.

template <class Alloc>
explicit stack(const Alloc& a);

Effects: Initializes c with a.

template <class Alloc>
stack(const container_type& cont, const Alloc& a);

Effects: Initializes c with cont as the first argument and a as the second
argument.

template <class Alloc>
stack(container_type&& cont, const Alloc& a);

Effects: Initializes c with std::move(cont) as the first argument and a as the
second argument.

template <class Alloc>
stack(const stack& s, const Alloc& a);

Effects: Initializes c with s.c as the first argument and a as the second
argument.

template <class Alloc>
stack(stack&& s, const Alloc& a);

Effects: Initializes c with std::move(s.c) as the first argument and a as the
second argument.

23.6.5.5 stack operators

template <class T, class Container>
bool operator==(const stack<T, Container>& x,
               const stack<T, Container>& y);

Returns: x.c == y.c.

template <class T, class Container>
bool operator!=(const stack<T, Container>& x,
               const stack<T, Container>& y);

Returns: x.c != y.c.
template <class T, class Container>
bool operator< (const stack<T, Container>& x,
const stack<T, Container>& y);

Returns: x.c < y.c.

template <class T, class Container>
bool operator<=(const stack<T, Container>& x,
const stack<T, Container>& y);

Returns: x.c <= y.c.

template <class T, class Container>
bool operator> (const stack<T, Container>& x,
const stack<T, Container>& y);

Returns: x.c > y.c.

template <class T, class Container>
bool operator>=(const stack<T, Container>& x,
const stack<T, Container>& y);

Returns: x.c >= y.c.

23.6.5.6 stack specialized algorithms

[stack.special]

template <class T, class Container>
void swap(stack<T, Container>& x, stack<T, Container>& y) noexcept(noexcept(x.swap(y)));

Effects: x.swap(y). 

§ 23.6.5.6
24 Iterators library

24.1 General

1 This Clause describes components that C++ programs may use to perform iterations over containers (Clause 23), streams (27.7), and stream buffers (27.6).

2 The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 104.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.2</td>
<td>Requirements</td>
</tr>
<tr>
<td>24.4</td>
<td>Iterator primitives &lt;iterator&gt;</td>
</tr>
<tr>
<td>24.5</td>
<td>Predefined iterators</td>
</tr>
<tr>
<td>24.6</td>
<td>Stream iterators</td>
</tr>
</tbody>
</table>

24.2 Iterator requirements

24.2.1 In general

1 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. All input iterators i support the expression *i, resulting in a value of some object type T, called the value type of the iterator. All output iterators support the expression *i = o where o is a value of some type that is in the set of types that are writable to the particular iterator type of i. All iterators i for which the expression (*i).m is well-defined, support the expression i->m with the same semantics as (*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.

2 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 International Standard 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 105.

<table>
<thead>
<tr>
<th>Random Access</th>
<th>Bidirectional</th>
<th>Forward</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
</table>

3 Forward iterators satisfy all the requirements of input iterators and can be used whenever an input iterator is specified; Bidirectional iterators also satisfy all the requirements of forward iterators and can be used whenever a forward iterator is specified; Random access iterators also satisfy all the requirements of bidirectional iterators and can be used whenever a bidirectional iterator is specified.

4 Iterators that further satisfy the requirements of output iterators are called mutable iterators. Nonmutable iterators are referred to as constant iterators.

§ 24.2.1
Iterators that further satisfy the requirement that, for integral values \(n\) and dereferenceable iterator values \(a\) and \((a + n)\), \(*((a + n))\) is equivalent to \(*((\text{addressof}(a) + n))\), are called contiguous iterators. [Note: For example, the type “pointer to int” is a contiguous iterator, but \(\text{reverse_iterator<int*>}\) is not.]

For a valid iterator range \([a,b]\) with dereferenceable \(a\), the corresponding range denoted by pointers is \([\text{addressof}(a),\text{addressof}(a) + (b - a)]\); \(b\) 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 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 \([i,i)\) 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.267

In the following sections, \(a\) and \(b\) denote values of type \(X\) or \(\text{const }X\), difference_type and reference refer to the types \(\text{iterator_traits}\langle X\rangle::\text{difference_type}\) and \(\text{iterator_traits}\langle X\rangle::\text{reference}\), respectively, \(n\) denotes a value of \(\text{difference_type}\), \(u\), \(m\), and \(t\) denote identifiers, \(r\) denotes a value of \(X\), \(o\) 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 \(\text{iterator_traits}\langle X\rangle\) (24.4.1). —end note]

24.2.2 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 (24.2.3) or write (24.2.4) values, or to provide a richer set of iterator movements (24.2.5, 24.2.6, 24.2.7).

A type \(X\) satisfies the Iterator requirements if:

(2.1) — \(X\) satisfies the CopyConstructible, CopyAssignable, and Destructible requirements (17.6.3.1) and lvalues of type \(X\) are swappable (17.6.3.2), and

(2.2) — the expressions in Table 106 are valid and have the indicated semantics.

267) This definition applies to pointers, since pointers are iterators. The effect of dereferencing an iterator that has been invalidated is undefined.
Table 106 — Iterator requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>*r</td>
<td>reference</td>
<td></td>
<td>pre: r is dereferenceable.</td>
</tr>
<tr>
<td>++r</td>
<td>X&amp;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 24.2.3 Input iterators

1. A class or pointer type X satisfies the requirements of an input iterator for the value type T if X satisfies the `Iterator (24.2.2)` and `EqualityComparable (Table 17)` requirements and the expressions in Table 107 are valid and have the indicated semantics.

2. In Table 107, the term *the domain of ==* is used in the ordinary mathematical sense to denote the set of values over which == is (required to be) defined. This set can change over time. Each algorithm places additional requirements on the domain of == for the iterator values it uses. These requirements can be inferred from the uses that algorithm makes of == and !=. [Example: the call `find(a,b,x)` is defined only if the value of a has the property p defined as follows: b has property p and a value i has property p if (*i==x) or if (*i!=x and ++i has property p). — end example]

Table 107 — 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>pre: (a,b) is in the domain of ==.</td>
</tr>
<tr>
<td>*a</td>
<td>convertible to T</td>
<td></td>
<td>pre: a is dereferenceable. The expression (void)*a, *a is equivalent to *a. If a == b and (a,b) is in the domain of == then *a is equivalent to *b.</td>
</tr>
<tr>
<td>a-&gt;m</td>
<td>(*a).m</td>
<td></td>
<td>pre: a is dereferenceable.</td>
</tr>
<tr>
<td>++r</td>
<td>X&amp;</td>
<td></td>
<td>pre: r is dereferenceable. post: r is dereferenceable or r is past-the-end. post: 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>(void)r++</td>
<td></td>
<td></td>
<td>equivalent to (void)++r</td>
</tr>
<tr>
<td>*r++</td>
<td>convertible to T</td>
<td>{ T tmp = *r; ++r; return tmp; }</td>
<td></td>
</tr>
</tbody>
</table>

3. [Note: For input iterators, a == b does not imply ++a == ++b. (Equality does not guarantee the substitution property or referential transparency.) Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be single pass algorithms. Value type T is not required to be |
a `CopyAssignable` type (Table 23). These algorithms can be used with istreams as the source of the input data through the `istream_iterator` class template. — end note]

### 24.2.4 Output iterators

A class or pointer type `X` satisfies the requirements of an output iterator if `X` satisfies the `Iterator` requirements (24.2.2) and the expressions in Table 108 are valid and have the indicated semantics.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>*r = o</code></td>
<td>result is not used</td>
<td><code>Remark: After this operation r is not required to be dereferenceable. post: r is incrementable.</code></td>
<td></td>
</tr>
<tr>
<td><code>++r</code></td>
<td>X&amp;</td>
<td><code>kr == k++r</code>. <code>Remark: After this operation r is not required to be dereferenceable. post: r is incrementable.</code></td>
<td></td>
</tr>
<tr>
<td><code>r++</code></td>
<td>convertible to <code>const X&amp;</code></td>
<td><code>{ X tmp = r; ++r; return tmp; }</code></td>
<td><code>Remark: After this operation r is not required to be dereferenceable. post: r is incrementable.</code></td>
</tr>
<tr>
<td><code>*r++ = o</code></td>
<td>result is not used</td>
<td><code>Remark: After this operation r is not required to be dereferenceable. post: r is incrementable.</code></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 ostreams as the destination for placing data through the `ostream_iterator` class as well as with insert iterators and insert pointers. — end note]

### 24.2.5 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 (24.2.3),
2. `X` satisfies the `DefaultConstructible` requirements (17.6.3.1),
3. if `X` is a mutable iterator, `reference` is a reference to `T`; if `X` is a const iterator, `reference` is a reference to `const T`,
4. the expressions in Table 109 are valid and have the indicated semantics, and
5. objects of type `X` offer the multi-pass guarantee, described below.

2. 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]

3 Two dereferenceable iterators \( a \) and \( b \) of type \( X \) offer the *multi-pass guarantee* if:

\[
\begin{align*}
(3.1) & \quad a == b \implies ++a == ++b \\
(3.2) & \quad X \text{ is a pointer type or the expression } (\text{void})++)X(a), \ *a \text{ is equivalent to the expression } *a.
\end{align*}
\]

[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 109 — 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 pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r++ )</td>
<td>convertible to ( X &amp; )</td>
<td>( { X \ tmp = r; ) ( ++r; ) ( \text{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.

### 24.2.6 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 110.

Table 110 — 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 pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( --r )</td>
<td>( X &amp; )</td>
<td></td>
<td>pre: there exists ( s ) such that ( r == ++s ). post: ( r ) is dereferenceable. ( --(++r) == r ). ( --r == --s ) implies ( r == s ). &amp;r == &amp;--r.</td>
</tr>
<tr>
<td>( r-- )</td>
<td>convertible to ( X &amp; )</td>
<td>( { X \ tmp = r; ) ( --r; ) ( \text{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]
24.2.7 Random access iterators

1 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 111.

Table 111 — 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>
</table>
| r += n     | X&          | { difference_type m = n;
|            |             | if (m >= 0)
|            |             | while (m--)
|            |             | ++r;
|            |             | else
|            |             | while (m++)
|            |             | --r;
|            |             | return r; }         | a + n == n + a. |
| a + n      | X           | { X tmp = a;
|            |             | return tmp += n; }  |                    |
| n + a      | X           | return r += -n;      |                  |
| r -= n     | X&          | return tmp -= n;     |                  |
| a - n      | X           | { X tmp = a;
|            |             | return tmp -= n; }  |                  |
| b - a      | difference_-
|            | type        | return n             | pre: there exists a value n of type difference_type such that a + n == b. b == a + (b - a). |
| a[n]       | convertible to reference | *(a + n) |                  |
| a < b      | contextually convertible to bool | b - a > 0 | < is a total ordering relation |
| a > b      | contextually convertible to bool | b < a | > is a total ordering relation opposite to <. |
| a >= b     | contextually convertible to bool | !(a < b) |                  |
| a <= b     | contextually convertible to bool | !(a > b) |                  |

24.3 Header <iterator> synopsis

namespace std {

// 24.4, primitives:
template<class Iterator> struct iterator_traits;
template<class T> struct iterator_traits<T*>;

template<class Category, class T, class Distance = ptrdiff_t,
class Pointer = T*, class Reference = T&) struct iterator;

§ 24.3
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 { };  

// 24.4.4, iterator operations:  
template <class InputIterator, class Distance>  
void advance(InputIterator& i, Distance n);  
template <class InputIterator>  
typename iterator_traits<InputIterator>::difference_type  
distance(InputIterator first, InputIterator last);  
template <class ForwardIterator>  
ForwardIterator next(ForwardIterator x,  
    typename std::iterator_traits<ForwardIterator>::difference_type n = 1);  
template <class BidirectionalIterator>  
BidirectionalIterator prev(BidirectionalIterator x,  
    typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);  

// 24.5, predefined iterators:  
template <class Iterator> class reverse_iterator;  

template <class Iterator1, class Iterator2>  
bool operator==(  
    const reverse_iterator<Iterator1>& x,  
    const reverse_iterator<Iterator2>& y);  
template <class Iterator1, class Iterator2>  
bool operator<(  
    const reverse_iterator<Iterator1>& x,  
    const reverse_iterator<Iterator2>& y);  
template <class Iterator1, class Iterator2>  
bool operator!=(  
    const reverse_iterator<Iterator1>& x,  
    const reverse_iterator<Iterator2>& y);  
template <class Iterator1, class Iterator2>  
bool operator>(  
    const reverse_iterator<Iterator1>& x,  
    const reverse_iterator<Iterator2>& y);  
template <class Iterator1, class Iterator2>  
bool operator>=(  
    const reverse_iterator<Iterator1>& x,  
    const reverse_iterator<Iterator2>& y);  
template <class Iterator1, class Iterator2>  
bool operator<=(  
    const reverse_iterator<Iterator1>& x,  
    const reverse_iterator<Iterator2>& y);  

template <class Iterator1, class Iterator2>  
auto operator-(  
    const reverse_iterator<Iterator1>& x,  
    const reverse_iterator<Iterator2>& y) ->decltype(y.base() - x.base());  
template <class Iterator>  
reverse_iterator<Iterator>  
operator+(  
    typename reverse_iterator<Iterator>::difference_type n,
const reverse_iterator<Iterator>& x);

template <class Iterator>
reverse_iterator<Iterator> make_reverse_iterator(Iterator i);

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 Container> class move_iterator;
template <class Container1, class Container2>
bool operator=(
    const move_iterator<Container1>& x, const move_iterator<Container2>& y);

template <class Container1, class Container2>
bool operator!=(
    const move_iterator<Container1>& x, const move_iterator<Container2>& y);

template <class Container1, class Container2>
bool operator<(const move_iterator<Container1>& x, const move_iterator<Container2>& y);

template <class Container1, class Container2>
bool operator<=(const move_iterator<Container1>& x, const move_iterator<Container2>& y);

template <class Container1, class Container2>
bool operator>(const move_iterator<Container1>& x, const move_iterator<Container2>& y);

template <class Container1, class Container2>
bool operator>=(const move_iterator<Container1>& x, const move_iterator<Container2>& y);

template <class Container1, class Container2>
auto operator-=(
    const move_iterator<Container1>& x,
    const move_iterator<Container2>& y) -> decltype(x.base() - y.base());

template <class Container>
move_iterator<Container> operator+(typename move_iterator<Container>::difference_type n, const move_iterator<Container>& x);

template <class Container>
move_iterator<Container> make_move_iterator(Iterator i);

// 24.6, 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>
24.4 Iterator primitives

To simplify the task of defining iterators, the library provides several classes and functions:
24.4.1 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 \texttt{Iterator} is the type of an iterator, the types

\begin{verbatim}
    iterator_traits<Iterator>::difference_type
    iterator_traits<Iterator>::value_type
    iterator_traits<Iterator>::iterator_category
\end{verbatim}

be defined as the iterator’s difference type, value type and iterator category, respectively. In addition, the types

\begin{verbatim}
    iterator_traits<Iterator>::reference
    iterator_traits<Iterator>::_pointer
\end{verbatim}

shall be defined as the iterator’s reference and pointer types, that is, for an iterator object \texttt{a}, the same type as the type of \texttt{*a} and \texttt{a->}, respectively. In the case of an output iterator, the types

\begin{verbatim}
    iterator_traits<Iterator>::difference_type
    iterator_traits<Iterator>::value_type
    iterator_traits<Iterator>::reference
    iterator_traits<Iterator>::pointer
\end{verbatim}

may be defined as \texttt{void}.

If \texttt{Iterator} has valid (14.8.2) member types \texttt{difference_type}, \texttt{value_type}, \texttt{pointer}, \texttt{reference}, and \texttt{iterator_category}, \texttt{iterator_traits<Iterator>} shall have the following as publicly accessible members and shall have no other members:

\begin{verbatim}
    typedef typename Iterator::difference_type difference_type;
    typedef typename Iterator::value_type value_type;
    typedef typename Iterator::pointer pointer;
    typedef typename Iterator::reference reference;
    typedef typename Iterator::iterator_category iterator_category;
\end{verbatim}

Otherwise, \texttt{iterator_traits<Iterator>} shall have no members.

It is specialized for pointers as

\begin{verbatim}
namespace std {
    template<class T> struct iterator_traits<T*> {
        typedef ptrdiff_t difference_type;
        typedef T value_type;
        typedef T* pointer;
        typedef T& reference;
        typedef random_access_iterator_tag iterator_category;
    };
}
\end{verbatim}

and for pointers to const as

\begin{verbatim}
namespace std {
    template<class T> struct iterator_traits<const T*> {
        typedef ptrdiff_t difference_type;
        typedef T value_type;
        typedef const T* pointer;
        typedef const T& reference;
        typedef random_access_iterator_tag iterator_category;
    };
}
\end{verbatim}
Example: To implement a generic `reverse` function, a C++ program can do the following:

```cpp
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

24.4.2 Basic iterator

The `iterator` template may be used as a base class to ease the definition of required types for new iterators.

```cpp
namespace std {
    template<class Category, class T, class Distance = ptrdiff_t,
        class Pointer = T*, class Reference = T&>
    struct iterator {
        typedef T value_type;
        typedef Distance difference_type;
        typedef Pointer pointer;
        typedef Reference reference;
        typedef Category iterator_category;
    };
}
```

24.4.3 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.

```cpp
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>> {
    typedef std::ptrdiff_t difference_type;
    typedef T value_type;
    typedef T* pointer;
    typedef T& reference;
    typedef bidirectionalIteratorTag iterator_category;
};

Typically, however, it would be easier to derive BinaryTreeIterator<T> from iterator<bidirectional_iterator_tag, T, ptrdiff_t, T*, 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]

4 [Example: If a C++ program wants to define a bidirectional iterator for some data structure containing double and such that it works on a large memory model of the implementation, it can do so with:

class MyIterator :
    public iterator<bidirectional_iterator_tag, double, long, T*, T&> {
        // code implementing ++, etc.
    };

Then there is no need to specialize the iterator_traits template. —end example]

24.4.4 Iterator operations [iterator.operations]

1 Since only random access iterators provide + and − operators, the library provides two function templates advance and distance. These function templates use + and − for random access iterators (and are, therefore, constant time for them); for input, forward and bidirectional iterators they use ++ to provide linear time implementations.

template <class InputIterator, class Distance>
void advance(InputIterator& i, Distance n);

2 Requires: n shall be negative only for bidirectional and random access iterators.

3 Effects: Increments (or decrements for negative n) iterator reference i by n.

§ 24.4.4
template<class InputIterator>
    typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);

4 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.

5 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.

template <class ForwardIterator>
    ForwardIterator next(ForwardIterator x,
        typename std::iterator_traits<ForwardIterator>::difference_type n = 1);

6 Effects: Equivalent to advance(x, n); return x;

template <class BidirectionalIterator>
    BidirectionalIterator prev(BidirectionalIterator x,
        typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);

7 Effects: Equivalent to advance(x, -n); return x;

24.5 Iterator adaptors

24.5.1 Reverse iterators

1 Class template reverse_iterator is an iterator adaptor that iterates from the end of the sequence defined by its underlying iterator to the beginning of that sequence. The fundamental relation between a reverse iterator and its corresponding iterator i is established by the identity: \&*(reverse_iterator(i)) == \&*(i - 1).

24.5.1.1 Class template reverse_iterator

    namespace std {
        template <class Iterator>
            class reverse_iterator : public
                        iterator<typename iterator_traits<Iterator>::iterator_category,
                               typename iterator_traits<Iterator>::value_type,
                               typename iterator_traits<Iterator>::difference_type,
                               typename iterator_traits<Iterator>::pointer,
                               typename iterator_traits<Iterator>::reference> {
                        public:
                            typedef Iterator iterator_type;
                            typedef typename iterator_traits<Iterator>::difference_type difference_type;
                            typedef typename iterator_traits<Iterator>::reference reference;
                            typedef typename iterator_traits<Iterator>::pointer pointer;

                            reverse_iterator();
                            explicit reverse_iterator(Iterator x);
                            template <class U> reverse_iterator(const reverse_iterator<U>& u);
                            template <class U> reverse_iterator& operator=(const reverse_iterator<U>& u);

                            Iterator base() const; // explicit
                            reference operator*() const;
                            pointer operator->() const;

                            reverse_iterator& operator++();
                            reverse_iterator& operator++(int);
                            reverse_iterator& operator--();

§ 24.5.1.1
reverse_iterator operator--(int);

reverse_iterator operator+ (difference_type n) const;
reverse_iterator& operator+=(difference_type n);

reverse_iterator operator- (difference_type n) const;
reverse_iterator& operator-=(difference_type n);

unspecified operator[](difference_type n) const;

protected:
    Iterator current;
};

template <class Iterator1, class Iterator2>
bool operator==(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
bool operator<(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
bool operator!=(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
bool operator>(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
bool operator<=(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
bool operator>=(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
auto operator-(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y) -> decltype(y.base() - x.base());

template <class Iterator>
reverse_iterator<Iterator> operator+(typename reverse_iterator<Iterator>::difference_type n, const reverse_iterator<Iterator>& x);

template <class Iterator>
reverse_iterator<Iterator> make_reverse_iterator(Iterator i);

24.5.1.2 reverse_iterator requirements

1 The template parameter Iterator shall meet all the requirements of a Bidirectional Iterator (24.2.6).

2 Additionally, Iterator shall meet the requirements of a Random Access Iterator (24.2.7) if any of the members operator+ (24.5.1.3.8), operator- (24.5.1.3.10), operator+= (24.5.1.3.9), operator-= (24.5.1.3.11),
operator [] (24.5.1.3.12), or the global operators operator< (24.5.1.3.14), operator> (24.5.1.3.16),
operator <= (24.5.1.3.18), operator>= (24.5.1.3.17), operator< (24.5.1.3.19) or operator+ (24.5.1.3.20) are
referenced in a way that requires instantiation (14.7.1).

24.5.1.3 reverse_iterator operations

24.5.1.3.1 reverse_iterator constructor

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.

explicit reverse_iterator(Iterator x);

Effects: Initializes current with x.

template <class U> reverse_iterator(const reverse_iterator<U> &u);

Effects: Initializes current with u.current.

24.5.1.3.2 reverse_iterator::operator=

template <class U>
reverse_iterator&
operator=(const reverse_iterator<U>& u);

Effects: Assigns u.base() to current.

24.5.1.3.3 Conversion

Iterator base() const; // explicit

Returns: current.

24.5.1.3.4 operator*

reference operator*() const;

Effects:
Iterator tmp = current;
return *--tmp;

24.5.1.3.5 operator->

pointer operator->() const;

Returns: std::addressof(operator*()).

24.5.1.3.6 operator++

reverse_iterator& operator++();

Effects: --current;

Returns: *this.

reverse_iterator operator++(int);

Effects:
reverse_iterator tmp = *this;
--current;
return tmp;
24.5.1.3.7 operator--

reverse_iterator& operator--();
1    Effects: ++current
2    Returns: *this.

reverse_iterator operator--(int);
3    Effects:
        reverse_iterator tmp = *this;
        ++current;
        return tmp;

24.5.1.3.8 operator+

reverse_iterator operator+(typename reverse_iterator<Iterator>::difference_type n) const;
1    Returns: reverse_iterator(current-n).

24.5.1.3.9 operator++

reverse_iterator& operator++(typename reverse_iterator<Iterator>::difference_type n);
1    Effects: current -= n;
2    Returns: *this.

24.5.1.3.10 operator-

reverse_iterator operator-(typename reverse_iterator<Iterator>::difference_type n) const;
1    Returns: reverse_iterator(current+n).

24.5.1.3.11 operator--

reverse_iterator& operator--(typename reverse_iterator<Iterator>::difference_type n);
1    Effects: current += n;
2    Returns: *this.

24.5.1.3.12 operator[]

unspecified operator[](typename reverse_iterator<Iterator>::difference_type n) const;
1    Returns: current[-n-1].

24.5.1.3.13 operator==

template <class Iterator1, class Iterator2>
bool operator==(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);
1    Returns: x.current == y.current.

§ 24.5.1.3.13
24.5.1.3.14 operator<

```
template <class Iterator1, class Iterator2>
  bool operator<(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

```
1 Returns: x.current > y.current.
```

24.5.1.3.15 operator!=

```
template <class Iterator1, class Iterator2>
  bool operator!=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

```
1 Returns: x.current != y.current.
```

24.5.1.3.16 operator>

```
template <class Iterator1, class Iterator2>
  bool operator>(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

```
1 Returns: x.current < y.current.
```

24.5.1.3.17 operator>=

```
template <class Iterator1, class Iterator2>
  bool operator>(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

```
1 Returns: x.current <= y.current.
```

24.5.1.3.18 operator<=

```
template <class Iterator1, class Iterator2>
  bool operator<=
    (const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

```
1 Returns: x.current >= y.current.
```

24.5.1.3.19 operator-  

```
template <class Iterator1, class Iterator2>
  auto operator-(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y)
  -> decltype(y.base() - x.base());
```

```
1 Returns: y.current - x.current.
```

24.5.1.3.20 operator+

```
template <class Iterator>
  reverse_iterator<Iterator> operator+( 
    typename reverse_iterator<Iterator>:::difference_type n, 
    const reverse_iterator<Iterator>& x);
```

```
1 Returns: reverse_iterator<Iterator> (x.current - n).
```
24.5.1.3.21  Non-member function make_reverse_iterator()

 template <class Iterator>
   reverse_iterator<Iterator> make_reverse_iterator(Iterator i);

   Returns: reverse_iterator<Iterator>(i).

24.5.2  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,

   while (first != last) *result++ = *first++;

causes a range [first, last) to be copied into a range starting with result. The same code with result being an insert iterator will insert corresponding elements into the container. This device allows all of the copying algorithms in the library to work in the insert mode instead of the regular overwrite mode.

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. operator* returns the insert iterator itself. The assignment operator=(const T& x) is defined on insert iterators to allow writing into them, it inserts x right before where the insert iterator is pointing. In other words, an insert iterator is like a cursor pointing into the container where the insertion takes place. back_insert_iterator inserts elements at the end of a container, front_insert_iterator inserts elements at the beginning of a container, and insert_iterator inserts elements where the iterator points to in a container. back_inserter, front_inserter, and inserter are three functions making the insert iterators out of a container.

24.5.2.1  Class template back_insert_iterator

 namespace std {
   template <class Container>
   class back_insert_iterator : 
     public iterator<output_iterator_tag,void,void,void,void> {
       protected:
         Container* container;

       public:
         typedef Container container_type;
         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);
 }

24.5.2.2  back_insert_iterator operations

24.5.2.2.1  back_insert_iterator constructor

   explicit back_insert_iterator(Container& x);

   Effects: Initializes container with std::addressof(x).

§ 24.5.2.2.1 869
24.5.2.2.2 back_insert_iterator::operator=

back_insert_iterator& operator=(const typename Container::value_type& value);

Effects: container->push_back(value);

Returns: *this.

back_insert_iterator& operator=(typename Container::value_type&& value);

Effects: container->push_back(std::move(value));

Returns: *this.

24.5.2.2.3 back_insert_iterator::operator*

back_insert_iterator& operator*();

Returns: *this.

24.5.2.2.4 back_insert_iterator::operator++

back_insert_iterator& operator++();

Returns: *this.

24.5.2.2.5 back_inserter

template <class Container>
back_insert_iterator<Container> back_inserter(Container& x);

Returns: back_insert_iterator<Container>(x).

24.5.2.3 Class template front_insert_iterator

namespace std {
    template <class Container>
    class front_insert_iterator :
        public iterator<output_iterator_tag,void,void,void,void> {
            protected:
                Container* container;

            public:
                typedef Container container_type;
                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);
}

24.5.2.4 front_insert_iterator operations

24.5.2.4.1 front_insert_iterator constructor
explicit front_insert_iterator(Container& x);

Effects: Initializes container with std::addressof(x).

24.5.2.4.2 front_insert_iterator::operator=

front_insert_iterator& operator=(const typename Container::value_type& value);

Effects: container->push_front(value);

Returns: *this.

front_insert_iterator& operator=(typename Container::value_type&& value);

Effects: container->push_front(std::move(value));

Returns: *this.

24.5.2.4.3 front_insert_iterator::operator*

front_insert_iterator& operator*();

Returns: *this.

24.5.2.4.4 front_insert_iterator::operator++

front_insert_iterator& operator++();

front_insert_iterator operator++(int);

Returns: *this.

24.5.2.4.5 front_inserter

template <class Container>
front_insert_iterator<Container> front_inserter(Container& x);

Returns: front_insert_iterator<Container>(x).

24.5.2.5 Class template insert_iterator

namespace std {

template <class Container>
class insert_iterator :

public iterator<output_iterator_tag,void,void,void,void> {

protected:
Container* container;
type_name Container::iterator iter;

public:

typedef Container container_type;
insert_iterator(Container& x, type_name Container::iterator i);
insert_iterator& operator=(const type_name 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, type_name Container::iterator i);
}

§ 24.5.2.5
24.5.2.6  insert_iterator operations

24.5.2.6.1  insert_iterator constructor

insert_iterator(Container& x, typename Container::iterator i);

Effects: Initializes container with std::addressof(x) and iter with i.

24.5.2.6.2  insert_iterator::operator=

insert_iterator& operator=(const typename Container::value_type& value);

Effects:
iter = container->insert(iter, value);
++iter;

Returns: *this.

insert_iterator& operator=(typename Container::value_type&& value);

Effects:
iter = container->insert(iter, std::move(value));
++iter;

Returns: *this.

24.5.2.6.3  insert_iterator::operator*

insert_iterator& operator*();

Returns: *this.

24.5.2.6.4  insert_iterator::operator++

insert_iterator& operator++();
insert_iterator& operator++(int);

Returns: *this.

24.5.2.6.5  inserter

template <class Container>
insert_iterator<Container> inserter(Container& x, typename Container::iterator i);

Returns: insert_iterator<Container>(x, i).

24.5.3  Move iterators

Class template move_iterator is an iterator adaptor with the same behavior as the underlying iterator except that its indirection operator implicitly converts the value returned by the underlying iterator’s indirection operator to an rvalue. Some generic algorithms can be called with move iterators to replace copying with moving.

Example:

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]
24.5.3.1 Class template move_iterator

namespace std {
    template <class Iterator>
    class move_iterator {
        public:
            typedef Iterator iterator_type;
            typedef typename iterator_traits<Iterator>::difference_type difference_type;
            typedef Iterator pointer;
            typedef typename iterator_traits<Iterator>::value_type value_type;
            typedef typename iterator_traits<Iterator>::iterator_category iterator_category;
            typedef see below reference;

            move_iterator();
            explicit move_iterator(Iterator i);
            template <class U> move_iterator(const move_iterator<U>& u);
            template <class U> move_iterator& operator=(const move_iterator<U>& u);

            iterator_type base() const;
            reference operator*() const;
            pointer operator->() const;

            move_iterator& operator++();
            move_iterator operator++(int);
            move_iterator& operator--();
            move_iterator operator--(int);

            move_iterator operator+(difference_type n) const;
            move_iterator& operator+=(difference_type n);
            move_iterator operator-(difference_type n) const;
            move_iterator& operator-=(difference_type n);
            unspecified operator[](difference_type n) const;

        private:
            Iterator current; // exposition only
    };

    template <class Iterator1, class Iterator2>
    bool operator==(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
    template <class Iterator1, class Iterator2>
    bool operator!=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
    template <class Iterator1, class Iterator2>
    bool operator<(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
    template <class Iterator1, class Iterator2>
    bool operator<=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
    template <class Iterator1, class Iterator2>
    bool operator>(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
    template <class Iterator1, class Iterator2>
    bool operator>=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

§ 24.5.3.1
template <class Iterator1, class Iterator2>
auto operator-(
    const move_iterator<Iterator1>& x,
    const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base());

template <class Iterator>
move_iterator<Iterator> operator+(
    typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x);

template <class Iterator>
move_iterator<Iterator> make_move_iterator(Iterator i);

1 Let R be iterator_traits<Iterator>::reference. If is_reference<R>::value is true, the template specialization move_iterator<Iterator> shall define the nested type named reference as a synonym for remove_reference<R>::type&&, otherwise as a synonym for R.

24.5.3.2 move_iterator requirements

The template parameter Iterator shall meet the requirements for an Input Iterator (24.2.3). Additionally, if any of the bidirectional or random access traversal functions are instantiated, the template parameter shall meet the requirements for a Bidirectional Iterator (24.2.6) or a Random Access Iterator (24.2.7), respectively.

24.5.3.3 move_iterator operations

24.5.3.3.1 move_iterator constructors

move_iterator();

Effects: Constructs a move_iterator, value initializing current. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type Iterator.

explicit move_iterator(Iterator i);

Effects: Constructs a move_iterator, initializing current with i.

template <class U> move_iterator(const move_iterator<U>& u);

Effects: Constructs a move_iterator, initializing current with u.base().

Requires: U shall be convertible to Iterator.

24.5.3.3.2 move_iterator::operator=

template <class U> move_iterator& operator=(const move_iterator<U>& u);

Effects: Assigns u.base() to current.

Requires: U shall be convertible to Iterator.

24.5.3.3.3 move_iterator conversion

Iterator base() const;

Returns: current.

24.5.3.3.4 move_iterator::operator*

reference operator*() const;

Returns: static_cast<reference>(*current).
24.5.3.3.5 move_iterator::operator->

pointer operator->() const;

Returns: current.

24.5.3.3.6 move_iterator::operator++

move_iterator& operator++();

Effects: ++current.

Returns: *this.

move_iterator operator++(int);

Effects:
move_iterator tmp = *this;
++current;
return tmp;

24.5.3.3.7 move_iterator::operator--

move_iterator& operator--();

Effects: --current.

Returns: *this.

move_iterator operator--(int);

Effects:
move_iterator tmp = *this;
--current;
return tmp;

24.5.3.3.8 move_iterator::operator+

move_iterator operator+(difference_type n) const;

Returns: move_iterator(current + n).

24.5.3.3.9 move_iterator::operator+=

move_iterator& operator+=(difference_type n);

Effects: current += n.

Returns: *this.

24.5.3.3.10 move_iterator::operator-

move_iterator operator-(difference_type n) const;

Returns: move_iterator(current - n).

24.5.3.3.11 move_iterator::operator-=

move_iterator& operator-=(difference_type n);

Effects: current -= n.

Returns: *this.
24.5.3.3.12 move_iterator::operator[]  

unspecified operator[](difference_type n) const;  

Returns: std::move(current[n]).

24.5.3.3.13 move_iterator comparisons

template <class Iterator1, class Iterator2>  
bool operator==(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);  

Returns: x.base() == y.base().

template <class Iterator1, class Iterator2>  
bool operator!=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);  

Returns: !(x == y).

template <class Iterator1, class Iterator2>  
bool operator<(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);  

Returns: x.base() < y.base().

template <class Iterator1, class Iterator2>  
bool operator<=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);  

Returns: !(y < x).

template <class Iterator1, class Iterator2>  
bool operator>(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);  

Returns: y < x.

template <class Iterator1, class Iterator2>  
bool operator>=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);  

Returns: !(x < y).

24.5.3.3.14 move_iterator non-member functions

template <class Iterator1, class Iterator2>  
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>  
move_iterator<Iterator> operator+(  
    typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x);  

Returns: x + n.

template <class Iterator>  
move_iterator<Iterator> make_move_iterator(Iterator i);  

Returns: move_iterator<Iterator>(i).
24.6 Stream iterators

To make it possible for algorithmic templates to work directly with input/output streams, appropriate iterator-like class templates are provided.

[Example:
partial_sum(istream_iterator<double, char>(cin),
    istream_iterator<double, char>(),
    ostream_iterator<double, char>(cout, "\n"));

reads a file containing floating point numbers from cin, and prints the partial sums onto cout. — end example]

24.6.1 Class template istream_iterator

The class template istream_iterator is an input iterator (24.2.3) 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.

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 = ptrdiff_t>
    class istream_iterator {
        public: iter<input_iterator_tag, T, Distance, const T*, const T&> {
            public:
                typedef charT char_type;
                typedef traits traits_type;
                typedef basic_istream<charT,traits> istream_type;
                see below 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);
}

24.6.1.1 istream_iterator constructors and destructor

see below istream_iterator();

1 Effects: Constructs the end-of-stream iterator. If T is a literal type, then this constructor shall be a constexpr constructor.
2 Postcondition: in_stream == 0.

istream_iterator(istream_type& s);

3 Effects: Initializes in_stream with &s.value may be initialized during construction or the first time it
2 Postcondition: in_stream == &s.

istream_iterator(const istream_iterator& x) = default;

5 Effects: Constructs a copy of x. If T is a literal type, then this constructor shall be a trivial copy
6 Postcondition: in_stream == x.in_stream.

~istream_iterator() = default;

7 Effects: The iterator is destroyed. If T is a literal type, then this destructor shall be a trivial destructor.

24.6.1.2 istream_iterator operations

const T& operator*() const;

1 Returns: value.

const T* operator->() const;

2 Returns: &(operator*()).

istream_iterator& operator++();

3 Requires: in_stream != 0.
4 Effects: *in_stream >>=value.
5 Returns: *this.

istream_iterator operator++(int);

6 Requires: in_stream != 0.
7 Effects:

    istream_iterator tmp = *this;
    *in_stream >>= value;
    return (tmp);

7 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);

§ 24.6.1.2
Returns: \(x\text{.}in\_stream == y\text{.}in\_stream\).

\[
\text{template <class T, class charT, class traits, class Distance>}
\]
\[
\text{bool operator!=<\text{const istream\_iterator<T,charT,traits,Distance>} &x,}
\]
\[
\text{const istream\_iterator<T,charT,traits,Distance>} &y);}
\]

Returns: \(! (x == y)\)

### 24.6.2 Class template ostream\_iterator

\text{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 \textit{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
\[
\text{while (first != last)}
\]
\[
*\text{result}++ = *\text{first}++;}
\]

\text{ostream\_iterator} is defined as:

\[
\text{namespace std} \{}
\]
\[
\text{template <class T, class charT = char, class traits = char\_traits<charT>>}
\]
\[
\text{class ostream\_iterator:}
\]
\[
\text{public iterator<output\_iterator\_tag, void, void, void, void>} \{}
\]
\[
\text{public:}
\]
\[
\text{typedef charT char\_type;}
\]
\[
\text{typedef traits traits\_type;}
\]
\[
\text{typedef basic\_ostream<charT,traits> ostrem\_type;}
\]
\[
\text{ostream\_iterator(ostream\_type& s);} \;
\]
\[
\text{ostream\_iterator(ostream\_type& s, const charT* delimiter);} \;
\]
\[
\text{ostream\_iterator(const ostream\_iterator& x);} \;
\]
\[
-\text{ostream\_iterator();}
\]
\[
\text{ostream\_iterator& operator=(const T& value);} \;
\]
\[
\text{ostream\_iterator& operator*();}
\]
\[
\text{ostream\_iterator& operator++();}
\]
\[
\text{ostream\_iterator& operator++(int);} \;
\]
\[
\text{private:}
\]
\[
\text{basic\_ostream<charT,traits>** out\_stream; // exposition only}
\]
\[
\text{const charT* delim; // exposition only}
\]
\[
\};}
\]

#### 24.6.2.1 ostream\_iterator constructors and destructor

\text{ostream\_iterator(ostream\_type& s);} \;

\textit{Effects}: Initializes out\_stream with \&s and delim with null.

\text{ostream\_iterator(ostream\_type& s, const charT* delimiter);} \;

\textit{Effects}: Initializes out\_stream with \&s and delim with delimiter.

\text{ostream\_iterator(const ostream\_iterator& x);} \;

\textit{Effects}: Constructs a copy of x.

\text{~ostream\_iterator();} \;

\textit{Effects}: The iterator is destroyed.
24.6.2.2 ostream_iterator operations

 ostream_iterator& operator=(const T& value);

 Effects:
 * out_stream << value;
 if (delim != 0)
   * out_stream << delim;
 return *this;

 ostream_iterator& operator*();

 Returns: *this.

 ostream_iterator& operator++();
 ostream_iterator& operator++(int);

 Returns: *this.

24.6.3 Class template istreambuf_iterator

 The class template istreambuf_iterator defines an input iterator (24.2.3) that reads successive characters from the streambuf for which it was constructed. 
operator* provides access to the current input character, if any. [Note: operator-> may return a proxy. —end note] 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 iterator<input_iterator_tag, charT,
                       typename traits::off_type, unspecified, charT> {
   public:
     typedef charT char_type;
     typedef traits traits_type;
     typedef typename traits::int_type int_type;
     typedef basic_streambuf<charT,traits> streambuf_type;
     typedef basic_istream<charT,traits> istream_type;

     class proxy; // exposition only

     constexpr istreambuf_iterator() noexcept;
     istreambuf_iterator(const istreambuf_iterator&) noexcept = default;
     ~istreambuf_iterator() = default;
     istreambuf_iterator(istreambuf_iterator&) noexcept = default;
     istreambuf_iterator(istream_type& s) noexcept;
     istreambuf_iterator(streambuf_type* s) noexcept;
     istreambuf_iterator(const proxy& p) noexcept;
     charT operator*() const;
     pointer operator->() const;
     istreambuf_iterator& operator++();
     proxy operator++(int);

§ 24.6.3 880
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);
template <class charT, class traits>
bool operator!=(const istreambuf_iterator<charT,traits>& a,
               const istreambuf_iterator<charT,traits>& b);

24.6.3.1 Class template istreambuf_iterator::proxy

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.

24.6.3.2 istreambuf_iterator constructors

constexpr istreambuf_iterator() noexcept;

1 Effects: Constructs the end-of-stream iterator.

istreambuf_iterator(basic_istream<charT,traits>& s) noexcept;
istreambuf_iterator(basic_streambuf<charT,traits>* s) noexcept;

2 Effects: Constructs an istreambuf_iterator<> that uses the basic_streambuf<> object *(s.rdbuf()), or *s, respectively. Constructs an end-of-stream iterator if s.rdbuf() is null.

istreambuf_iterator(const proxy& p) noexcept;

3 Effects: Constructs a istreambuf_iterator<> that uses the basic_streambuf<> object pointed to by the proxy object’s constructor argument p.

24.6.3.3 istreambuf_iterator::operator*

charT operator*() const

1 Returns: The character obtained via the streambuf member sbuf_->sgetc().
24.6.3.4 `istreambuf_iterator::operator++`  
`istreambuf_iterator& operator++();`  
1. **Effects**: `sbuf_->sbumpc()`.  
2. **Returns**: `*this`.

proxy `operator++(int);`  
3. **Returns**: `proxy(sbuf_->sbumpc(), sbuf_)`.

24.6.3.5 `istreambuf_iterator::equal`  
bool `equal(const istreambuf_iterator& b) const;`  
1. **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.

24.6.3.6 `operator==`  
template <class charT, class traits>  
bool `operator==(const istreambuf_iterator<charT, traits>& a,  
const istreambuf_iterator<charT, traits>& b);`  
1. **Returns**: `a.equal(b)`.

24.6.3.7 `operator!=`  
template <class charT, class traits>  
bool `operator!=(const istreambuf_iterator<charT, traits>& a,  
const istreambuf_iterator<charT, traits>& b);`  
1. **Returns**: `!a.equal(b)`.

24.6.4 Class template `ostreambuf_iterator`  
namespace std {  
    template <class charT, class traits = char_traits<charT> >  
    class `ostreambuf_iterator` :  
    public iterator<output_iterator_tag, void, void, void, void> {  
    public:  
        typedef charT char_type;  
        typedef traits traits_type;  
        typedef basic_streambuf<charT, traits> streambuf_type;  
        typedef basic_ostream<charT, traits> ostream_type;  
        public:  
            `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
    };  
}  

§ 24.6.4
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.

24.6.4.1 `ostreambuf_iterator` constructors

```cpp
ostreambuf_iterator(ostream_type& s) noexcept;
```

1 Requires: `s.rdbuf()` shall not be a null pointer.

```cpp
Effects: Initializes `sbuf_` with `s.rdbuf()`.
```

```cpp
ostreambuf_iterator(streambuf_type* s) noexcept;
```

2 Requires: `s` shall not be a null pointer.

```cpp
Effects: Initializes `sbuf_` with `s`.
```

24.6.4.2 `ostreambuf_iterator` operations

```cpp
ostreambuf_iterator& operator=(charT c);
```

1 Effects: If `failed()` yields `false`, calls `sbuf_->sputc(c)`; otherwise has no effect.

```cpp
Returns: *this.
```

```cpp
ostreambuf_iterator& operator*();
```

3 Returns: *this.

```cpp
ostreambuf_iterator& operator++();
ostreambuf_iterator& operator++(int);
```

4 Returns: *this.

```cpp
bool failed() const noexcept;
```

5 Returns: true if in any prior use of member `operator=`, the call to `sbuf_->sputc()` returned `traits::eof()`; or `false` otherwise.

24.7 Range access

In addition to being available via inclusion of the `<iterator>` header, the function templates in 24.7 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>`.

```cpp
template <class C> auto begin(C& c) -> decltype(c.begin());
template <class C> auto begin(const C& c) -> decltype(c.begin());
```

2 Returns: `c.begin()`.

```cpp
template <class C> auto end(C& c) -> decltype(c.end());
template <class C> auto end(const C& c) -> decltype(c.end());
```

3 Returns: `c.end()`.

```cpp
template <class T, size_t N> constexpr T* begin(T (&array)[N]) noexcept;
```

4 Returns: `array`.

```cpp
template <class T, size_t N> constexpr T* end(T (&array)[N]) noexcept;
```

5 Returns: `array + N`.

§ 24.7
template <class C> constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
    -> decltype(std::begin(c));

    Returns: std::begin(c).

template <class C> constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
    -> decltype(std::end(c));

    Returns: std::end(c).

template <class C> auto rbegin(C& c) -> decltype(c.rbegin());
template <class C> auto rbegin(const C& c) -> decltype(c.rbegin());

    Returns: c.rbegin().

template <class C> auto rend(C& c) -> decltype(c.rend());
template <class C> auto rend(const C& c) -> decltype(c.rend());

    Returns: c.rend().

template <class T, size_t N> reverse_iterator<T*> rbegin(T (&array)[N]);

    Returns: reverse_iterator<T*>(array + N).

template <class T, size_t N> reverse_iterator<T*> rend(T (&array)[N]);

    Returns: reverse_iterator<T*>(array).

template <class E> reverse_iterator<const E*> rbegin(initializer_list<E> il);

    Returns: reverse_iterator<const E*>(il.end()).

template <class E> reverse_iterator<const E*> rend(initializer_list<E> il);

    Returns: reverse_iterator<const E*>(il.begin()).

template <class C> auto crbegin(const C& c) -> decltype(std::rbegin(c));

    Returns: std::rbegin(c).

template <class C> auto crend(const C& c) -> decltype(std::rend(c));

    Returns: std::rend(c).

24.8 Container access

In addition to being available via inclusion of the <iterator> header, the function templates in 24.8 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> constexpr auto empty(const C& c) -> decltype(c.empty());

    Returns: c.empty().
template <class T, size_t N> constexpr bool empty(const T (&array)[N]) noexcept;

Returns: false.

template <class E> 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().
25 Algorithms library

25.1 General

This Clause describes components that C++ programs may use to perform algorithmic operations on containers (Clause 23) and other sequences.

The following subclauses describe components for non-modifying sequence operation, modifying sequence operations, sorting and related operations, and algorithms from the ISO C library, as summarized in Table 112.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.2 Non-modifying sequence operations</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>25.3 Mutating sequence operations</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>25.4 Sorting and related operations</td>
<td></td>
</tr>
<tr>
<td>25.5 C library algorithms</td>
<td>&lt;cstdlib&gt;</td>
</tr>
</tbody>
</table>

Header <algorithm> synopsis

```cpp
#include <initializer_list>
namespace std {

    // 25.2, non-modifying sequence operations:
    template <class InputIterator, class Predicate>
    bool all_of(InputIterator first, InputIterator last, Predicate pred);
    template <class InputIterator, class Predicate>
    bool any_of(InputIterator first, InputIterator last, Predicate pred);
    template <class InputIterator, class Predicate>
    bool none_of(InputIterator first, InputIterator last, Predicate pred);

    template<class InputIterator, class Function>
    Function for_each(InputIterator first, InputIterator last, Function f);
    template<class InputIterator, class T>
    InputIterator find(InputIterator first, InputIterator last, const T& value);
    template<class InputIterator, class Predicate>
    InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);
    template<class InputIterator, class Predicate>
    InputIterator find_if_not(InputIterator first, InputIterator last, Predicate pred);
    template<class ForwardIterator1, class ForwardIterator2>
    ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2);
    template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
    ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1, ...
```
ForwardIterator2 first2, ForwardIterator2 last2,
BinaryPredicate pred);

template<class InputIterator, class ForwardIterator>
InputIterator
find_first_of(InputIterator first1, InputIterator last1,
ForwardIterator first2, ForwardIterator last2);

template<class InputIterator, class ForwardIterator,
class BinaryPredicate>
InputIterator
find_first_of(InputIterator first1, InputIterator last1,
ForwardIterator first2, ForwardIterator last2,
BinaryPredicate pred);

template<class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first,
ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first,
ForwardIterator last,
BinaryPredicate pred);

template<class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& value);

template<class InputIterator, class Predicate>
typename iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred);

template<class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2);

template
<class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, InputIterator2 last2);

template
<class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, InputIterator2 last2,
           BinaryPredicate pred);

template<class ForwardIterator1, class ForwardIterator2>
bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                    ForwardIterator2 first2);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                    ForwardIterator2 first2, BinaryPredicate pred);

template<class ForwardIterator1, class ForwardIterator2>
bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                    ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                    ForwardIterator2 first2, ForwardIterator2 last2,
                    BinaryPredicate pred);

template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
ForwardIterator1 search(
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2,
    BinaryPredicate pred);

template<class ForwardIterator, class Size, class T>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                         Size count, const T& value);

template
<class ForwardIterator, class Size, class T, class BinaryPredicate>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                         Size count, const T& value,
                         BinaryPredicate pred);

// 25.3, modifying sequence operations:
// 25.3.1, copy:
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
OutputIterator result);

template<class InputIterator, class Size, class OutputIterator>
  OutputIterator copy_n(InputIterator first, Size n,
                          OutputIterator result);

template<class InputIterator, class OutputIterator, class Predicate>
  OutputIterator copy_if(InputIterator first, InputIterator last,
                          OutputIterator result, Predicate pred);

template<class BidirectionalIterator1, class BidirectionalIterator2>
  BidirectionalIterator2 copy_backward(
    BidirectionalIterator1 first, BidirectionalIterator1 last,
    BidirectionalIterator2 result);

// 25.3.2, move:

template<class InputIterator, class OutputIterator>
  OutputIterator move(InputIterator first, InputIterator last,
                       OutputIterator result);

template<class BidirectionalIterator1, class BidirectionalIterator2>
  BidirectionalIterator2 move_backward(
    BidirectionalIterator1 first, BidirectionalIterator1 last,
    BidirectionalIterator2 result);

// 25.3.3, swap:

template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 swap_ranges(ForwardIterator1 first1,
                              ForwardIterator1 last1, ForwardIterator2 first2);

template<class ForwardIterator1, class ForwardIterator2>
  void iter_swap(ForwardIterator1 a, ForwardIterator2 b);

  template<class InputIterator, class OutputIterator, class UnaryOperation>
    OutputIterator transform(InputIterator first, InputIterator last,
                              OutputIterator result, UnaryOperation op);

  template<class InputIterator1, class InputIterator2, class OutputIterator,
           class BinaryOperation>
    OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                              InputIterator2 first2, OutputIterator result,
                              BinaryOperation binary_op);

  template<class ForwardIterator, class T>
    void replace(ForwardIterator first, ForwardIterator last,
                 const T& old_value, const T& new_value);

  template<class ForwardIterator, class Predicate, class T>
    void replace_if(ForwardIterator first, ForwardIterator last,
                    Predicate pred, const T& new_value);

  template<class InputIterator, class OutputIterator, class T>
    OutputIterator replace_copy(InputIterator first, InputIterator last,
                                 OutputIterator result,
                                 const T& old_value, const T& new_value);

  template<class InputIterator, class OutputIterator, class Predicate, class T>
    OutputIterator replace_copy_if(InputIterator first, InputIterator last,
                                     OutputIterator result,
                                     Predicate pred, const T& new_value);

  template<class ForwardIterator, class T>
    void fill(ForwardIterator first, ForwardIterator last, const T& value);

  template<class OutputIterator, class Size, class T>
OutputIterator fill_n(OutputIterator first, Size n, const T& value);

template<class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last,
             Generator gen);

template<class OutputIterator, class Size, class Generator>
OutputIterator generate_n(OutputIterator first, Size n, Generator gen);

template<class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                      const T& value);

template<class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                         Predicate pred);

template<class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first, InputIterator last,
                         OutputIterator result, const T& value);

template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                               OutputIterator result, Predicate pred);

template<class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                     BinaryPredicate pred);

template<class InputIterator, class OutputIterator>
OutputIterator unique_copy(InputIterator first, InputIterator last,
                         OutputIterator result);

template<class InputIterator, class OutputIterator, class BinaryPredicate>
OutputIterator unique_copy(InputIterator first, InputIterator last,
                         OutputIterator result, BinaryPredicate pred);

template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);

template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
                         BidirectionalIterator last, OutputIterator result);

template<class ForwardIterator>
ForwardIterator rotate(ForwardIterator first, ForwardIterator middle,
                   ForwardIterator last);

template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(
                   ForwardIterator first, ForwardIterator middle,
                   ForwardIterator last, OutputIterator result);

// 25.3.12, shuffle:
template<class RandomAccessIterator, class UniformRandomNumberGenerator>
void shuffle(RandomAccessIterator first,
            RandomAccessIterator last,
            UniformRandomNumberGenerator&& g);

// 25.3.13, partitions:
template <class InputIterator, class Predicate>
bool is_partitioned(InputIterator first, InputIterator last, Predicate pred);

template<class ForwardIterator, class Predicate>
ForwardIterator partition(ForwardIterator first,
ForwardIterator last,
Predicate pred);

template<class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition(BidirectionalIterator first,
BidirectionalIterator last,
Predicate pred);

template <class InputIterator, class OutputIterator1,
class OutputIterator2, class Predicate>
pair<OutputIterator1, OutputIterator2>
partition_copy(InputIterator first, InputIterator last,
OutputIterator1 out_true, OutputIterator2 out_false,
Predicate pred);

template<class ForwardIterator, class Predicate>
ForwardIterator partition_point(ForwardIterator first,
ForwardIterator last,
Predicate pred);

// 25.4, sorting and related operations:
// 25.4.1, sorting:

template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);

template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);

template<class 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 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 ForwardIterator>
    bool is_sorted(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
    bool is_sorted(ForwardIterator first, ForwardIterator last, Compare comp);

template<class ForwardIterator>
    ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
    ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last, Compare comp);

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);

// 25.4.3, binary search:

template<class ForwardIterator, class T>
    ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
    ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);

template<class ForwardIterator, class T>
    ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
    ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);

template<class ForwardIterator, class T>
    pair<ForwardIterator, ForwardIterator>
        equal_range(ForwardIterator first, ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
    pair<ForwardIterator, ForwardIterator>
        equal_range(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);

template<class ForwardIterator, class T>
    bool binary_search(ForwardIterator first, ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
    bool binary_search(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);

// 25.4.4, merge:

template<class InputIterator1, class InputIterator2, class OutputIterator>
    OutputIterator merge(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);

template<class InputIterator1, class InputIterator2, class OutputIterator,
class Compare>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
        OutputIterator result, Compare comp);

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);

// 25.4.5, set operations:
template<class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool includes(
        InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2, Compare comp);

template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
        OutputIterator result); / * 893 */

template<class InputIterator1, class InputIterator2, class OutputIterator,
        class Compare>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
        OutputIterator result, Compare comp);

template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_intersection(
        InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
        OutputIterator result); / * 893 */

template<class InputIterator1, class InputIterator2, class OutputIterator,
        class Compare>
OutputIterator set_intersection(
        InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
        OutputIterator result, Compare comp);

template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_difference(
        InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
        OutputIterator result); / * 893 */

template<class InputIterator1, class InputIterator2, class OutputIterator,
        class Compare>
OutputIterator set_difference(
        InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
OutputIterator result, Compare comp);

template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_symmetric_difference(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result);

template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
OutputIterator set_symmetric_difference(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result, Compare comp);

// 25.4.6, 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>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

template<class RandomAccessIterator>
RandomAccessIterator is_heap_until(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
RandomAccessIterator is_heap_until(RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);

// 25.4.7, 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 T> constexpr pair<const T&, const T&> minmax(const T& a, const T& b);
template<class T, class Compare>
    constexpr pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
template<class T>
    constexpr pair<T, T> minmax(initializer_list<T> t);
template<class T, class Compare>
    constexpr pair<T, T> minmax(initializer_list<T> t, Compare comp);

template<class ForwardIterator>
    ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
    ForwardIterator min_element(ForwardIterator first, ForwardIterator last, Compare comp);
template<class ForwardIterator>
    ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
    ForwardIterator max_element(ForwardIterator first, ForwardIterator last, Compare comp);
template<class ForwardIterator>
    pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
    pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);

template<class InputIterator1, class InputIterator2>
    bool lexicographical_compare(
        InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class Compare>
    bool lexicographical_compare(
        InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2, Compare comp);

// 25.4.9, 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);  

```cpp
template<class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first,  
                    BidirectionalIterator last, Compare comp);
```

3 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.

4 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.

5 Throughout this Clause, the names of template parameters are used to express type requirements.

(5.1) — If an algorithm’s template parameter is named InputIterator, InputIterator1, or InputIterator2, the template argument shall satisfy the requirements of an input iterator (24.2.3).

(5.2) — If an algorithm’s template parameter is named OutputIterator, OutputIterator1, or OutputIterator2, the template argument shall satisfy the requirements of an output iterator (24.2.4).

(5.3) — If an algorithm’s template parameter is named ForwardIterator, ForwardIterator1, or ForwardIterator2, the template argument shall satisfy the requirements of a forward iterator (24.2.5).

(5.4) — If an algorithm’s template parameter is named BidirectionalIterator, BidirectionalIterator1, or BidirectionalIterator2, the template argument shall satisfy the requirements of a bidirectional iterator (24.2.6).

(5.5) — If an algorithm’s template parameter is named RandomAccessIterator, RandomAccessIterator1, or RandomAccessIterator2, the template argument shall satisfy the requirements of a random-access iterator (24.2.7).

6 If an algorithm’s Effects section says 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 (24.2).  

7 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).

8 The Predicate parameter is used whenever an algorithm expects a function object (20.9) 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 4). The function object pred shall not apply any non-constant function through the dereferenced iterator.

9 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

268 The decision whether to include a copying version was usually based on complexity considerations. When the cost of doing the operation dominates the cost of copy, the copying version is not included. For example, sort_copy is not included because the cost of sorting is much more significant, and users might as well do copy followed by sort.

§ 25.1
work correctly in the construct `binary_pred(*first1, *first2)` contextually converted to `bool` (Clause 4). `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 4). `binary_pred` shall not apply any non-constant function through the dereferenced iterators.

10 [Note: Unless otherwise specified, algorithms that take function objects as arguments are permitted to copy those function objects freely. Programmers for whom object identity is important should consider using a wrapper class that points to a noncopied implementation object such as `reference_wrapper<T>` (20.9.4), or some equivalent solution. —end note]

11 When the description of an algorithm gives an expression such as `*first == value` for a condition, the expression shall evaluate to either true or false in boolean contexts.

12 In the description of the algorithms 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

```c
    X tmp = a;
    advance(tmp, n);
    return tmp;
```

and that of `b-a` is the same as of

```c
    return distance(a, b);
```

25.2 Non-modifying sequence operations

25.2.1 All of

```c
    template <class InputIterator, class Predicate>
    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.

25.2.2 Any of

```c
    template <class InputIterator, class Predicate>
    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.

25.2.3 None of

```c
    template <class InputIterator, class Predicate>
    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.

25.2.4 For each

```c
    template<class InputIterator, class Function>
    Function for_each(InputIterator first, InputIterator last, Function f);
```
25.2.5 Find

\[ \text{template} \langle \text{class InputIterator, class T} \rangle \]
\[ \text{InputIterator find(InputIterator first, InputIterator last,} \]
\[ \text{const T & value);} \]
\[ \text{template} \langle \text{class InputIterator, class Predicate} \rangle \]
\[ \text{InputIterator find_if(InputIterator first, InputIterator last,} \]
\[ \text{Predicate pred);} \]
\[ \text{template} \langle \text{class InputIterator, class Predicate} \rangle \]
\[ \text{InputIterator find_if_not(InputIterator first, InputIterator last,} \]
\[ \text{Predicate pred);} \]

\[ \text{Returns: The first iterator i in the range [first, last) for which the following corresponding conditions hold: } *i == \text{value, pred(*i) != false, pred(*i) == false. Returns last if no such iterator is found.} \]

\[ \text{Complexity: At most last - first applications of the corresponding predicate.} \]

25.2.6 Find end

\[ \text{template} \langle \text{class ForwardIterator1, class ForwardIterator2} \rangle \]
\[ \text{ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,} \]
\[ \text{ForwardIterator2 first2, ForwardIterator2 last2);} \]
\[ \text{template} \langle \text{class ForwardIterator1, class ForwardIterator2,} \]
\[ \text{class BinaryPredicate} \rangle \]
\[ \text{ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,} \]
\[ \text{ForwardIterator2 first2, ForwardIterator2 last2,} \]
\[ \text{BinaryPredicate pred);} \]

\[ \text{Effects: Finds a subsequence of equal values in a sequence.} \]

\[ \text{Returns: The last iterator i in the range [first1, last1 - (last2 - first2)) such that for every non-negative integer n < (last2 - first2), the following corresponding conditions hold: } *(i + n) == *(first2 + n), \text{pred}(*(i + n), *(first2 + n)) != false. Returns last1 if [first2, last2) is empty or if no such iterator is found.} \]

\[ \text{Complexity: At most } (last2 - first2) * (last1 - first1 - (last2 - first2) + 1) \text{ applications of the corresponding predicate.} \]

25.2.7 Find first

\[ \text{template} \langle \text{class InputIterator, class ForwardIterator} \rangle \]
\[ \text{InputIterator} \]

§ 25.2.7
find_first_of(InputIterator first1, InputIterator last1,
             ForwardIterator first2, ForwardIterator last2);

template<class InputIterator, class ForwardIterator,
         class BinaryPredicate>
InputIterator
find_first_of(InputIterator first1, InputIterator last1,
             ForwardIterator first2, ForwardIterator last2,
             BinaryPredicate pred);

Effects: Finds an element that matches one of a set of values.

Returns: The first iterator \( i \) in the range \([\text{first1}, \text{last1})\) such that for some iterator \( j \) in the range \([\text{first2}, \text{last2})\) the following conditions hold: \( i == j, \text{pred}(i, j) \neq \text{false} \). Returns \text{last1} if \([\text{first2}, \text{last2})\) is empty or if no such iterator is found.

Complexity: At most \((\text{last1}-\text{first1}) \times (\text{last2}-\text{first2})\) applications of the corresponding predicate.

### 25.2.8 Adjacent find

[alg.adjacent.find]

template<class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last,
                              BinaryPredicate pred);

Returns: The first iterator \( i \) such that both \( i \) and \( i + 1 \) are in the range \([\text{first}, \text{last})\) for which the following corresponding conditions hold: \( i == *i + 1, \text{pred}(i, *(i + 1)) \neq \text{false} \). Returns \text{last} if no such iterator is found.

Complexity: For a nonempty range, exactly \( \min((i - \text{first}) + 1, (\text{last} - \text{first}) - 1) \) applications of the corresponding predicate, where \( i \) is \text{adjacent_find}'s return value.

### 25.2.9 Count

[alg.count]

template<class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& value);

template<class InputIterator, class Predicate>
typename iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred);

Effects: Returns the number of iterators \( i \) in the range \([\text{first}, \text{last})\) for which the following corresponding conditions hold: \( i == \text{value}, \text{pred}(i) \neq \text{false} \).

Complexity: Exactly \( \text{last} - \text{first} \) applications of the corresponding predicate.

### 25.2.10 Mismatch

[mismatch]

template<class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
         InputIterator2 first2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2);

template <class InputIterator1, class InputIterator2,
class BinaryPredicate>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
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 i and j such that j == first2 + (i - first1) and i is the first iterator in the range [first1, last1) for which the following corresponding conditions hold:
   (2.1)  j is in the range [first2, last2).
   (2.2)  !(i == *(first2 + (i - first1)))
   (2.3)  pred(*i, *(first2 + (i - first1))) == false
Returs the pair first1 + min(last1 - first1, last2 - first2) and first2 + min(last1 - first1, last2 - first2) if such an iterator i is not found.
3 Complexity: At most min(last1 - first1, last2 - first2) applications of the corresponding predicate.

25.2.11 Equal [alg.equal]

template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2);

template<class InputIterator1, class InputIterator2,
class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2,
class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
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.
Complexity: No applications of the corresponding predicate if \texttt{InputIterator1} and \texttt{InputIterator2} meet the requirements of random access iterators and \texttt{last1 - first1} != \texttt{last2 - first2}. Otherwise, at most \(\min(last1 - first1, last2 - first2)\) applications of the corresponding predicate.

25.2.12 \textbf{Is permutation} \hfill [\texttt{alg.is_permutation}]

\begin{verbatim}
template<class ForwardIterator1, class ForwardIterator2>
bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, BinaryPredicate pred);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, ForwardIterator2 last2,
                   BinaryPredicate pred);
\end{verbatim}

\textbf{Requires:} \texttt{ForwardIterator1} and \texttt{ForwardIterator2} shall have the same value type. The comparison function shall be an equivalence relation.

\textbf{Remarks:} If \texttt{last2} was not given in the argument list, it denotes \texttt{first2 + (last1 - first1)} below.

\textbf{Returns:} If \texttt{last1 - first1} != \texttt{last2 - first2}, return \texttt{false}. Otherwise return \texttt{true} if there exists a permutation of the elements in the range \([\texttt{first2, first2 + (last1 - first1)}]\), beginning with \texttt{ForwardIterator2 begin}, such that \texttt{equal(first1, last1, begin)} returns \texttt{true} or \texttt{equal(first1, last1, begin, pred)} returns \texttt{true}; otherwise, returns \texttt{false}.

\textbf{Complexity:} No applications of the corresponding predicate if \texttt{ForwardIterator1} and \texttt{ForwardIterator2} meet the requirements of random access iterators and \texttt{last1 - first1} != \texttt{last2 - first2}. Otherwise, exactly \(\texttt{distance(first1, last1)}\) applications of the corresponding predicate if \texttt{equal(first1, last1, first2, last2)} would return \texttt{true} if \texttt{pred} was not given in the argument list or \texttt{equal(first1, last1, first2, last2, pred)} would return \texttt{true} if \texttt{pred} was given in the argument list; otherwise, at worst \(O(N^2)\), where \(N\) has the value \(\texttt{distance(first1, last1)}\).

25.2.13 \textbf{Search} \hfill [\texttt{alg.search}]

\begin{verbatim}
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, ForwardIterator2 last2,
                   BinaryPredicate pred);
\end{verbatim}

\textbf{Effects:} Finds a subsequence of equal values in a sequence.

\textbf{Returns:} The first iterator \(i\) in the range \([\texttt{first1, last1 - (last2-first2)}]\) such that for every non-negative integer \(n\) less than \(\texttt{last2 - first2}\) the following corresponding conditions hold: \(*(i +
n) == *(first2 + n), pred(*(i + n), *(first2 + n)) != false. Returns first1 if [first2, last2) is empty, otherwise returns last1 if no such iterator is found.

**Complexity:** At most \((\text{last1} - \text{first1}) \times (\text{last2} - \text{first2})\) applications of the corresponding predicate.

```cpp
template<class ForwardIterator, class Size, class T>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last, Size count, const T& value);
```

```cpp
template<class ForwardIterator, class Size, class T, class BinaryPredicate>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last, Size count, const T& value, BinaryPredicate pred);
```

**Requires:** The type \(\text{Size}\) shall be convertible to integral type (4.7, 12.3).

**Effects:** Finds a subsequence of equal values in a sequence.

**Returns:** The first iterator \(i\) in the range \([\text{first},\text{last-count})\) such that for every non-negative integer \(n\) less than \(\text{count}\) the following corresponding conditions hold: \(*(i + n) = \text{value}, \text{pred}(*(i + n),\text{value}) = false. \) Returns last if no such iterator is found.

**Complexity:** At most \(\text{last} - \text{first}\) applications of the corresponding predicate.

### 25.3 Mutating sequence operations

#### 25.3.1 Copy

```cpp
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result);
```

**Effects:** Copies elements in the range \([\text{first},\text{last})\) into the range \([\text{result},\text{result} + (\text{last} - \text{first}))\) starting from \text{first} and proceeding to \text{last}. For each non-negative integer \(n < (\text{last} - \text{first})\), performs \(*(\text{result} + n) = *(\text{first} + n).\)

**Returns:** \(\text{result} + (\text{last} - \text{first}).\)

**Requires:** \text{result} shall not be in the range \([\text{first},\text{last}).\)

**Complexity:** Exactly \(\text{last} - \text{first}\) assignments.

```cpp
template<class InputIterator, class Size, class OutputIterator>
OutputIterator copy_n(InputIterator first, Size n, OutputIterator result);
```

**Effects:** For each non-negative integer \(i < n\), performs \(*(\text{result} + i) = *(\text{first} + i).\)

**Returns:** \(\text{result} + n.\)

**Complexity:** Exactly \(n\) assignments.

```cpp
template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator copy_if(InputIterator first, InputIterator last, OutputIterator result, Predicate pred);
```

**Requires:** The ranges \([\text{first},\text{last})\) and \([\text{result},\text{result} + (\text{last} - \text{first}))\) shall not overlap.

**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 last - first applications of the corresponding predicate.
Remarks: Stable (17.6.5.7).

template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2
    copy_backward(BidirectionalIterator1 first,
               BidirectionalIterator1 last,
               BidirectionalIterator2 result);
Effects: Copies 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) = *(last - n).
Requires: result shall not be in the range [first,last].
Returns: result - (last - first).
Complexity: Exactly last - first assignments.

25.3.2 Move

template<class InputIterator, class OutputIterator>
OutputIterator move(InputIterator first, InputIterator last,
               OutputIterator result);
Effects: Moves 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) = std::move(*(first + n)).
Returns: result + (last - first).
Requires: result shall not be in the range [first,last).
Complexity: Exactly last - first move assignments.

template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2
    move_backward(BidirectionalIterator1 first,
               BidirectionalIterator1 last,
               BidirectionalIterator2 result);
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)).
Requires: result shall not be in the range [first,last].
Returns: result - (last - first).
Complexity: Exactly last - first assignments.

25.3.3 swap

template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2
    swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
               ForwardIterator2 first2);

269) copy_backward should be used instead of copy when last is in the range [result - (last - first),result).
270) move_backward should be used instead of move when last is in the range [result - (last - first),result).
Effects: For each non-negative integer \( n < (\text{last1} - \text{first1}) \) performs: swap(*(\text{first1} + n), *(\text{first2} + n)).

Requires: The two ranges \([\text{first1}, \text{last1})\) and \([\text{first2}, \text{first2} + (\text{last1} - \text{first1}))\) shall not overlap. *(\text{first1} + n) shall be swappable with \((17.6.3.2)\) *(\text{first2} + n).

Returns: \(\text{first2} + (\text{last1} - \text{first1})\).

Complexity: Exactly \(\text{last1} - \text{first1}\) 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 \((17.6.3.2)\) *b.

### 25.3.4 Transform

[alg.transform]

template<class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator
transform(InputIterator first, InputIterator last, OutputIterator result, UnaryOperation op);

template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryOperation>
OutputIterator
transform(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, OutputIterator result, BinaryOperation binary_op);

Effects: Assigns through every iterator \(i\) in the range \([\text{result}, \text{result} + (\text{last1} - \text{first1}))\) a new corresponding value equal to \(\text{op}(*(\text{first1} + (i - \text{result})))\) or \(\text{binary_op}(*(\text{first1} + (i - \text{result})), *(\text{first2} + (i - \text{result})))\).

Requires: \(\text{op}\) and \(\text{binary_op}\) shall not invalidate iterators or subranges, or modify elements in the ranges \([\text{first1}, \text{last1})\), \([\text{first2}, \text{first2} + (\text{last1} - \text{first1}))\), and \([\text{result}, \text{result} + (\text{last1} - \text{first1}))\].

Returns: \(\text{result} + (\text{last1} - \text{first1})\).

Complexity: Exactly \(\text{last1} - \text{first1}\) applications of \(\text{op}\) or \(\text{binary_op}\).

Remarks: \(\text{result}\) may be equal to \(\text{first}\) in case of unary transform, or to \(\text{first1}\) or \(\text{first2}\) in case of binary transform.

### 25.3.5 Replace

[alg.replace]

template<class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last, const T& old_value, const T& new_value);

template<class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last, Predicate pred, const T& new_value);

Requires: The expression *\(\text{first} = \text{new_value}\) shall be valid.

---

271) The use of fully closed ranges is intentional.
Effects: Substitutes elements referred by the iterator \(i\) in the range \([\text{first}, \text{last})\) with \texttt{new\_value}, when the following corresponding conditions hold: \(*i == \texttt{old\_value}, \texttt{pred}(\!*i) != \text{false}.*\)

Complexity: Exactly \(\texttt{last} - \texttt{first}\) applications of the corresponding predicate.

```cpp
template<class InputIterator, class OutputIterator, class T>
OutputIterator
replace_copy(InputIterator first, InputIterator last,
OutputIterator result,
const T& old_value, const T& new_value);
```

```cpp
template<class InputIterator, class OutputIterator, class Predicate, class T>
OutputIterator
replace_copy_if(InputIterator first, InputIterator last,
OutputIterator result,
Predicate pred, const T& new_value);
```

Requires: The results of the expressions \(*\texttt{first}\) and \texttt{new\_value}\) shall be writable to the \texttt{result}\ output iterator. The ranges \([\texttt{first}, \texttt{last})\) and \([\texttt{result}, \texttt{result} + (\texttt{last} - \texttt{first}))\) shall not overlap.

Effects: Assigns to every iterator \(i\) in the range \([\texttt{result}, \texttt{result} + (\texttt{last} - \texttt{first}))\) either \texttt{new\_value} or \(*(\texttt{first} + (i - \texttt{result}))\) depending on whether the following corresponding conditions hold:

\[ *(\texttt{first} + (i - \texttt{result})) == \texttt{old\_value} \]
\[ \texttt{pred}(*(\texttt{first} + (i - \texttt{result}))) != \text{false} \]

Returns: \texttt{result} + (\texttt{last} - \texttt{first}).

Complexity: Exactly \(\texttt{last} - \texttt{first}\) applications of the corresponding predicate.

25.3.6 Fill

```cpp
template<class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value);
```

```cpp
template<class OutputIterator, class Size, class T>
OutputIterator fill_n(OutputIterator first, Size n, const T& value);
```

Requires: The expression \texttt{value}\) shall be writable to the output iterator. The type \texttt{Size}\) shall be convertible to an integral type (4.7, 12.3).

Effects: The first algorithm assigns \texttt{value}\ through all the iterators in the range \([\texttt{first}, \texttt{last})\). The second algorithm assigns \texttt{value}\ through all the iterators in the range \([\texttt{first}, \texttt{first} + \texttt{n})\) if \texttt{n}\ is positive, otherwise it does nothing.

Returns: \texttt{fill\_n}\ returns \texttt{first} + \texttt{n}\ for non-negative values of \texttt{n}\ and \texttt{first}\ for negative values.

Complexity: Exactly \(\texttt{last} - \texttt{first}, \texttt{n}, \text{or 0}\) assignments, respectively.

25.3.7 Generate

```cpp
template<class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last,
Generator gen);
```

```cpp
template<class OutputIterator, class Size, class Generator>
OutputIterator generate_n(OutputIterator first, Size n, Generator gen);
```

§ 25.3.7
Effects: The first algorithm invokes the function object \texttt{gen} and assigns the return value of \texttt{gen} through all the iterators in the range \texttt{[first, last)}. The second algorithm invokes the function object \texttt{gen} and assigns the return value of \texttt{gen} through all the iterators in the range \texttt{[first, first + n)} if \texttt{n} is positive, otherwise it does nothing.

Requires: \texttt{gen} takes no arguments, \texttt{Size} shall be convertible to an integral type (4.7, 12.3).

Returns: \texttt{generate_n} returns \texttt{first + n} for non-negative values of \texttt{n} and \texttt{first} for negative values.

Complexity: Exactly \texttt{last - first}, \texttt{n}, or 0 invocations of \texttt{gen} and assignments, respectively.

25.3.8 Remove

\texttt{template<class ForwardIterator, class T>}
\texttt{ForwardIterator remove(ForwardIterator first, ForwardIterator last,}
\texttt{const T& value);}\
\texttt{template<class ForwardIterator, class Predicate>}
\texttt{ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,}
\texttt{Predicate pred);}\

Requires: The type of \texttt{*first} shall satisfy the \texttt{MoveAssign} requirements (Table 22).

Effects: Eliminates all the elements referred to by iterator \texttt{i} in the range \texttt{[first, last)} for which the following corresponding conditions hold: \texttt{*i == value, pred(*i) != false}.

Returns: The end of the resulting range.

Remarks: Stable (17.6.5.7).

Note: each element in the range \texttt{[ret, last)}, where \texttt{ret} is the returned value, has a valid but unspecified state, because the algorithms can eliminate elements by moving from elements that were originally in that range.

\texttt{template<class InputIterator, class OutputIterator, class T>}
\texttt{OutputIterator remove_copy(InputIterator first, InputIterator last,}
\texttt{OutputIterator result, const T& value);}\
\texttt{template<class InputIterator, class OutputIterator, class Predicate>}
\texttt{OutputIterator remove_copy_if(InputIterator first, InputIterator last,}
\texttt{OutputIterator result, Predicate pred);}\

Requires: The ranges \texttt{[first,last)} and \texttt{[result,result + (last - first))} shall not overlap. The expression \texttt{*result = *first} shall be valid.

Effects: Copies all the elements referred to by the iterator \texttt{i} in the range \texttt{[first,last)} for which the following corresponding conditions do not hold: \texttt{*i == value, pred(*i) != false}.

Returns: The end of the resulting range.

Complexity: Exactly \texttt{last - first} applications of the corresponding predicate.

Remarks: Stable (17.6.5.7).

25.3.9 Unique

\texttt{template<class ForwardIterator>}
\texttt{ForwardIterator unique(ForwardIterator first, ForwardIterator last);}
template<class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
   BinaryPredicate pred);

Effects: For a nonempty range, eliminates all but the first element from every consecutive group of
   equivalent elements referred to by the iterator i in the range [first + 1,last) for which the following
   conditions hold: *(i - 1) == *i or pred(*(i - 1), *i) != false.

Requires: The comparison function shall be an equivalence relation. The type of *first shall satisfy the MoveAssignable requirements (Table 22).

Returns: The end of the resulting range.

Complexity: For nonempty ranges, exactly (last - first) - 1 applications of the corresponding predicate.

template<class InputIterator, class OutputIterator>
OutputIterator unique_copy(InputIterator first, InputIterator last,
   OutputIterator result);

template<class InputIterator, class OutputIterator,
class BinaryPredicate>
OutputIterator unique_copy(InputIterator first, InputIterator last,
   OutputIterator result, BinaryPredicate pred);

Requires: The comparison function shall be an equivalence relation. The ranges [first,last) and
   [result,result+(last-first)) shall not overlap. The expression *result = *first shall be valid.
   If neither InputIterator nor OutputIterator meets the requirements of forward iterator then the
   value type of InputIterator shall be CopyConstructible (Table 21) and CopyAssignable (Table 23).
   Otherwise CopyConstructible is not required.

Effects: Copies only the first element from every consecutive group of equal elements referred to by
   the iterator i in the range [first,last) for which the following corresponding conditions hold: *i
   == *(i - 1) or pred(*i, *(i - 1)) != false.

Returns: The end of the resulting range.

Complexity: For nonempty ranges, exactly last - first - 1 applications of the corresponding predicate.

25.3.10 Reverse [alg.reverse]

template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);

Effects: For each non-negative integer i < (last - first)/2, applies iter_swap to all pairs of iter-
   ators first + i, (last - i) - 1.

Requires: *first shall be swappable (17.6.3.2).

Complexity: Exactly (last - first)/2 swaps.

template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
   BidirectionalIterator last, OutputIterator result);
Effects: Copies the range \([\text{first}, \text{last})\) to the range \([\text{result}, \text{result} + (\text{last} - \text{first}))\) such that for every non-negative integer \(i < (\text{last} - \text{first})\) the following assignment takes place: 
\[ *(\text{result} + (\text{last} - \text{first}) - 1 - i) = *(\text{first} + i) \].

Requires: The ranges \([\text{first}, \text{last})\) and \([\text{result}, \text{result} + (\text{last} - \text{first}))\) shall not overlap.

Returns: \(\text{result} + (\text{last} - \text{first})\).

Complexity: Exactly \(\text{last} - \text{first}\) assignments.

25.3.11 Rotate

\[ \text{alg.rotate} \]

```cpp
template<class ForwardIterator>
ForwardIterator rotate(ForwardIterator first, ForwardIterator middle,
                      ForwardIterator last);
```

Effects: For each non-negative integer \(i < (\text{last} - \text{first})\), places the element from the position \(\text{first} + i\) into position \(\text{first} + (i + (\text{last} - \text{middle})) \% (\text{last} - \text{first})\).

Returns: \(\text{first} + (\text{last} - \text{middle})\).

Remarks: This is a left rotate.

Requires: \([\text{first}, \text{middle})\) and \([\text{middle}, \text{last})\) shall be valid ranges. \(\text{ForwardIterator}\) shall satisfy the requirements of \(\text{ValueSwappable}\) (17.6.3.2). The type of \(*\text{first}\) shall satisfy the requirements of \(\text{MoveConstructible}\) (Table 20) and the requirements of \(\text{MoveAssignable}\) (Table 22).

Complexity: At most \(\text{last} - \text{first}\) swaps.

```cpp
template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
                          ForwardIterator last, OutputIterator result);
```

Effects: Copies the range \([\text{first}, \text{last})\) to the range \([\text{result}, \text{result} + (\text{last} - \text{first}))\) such that for each non-negative integer \(i < (\text{last} - \text{first})\) the following assignment takes place: 
\[ *(\text{result} + i) = *(\text{first} + (i + (\text{middle} - \text{first})) \% (\text{last} - \text{first})) \].

Returns: \(\text{result} + (\text{last} - \text{first})\).

Requires: The ranges \([\text{first}, \text{last})\) and \([\text{result}, \text{result} + (\text{last} - \text{first}))\) shall not overlap.

Complexity: Exactly \(\text{last} - \text{first}\) assignments.

25.3.12 Shuffle

\[ \text{alg.random.shuffle} \]

```cpp
template<class RandomAccessIterator, class UniformRandomNumberGenerator>
void shuffle(RandomAccessIterator first,
             RandomAccessIterator last,
             UniformRandomNumberGenerator&& g);
```

Effects: Permutes the elements in the range \([\text{first}, \text{last})\) such that each possible permutation of those elements has equal probability of appearance.

Requires: \(\text{RandomAccessIterator}\) shall satisfy the requirements of \(\text{ValueSwappable}\) (17.6.3.2). The type \(\text{UniformRandomNumberGenerator}\) shall meet the requirements of a uniform random number generator (26.5.1.3) type whose return type is convertible to \(\text{iterator_traits}<\text{RandomAccessIterator}>::\text{difference_type}\).

Complexity: Exactly \((\text{last} - \text{first}) - 1\) swaps.

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.
template <class InputIterator, class Predicate>
bool is_partitioned(InputIterator first, InputIterator last, Predicate pred);

1 Requires: InputIterator’s value type shall be convertible to Predicate’s argument type.
2 Returns: true if [first, last) is empty or if [first, last) is partitioned by pred, i.e. if all elements 
that satisfy pred appear before those that do not.
3 Complexity: Linear. At most last - first applications of pred.

template<class ForwardIterator, class Predicate>
ForwardIterator
partition(ForwardIterator first, 
ForwardIterator last, Predicate pred);

4 Effects: Places all the elements in the range [first, last) that satisfy pred before all the elements 
that do not satisfy it.
5 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.
6 Requires: ForwardIterator shall satisfy the requirements of ValueSwappable (17.6.3.2).
7 Complexity: If ForwardIterator meets the requirements for a BidirectionalIterator, at most (last - 
first) / 2 swaps are done; otherwise at most last - first swaps are done. Exactly last - first 
applications of the predicate are done.

template<class BidirectionalIterator, class Predicate>
BidirectionalIterator
stable_partition(BidirectionalIterator first, 
BidirectionalIterator last, Predicate pred);

8 Effects: Places all the elements in the range [first, last) that satisfy pred before all the elements 
that do not satisfy it.
9 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.
10 Requires: BidirectionalIterator shall satisfy the requirements of ValueSwappable (17.6.3.2). The 
type of *first shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable 
(Table 22).
11 Complexity: At most (last - first) * log(last - first) swaps, but only linear number of swaps 
if there is enough extra memory. Exactly last - first applications of the predicate.

template <class InputIterator, class OutputIterator1, 
class OutputIterator2, class Predicate>
pair<OutputIterator1, OutputIterator2>
partition_copy(InputIterator first, InputIterator last, 
OutputIterator1 out_true, OutputIterator2 out_false, 
Predicate pred);

12 Requires: InputIterator’s value type shall be CopyAssignable, and shall be writable to the out_true 
and out_false OutputIterators, and shall be convertible to Predicate’s argument type. The input 
range shall not overlap with either of the output ranges.
13 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.

§ 25.3.13
Returns: A pair \( p \) such that \( p\.first \) is the end of the output range beginning at \( \text{out\_true} \) and \( p\.second \) is the end of the output range beginning at \( \text{out\_false} \).

Complexity: Exactly \( \text{last} - \text{first} \) applications of \( \text{pred} \).

\[
\text{template<class ForwardIterator, class Predicate>}
\]
\[
\text{ForwardIterator partition_point(ForwardIterator first, ForwardIterator last, Predicate pred);}\]

Requires: \( \text{ForwardIterator}'s \) value type shall be convertible to \( \text{Predicate}'s \) argument type. \([\text{first}, \text{last}) \) shall be partitioned by \( \text{pred} \), i.e. all elements that satisfy \( \text{pred} \) shall appear before those that do not.

Returns: An iterator \( \text{mid} \) such that \( \text{all\_of(first, mid, pred)} \) and \( \text{none\_of(mid, last, pred)} \) are both true.

Complexity: \( O(\log(\text{last} - \text{first})) \) applications of \( \text{pred} \).

25.4 Sorting and related operations [alg.sorting]

1 All the operations in 25.4 have two versions: one that takes a function object of type \( \text{Compare} \) and one that uses an \( \text{operator}< \).

2 \( \text{Compare} \) is a function object type (20.9). The return value of the function call operation applied to an object of type \( \text{Compare} \), when contextually converted to \( \text{bool} \) (Clause 4), yields \( \text{true} \) if the first argument of the call is less than the second, and \( \text{false} \) otherwise. \( \text{Compare comp} \) is used throughout for algorithms assuming an ordering relation. It is assumed that \( \text{comp} \) will not apply any non-constant function through the dereferenced iterator.

3 For all algorithms that take \( \text{Compare} \), there is a version that uses \( \text{operator}< \) instead. That is, \( \text{comp(*i, *j)} != \text{false} \) defaults to \( *i < *j != \text{false} \). For algorithms other than those described in 25.4.3 to work correctly, \( \text{comp} \) has to induce a strict weak ordering on the values.

4 The term \( \text{strict} \) refers to the requirement of an irreflexive relation (\( !\text{comp}(x, x) \) for all \( x \)), and the term \( \text{weak} \) to requirements that are not as strong as those for a total ordering, but stronger than those for a partial ordering. If we define \( \text{equiv}(a, b) \) as \( !\text{comp}(a, b) \&\& !\text{comp}(b, a) \), then the requirements are that \( \text{comp} \) and \( \text{equiv} \) both be transitive relations:

\[
\begin{align*}
\text{(4.1)} & \quad \text{comp}(a, b) \&\& \text{comp}(b, c) \implies \text{comp}(a, c) \\
\text{(4.2)} & \quad \text{equiv}(a, b) \&\& \text{equiv}(b, c) \implies \text{equiv}(a, c) \quad \text{[Note: Under these conditions, it can be shown that]} \\
\text{(4.2.1)} & \quad \text{equiv} \text{ is an equivalence relation} \\
\text{(4.2.2)} & \quad \text{comp} \text{ induces a well-defined relation on the equivalence classes determined by equiv} \\
\text{(4.2.3)} & \quad \text{The induced relation is a strict total ordering. — end note]}
\end{align*}
\]

5 A sequence is \( \text{sorted with respect to a comparator} \ \text{comp} \) 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, \( \text{comp}(*(i + n), *i) == \text{false} \).

6 A sequence \( [\text{start,finish}) \) is \( \text{partitioned with respect to an expression} \ f(e) \) if there exists an integer \( n \) such that for all \( 0 <= i < \text{distance(}\text{start, finish}), f(*(\text{start} + i)) \) is true if and only if \( i < n \).

7 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 \( \text{operator==} \), but an equivalence relation induced by the strict weak ordering. That is, two elements \( a \) and \( b \) are considered equivalent if and only if \( !(a < b) \&\& !(b < a) \).

§ 25.4
25.4.1 Sorting

25.4.1.1 sort

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last,
         Compare comp);
```

1 Effects: Sorts the elements in the range [first, last).

2 Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (17.6.3.2). The type of *first shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable (Table 22).

3 Complexity: \( \Theta(N \log(N)) \) (where \( N = last - first \)) comparisons.

25.4.1.2 stable_sort

```
template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

1 Effects: Sorts the elements in the range [first, last).

2 Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (17.6.3.2). The type of *first shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable (Table 22).

3 Complexity: It does at most \( N \log^2(N) \) (where \( N = last - first \)) comparisons; if enough extra memory is available, it is \( N \log(N) \).

4 Remarks: Stable (17.6.5.7).

25.4.1.3 partial_sort

```
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);
```

1 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.

2 Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (17.6.3.2). The type of *first shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable (Table 22).

3 Complexity: It takes approximately \( (last - first) \times \log(middle - first) \) comparisons.
25.4.1.4 partial_sort_copy

```cpp
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);
```

1 Effects: Places the first \( \min(last - first, result_last - result_first) \) sorted elements into the range \([result_first, result_first + \min(last - first, result_last - result_first))\).

2 Returns: The smaller of: \(result_last\) or \(result_first + (last - first)\).

3 Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (17.6.3.2). The type of \(*result_first\) shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable (Table 22).

4 Complexity: Approximately \((last - first) \times \log(\min(last - first, result_last - result_first))\) comparisons.

25.4.1.5 is_sorted

```cpp
template<class ForwardIterator>
    bool is_sorted(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
    bool is_sorted(ForwardIterator first, ForwardIterator last,
                   Compare comp);
```

1 Returns: \(\text{is_sorted} until(first, last) == last\)

2 Returns: \(\text{is_sorted} until(first, last, comp) == last\)

3 Returns: If \(\text{distance(first, last)} < 2\), returns \(last\). Otherwise, returns the last iterator \(i\) in \([first, last]\) for which the range \([first, i)\) is sorted.

4 Complexity: Linear.

25.4.2 Nth element

```cpp
template<class RandomAccessIterator>
    void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                     RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
    void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                     RandomAccessIterator last, Compare comp);
```
After \texttt{nth\_element} the element in the position pointed to by \texttt{nth} is the element that would be in that position if the whole range were sorted, unless \texttt{nth} == \texttt{last}. Also for every iterator \texttt{i} in the range \([	exttt{first}, \texttt{nth})\) and every iterator \texttt{j} in the range \([\texttt{nth}, \texttt{last})\) it holds that: \(!(*\texttt{j} < *\texttt{i})\) or \(\text{comp(*j, *i)} == \text{false}\).

\textit{Requires:} RandomAccessIterator shall satisfy the requirements of ValueSwappable (17.6.3.2). The type of \texttt{*first} shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable (Table 22).

\textit{Complexity:} Linear on average.

### 25.4.3 Binary search

All of the algorithms in this section 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.

#### 25.4.3.1 lower\_bound

\begin{verbatim}
template<class ForwardIterator, class T>
ForwardIterator
lower_bound(ForwardIterator first, ForwardIterator last,
const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator
lower_bound(ForwardIterator first, ForwardIterator last,
const T& value, Compare comp);
\end{verbatim}

\textit{Requires:} The elements \(e\) of \([\texttt{first}, \texttt{last})\) shall be partitioned with respect to the expression \(e < \texttt{value}\) or \(\text{comp(e, value)}\).

\textit{Returns:} The furthermost iterator \(i\) in the range \([\texttt{first}, \texttt{last})\) such that for every iterator \(j\) in the range \([\texttt{first}, i)\) the following corresponding conditions hold: \(*j < \texttt{value}\) or \(\text{comp(*j, \texttt{value}) != \text{false}}\).

\textit{Complexity:} At most \(\log_2(\texttt{last} - \texttt{first}) + O(1)\) comparisons.

#### 25.4.3.2 upper\_bound

\begin{verbatim}
template<class ForwardIterator, class T>
ForwardIterator
upper_bound(ForwardIterator first, ForwardIterator last,
const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator
upper_bound(ForwardIterator first, ForwardIterator last,
const T& value, Compare comp);
\end{verbatim}

\textit{Requires:} The elements \(e\) of \([\texttt{first}, \texttt{last})\) shall be partitioned with respect to the expression \(! (\texttt{value} < e)\) or \(!\text{comp(value, e)}\).

\textit{Returns:} The furthermost iterator \(i\) in the range \([\texttt{first}, \texttt{last})\) such that for every iterator \(j\) in the range \([\texttt{first}, i)\) the following corresponding conditions hold: \(! (\texttt{value} < *j)\) or \(\text{comp(value, *j) == false}\).
Complexity: At most $\log_2(\text{last} - \text{first}) + \Theta(1)$ comparisons.

25.4.3.3 equal_range

\[
\text{equal_range}(\text{ForwardIterator first}, \text{ForwardIterator last}, \text{const T& value});
\]

\[
\text{equal_range}(\text{ForwardIterator first}, \text{ForwardIterator last}, \text{const T& value}, \text{Compare comp});
\]

Requires: The elements $e$ of $[\text{first}, \text{last})$ shall be partitioned with respect to the expressions $e < \text{value}$ and $!(\text{value} < e)$ or $\text{comp}(e, \text{value})$ and $!\text{comp}(\text{value}, e)$. Also, for all elements $e$ of $[\text{first}, \text{last})$, $e < \text{value}$ shall imply $!(\text{value} < e)$ or $\text{comp}(e, \text{value})$ shall imply $!\text{comp}(\text{value}, e)$.

Returns:

make_pair(lower_bound(first, last, value),
          upper_bound(first, last, value))

or

make_pair(lower_bound(first, last, value, comp),
          upper_bound(first, last, value, comp))

Complexity: At most $2 \times \log_2(\text{last} - \text{first}) + \Theta(1)$ comparisons.

25.4.3.4 binary_search

\[
\text{binary_search}(\text{ForwardIterator first}, \text{ForwardIterator last}, \text{const T& value});
\]

\[
\text{binary_search}(\text{ForwardIterator first}, \text{ForwardIterator last}, \text{const T& value}, \text{Compare comp});
\]

Requires: The elements $e$ of $[\text{first}, \text{last})$ are partitioned with respect to the expressions $e < \text{value}$ and $!(\text{value} < e)$ or $\text{comp}(e, \text{value})$ and $!\text{comp}(\text{value}, e)$. Also, for all elements $e$ of $[\text{first}, \text{last})$, $e < \text{value}$ implies $!(\text{value} < e)$ or $\text{comp}(e, \text{value})$ implies $!\text{comp}(\text{value}, e)$.

Returns: true if there is an iterator $i$ in the range $[\text{first}, \text{last})$ that satisfies the corresponding conditions: $!(\ast i < \text{value}) \&\& !(\text{value} < \ast i)$ or $\text{comp}(\ast i, \text{value}) == \text{false} \&\& \text{comp}(\text{value}, \ast i) == \text{false}$.

Complexity: At most $\log_2(\text{last} - \text{first}) + \Theta(1)$ comparisons.

25.4.4 Merge

\[
\text{merge}(\text{InputIterator1 first1, InputIterator1 last1,}
          \text{InputIterator2 first2, InputIterator2 last2},
          \text{OutputIterator});
\]

§ 25.4.4
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator
merge(InputIterator1 first1, InputIterator1 last1,
      InputIterator2 first2, InputIterator2 last2,
      OutputIterator result, Compare comp);
1
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.
2
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.
3
Returns: result + (last1 - first1) + (last2 - first2).
4
Complexity: At most (last1 - first1) + (last2 - first2) - 1 comparisons.
5
Remarks: Stable (17.6.5.7).

template<class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first,
                   BidirectionalIterator middle,
                   BidirectionalIterator last);
6
Effects: Merges two sorted consecutive ranges [first,middle) and [middle,last), putting the result
of the merge into the range [first,last). The resulting range will be in non-decreasing order; that is,
for every iterator i in [first,last) other than first, the condition *i < *(i - 1) or, respectively,
comp(*i, *(i - 1)) will be false.
7
Requires: The ranges [first,middle) and [middle,last) shall be sorted with respect to operator< or comp. BidirectionalIterator shall satisfy the requirements of ValueSwappable (17.6.3.2). The type of *first shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable (Table 22).
8
Complexity: When enough additional memory is available, (last - first) - 1 comparisons. If no
additional memory is available, an algorithm with complexity \(N \log(N)\) (where \(N\) is equal to last -
first) may be used.
9
Remarks: Stable (17.6.5.7).

25.4.5 Set operations on sorted structures

This section defines all the basic set operations on sorted structures. They also work with multisets (23.4.7)
containing multiple copies of equivalent elements. The semantics of the set operations are generalized to
multisets in a standard way by defining set_union() to contain the maximum number of occurrences of
every element, set_intersection() to contain the minimum, and so on.

25.4.5.1 includes

template<class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              Compare comp);

1 Returns: true if [first2,last2) is empty or if every element in the range [first2,last2) is con-
tained in the range [first1,last1). Returns false otherwise.
2 Complexity: At most 2 * ((last1 - first1) + (last2 - first2)) - 1 comparisons.

25.4.5.2 set_union

template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator
  set_union(InputIterator1 first1, InputIterator1 last1,
            InputIterator2 first2, InputIterator2 last2,
            OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator
  set_union(InputIterator1 first1, InputIterator1 last1,
            InputIterator2 first2, InputIterator2 last2,
            OutputIterator result, Compare comp);

1 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.
2 Requires: The resulting range shall not overlap with either of the original ranges.
3 Returns: The end of the constructed range.
4 Complexity: At most 2 * ((last1 - first1) + (last2 - 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.

25.4.5.3 set_intersection

template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator
  set_intersection(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2,
                  OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator
  set_intersection(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2,
                  OutputIterator result, Compare comp);

1 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.
Requires: The resulting range shall not overlap with either of the original 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.

**25.4.5.4 set_difference**

```cpp
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator
set_difference(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
OutputIterator result);
```

```cpp
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
OutputIterator
set_difference(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
OutputIterator result, Compare comp);
```

1 Effects: Copies the elements of the range \([first1, last1)\) which are not present in the range \([first2, last2)\) to the range beginning at \(result\). The elements in the constructed range are sorted.

Requires: The resulting range shall not overlap with either of the original 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 last \(\max(m - n, 0)\) elements from \([first1, last1)\) shall be copied to the output range.

**25.4.5.5 set_symmetric_difference**

```cpp
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator
set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
OutputIterator result);
```

```cpp
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
OutputIterator
set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
OutputIterator result, Compare comp);
```

1 Effects: Copies the elements of the range \([first1, last1)\) that are not present in the range \([first2, last2)\), and the elements of the range \([first2, last2)\) that are not present in the range \([first1, last1)\) to the range beginning at \(result\). The elements in the constructed range are sorted.

Requires: The resulting range shall not overlap with either of the original ranges.

Returns: The end of the constructed range.
Complexity: At most $2 \times ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1$ comparisons.

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 $m < n$.

25.4.6 Heap operations

A heap is a particular organization of elements in a range between two random access iterators $[a, b)$. Its two key properties are:

1. There is no element greater than $a$ in the range and
2. $a$ may be removed by pop_heap(), or a new element added by push_heap(), in $O(\log(N))$ time.

These properties make heaps useful as priority queues.

make_heap() converts a range into a heap and sort_heap() turns a heap into a sorted sequence.

25.4.6.1 push_heap

template<class RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

Effects: Places the value in the location last - 1 into the resulting heap [first, last).

Requires: The range [first, last - 1) shall be a valid heap. The type of *first shall satisfy the MoveConstructible requirements (Table 20) and the MoveAssignable requirements (Table 22).

Complexity: At most $\log(\text{last} - \text{first})$ comparisons.

25.4.6.2 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 [first, last) shall be a valid non-empty heap. RandomAccessIterator shall satisfy the requirements of ValueSwappable (17.6.3.2). The type of *first shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable (Table 22).

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 $2 \times \log(\text{last} - \text{first})$ comparisons.

25.4.6.3 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);
Effects: Constructs a heap out of the range \([\text{first}, \text{last})\).

Requires: The type of \(*\text{first}\) shall satisfy the `MoveConstructible` requirements (Table 20) and the `MoveAssignable` requirements (Table 22).

Complexity: At most \(3 \times (\text{last} - \text{first})\) comparisons.

25.4.6.4 `sort_heap` [sort.heap]

```cpp
template<class RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Effects: Sorts elements in the heap \([\text{first}, \text{last})\).

Requires: The range \([\text{first}, \text{last})\) shall be a valid heap. `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (17.6.3.2). The type of \(*\text{first}\) shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22).

Complexity: At most \(N \log(N)\) comparisons (where \(N = \text{last} - \text{first}\)).

25.4.6.5 `is_heap` [is.heap]

```cpp
template<class RandomAccessIterator>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Returns: \(\text{is_heap} \text{ until}(\text{first}, \text{last}) == \text{last}\)

```cpp
template<class RandomAccessIterator, class Compare>
RandomAccessIterator is_heap_until(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Returns: \(\text{is_heap} \text{ until}(\text{first}, \text{last}, \text{comp}) == \text{last}\)

Returns: If \(\text{distance}(\text{first}, \text{last}) < 2\), returns \(\text{last}\). Otherwise, returns the last iterator \(i\) in \([\text{first}, \text{last}]\) for which the range \([\text{first}, i)\) is a heap.

Complexity: Linear.

25.4.7 Minimum and maximum [alg.min.max]

```cpp
template<class T> constexpr const T& min(const T& a, const T& b);
```

```cpp
template<class T, class Compare>
constexpr const T& min(const T& a, const T& b, Compare comp);
```

Requires: Type \(T\) is `LessThanComparable` (Table 18).

Returns: The smaller value.

Remarks: Returns the first argument when the arguments are equivalent.

```cpp
template<class T>
constexpr T min(initializer_list<T> t);
```

```cpp
template<class T, class Compare>
constexpr T min(initializer_list<T> t, Compare comp);
```
Requires: $T$ is LessThanComparable and CopyConstructible and $t$.size() > 0.

Returns: The smallest value in the initializer_list.

Remarks: Returns a copy of the leftmost argument when several arguments are equivalent to the smallest.

template<class T> constexpr const T& max(const T& a, const T& b);
template<class T, class Compare>
  constexpr const T& max(const T& a, const T& b, Compare comp);

Requires: Type $T$ is LessThanComparable (Table 18).

Returns: The larger value.

Remarks: Returns the first argument when the arguments are equivalent.

template<class T>
  constexpr T max(initializer_list<T> t);
template<class T, class Compare>
  constexpr T max(initializer_list<T> t, Compare comp);

Requires: $T$ is LessThanComparable and CopyConstructible and $t$.size() > 0.

Returns: The largest value in the initializer_list.

Remarks: Returns a copy of the leftmost argument when several arguments are equivalent to the largest.

template<class T> constexpr pair<const T&, const T&> minmax(const T& a, const T& b);
template<class T, class Compare>
  constexpr pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);

Requires: Type $T$ shall be LessThanComparable (Table 18).

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.

template<class T>
  constexpr pair<T, T> minmax(initializer_list<T> t);
template<class T, class Compare>
  constexpr pair<T, T> minmax(initializer_list<T> t, Compare comp);

Requires: $T$ is LessThanComparable and CopyConstructible and $t$.size() > 0.

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.

template<class ForwardIterator>
  ForwardIterator min_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
  ForwardIterator min_element(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 \(\text{comp}(*j, *i) == \text{false}\). Returns \(last\) if \(first == last\).

Complexity: Exactly \(\max((last - first) - 1, 0)\) applications of the corresponding comparisons.

template<class ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
ForwardIterator max_element(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: !(\(*i < *j\)) or \(\text{comp}(*i, *j) == \text{false}\). Returns \(last\) if \(first == last\).

Complexity: Exactly \(\max((last - first) - 1, 0)\) applications of the corresponding comparisons.

template<class ForwardIterator>
pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);

Returns: \(\text{make_pair}(first, first)\) if \([first, last)\) is empty, otherwise \(\text{make_pair}(m, M)\), where \(m\) is the first iterator in \([first, last)\) such that no iterator in the range refers to a smaller element, and where \(M\) is the last iterator \(272\) in \([first, last)\) such that no iterator in the range refers to a larger element.

Complexity: At most \(\max(\lfloor \frac{3}{2}(N - 1)\rfloor, 0)\) applications of the corresponding predicate, where \(N\) is \(\text{distance}(first, last)\).

25.4.8 Lexicographical comparison

template<class InputIterator1, class InputIterator2>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, Compare comp);

Returns: \(\text{true}\) 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 \(\text{false}\) otherwise.

Complexity: At most \(2*\min((last1 - first1), (last2 - first2))\) applications of the corresponding comparison.

Remarks: If two sequences have the same number of elements and their corresponding elements 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.

\(272\) This behavior intentionally differs from \(\text{max_element}\).
for ( ; first1 != last1 && first2 != last2 ; ++first1, ++first2) {
    if (*first1 < *first2) return true;
    if (*first2 < *first1) return false;
}
return first1 == last1 && first2 != last2;

Remarks: An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence.

25.4.9 Permutation generators

template<class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first, BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool next_permutation(BidirectionalIterator first, BidirectionalIterator last, Compare comp);

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. If such a permutation exists, it returns true. Otherwise, it transforms the sequence into the smallest permutation, that is, the ascendingly sorted one, and returns false.

Requires: BidirectionalIterator shall satisfy the requirements of ValueSwappable (17.6.3.2).

Complexity: At most (last - first)/2 swaps.

template<class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last, Compare comp);

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.

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.

Requires: BidirectionalIterator shall satisfy the requirements of ValueSwappable (17.6.3.2).

Complexity: At most (last - first)/2 swaps.

25.5 C library algorithms

Table 113 describes some of the contents of the header <cstdlib>.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>size_t</td>
</tr>
<tr>
<td>Functions:</td>
<td>bsearch qsort</td>
</tr>
</tbody>
</table>

The contents are the same as the Standard C library header <stdlib.h> with the following exceptions:

§ 25.5
The function signature:

```c
bsearch(const void *, const void *, size_t, size_t,
        int (*)(const void *, const void *));
```

is replaced by the two declarations:

```c
extern "C" void* bsearch(const void* key, const void* base,
                          size_t nmemb, size_t size,
                          int (*compar)(const void*, const void*));
extern "C++" void* bsearch(const void* key, const void* base,
                          size_t nmemb, size_t size,
                          int (*compar)(const void*, const void*));
```

both of which have the same behavior as the original declaration.

The function signature:

```c
qsort(void *, size_t, size_t,
      int (*)(const void *, const void *));
```

is replaced by the two declarations:

```c
extern "C" void qsort(void* base, size_t nmemb, size_t size,
                      int (*compar)(const void*, const void*));
extern "C++" void qsort(void* base, size_t nmemb, size_t size,
                      int (*compar)(const void*, const void*));
```

both of which have the same behavior as the original declaration. The behavior is undefined unless the objects in the array pointed to by `base` are of trivial type.

[Note: Because the function argument `compar()` may throw an exception, `bsearch()` and `qsort()` are allowed to propagate the exception (17.6.5.12). — end note]

See also: ISO C 7.10.5.
26 Numerics library

26.1 General

This Clause describes components that C++ programs may use to perform seminumerical operations.

The following subclauses describe components for complex number types, random number generation, numeric \((n\text{-at-a-time})\) arrays, generalized numeric algorithms, and facilities included from the ISO C library, as summarized in Table 114.

Table 114 — Numerics library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>26.2</td>
<td>Requirements</td>
</tr>
<tr>
<td>26.3</td>
<td>Floating-Point Environment</td>
</tr>
<tr>
<td>26.4</td>
<td>Complex Numbers</td>
</tr>
<tr>
<td>26.5</td>
<td>Random number generation</td>
</tr>
<tr>
<td>26.6</td>
<td>Numeric arrays</td>
</tr>
<tr>
<td>26.7</td>
<td>Generalized numeric operations</td>
</tr>
<tr>
<td>26.8</td>
<td>C library</td>
</tr>
</tbody>
</table>

26.2 Numeric type requirements

The \texttt{complex} and \texttt{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:\(^{273}\)

\begin{enumerate}
\item \(T\) is not an abstract class (it has no pure virtual member functions);
\item \(T\) is not a reference type;
\item \(T\) is not \texttt{cv}-qualified;
\item If \(T\) is a class, it has a public default constructor;
\item If \(T\) is a class, it has a public copy constructor with the signature \(T::\text{operator}=(\text{const } T&)\)
\item If \(T\) is a class, it has a public destructor;
\item If \(T\) is a class, it has a public assignment operator whose signature is either \(T& T::\text{operator}=(\text{const } T&)\) or \(T& T::\text{operator}=(T)\)
\item If \(T\) is a class, its assignment operator, copy and default constructors, and destructor shall correspond to each other in the following sense: Initialization of raw storage using the default constructor, followed by assignment, is semantically equivalent to initialization of raw storage using the copy constructor.
\end{enumerate}

\(^{273}\)In other words, value types. These include arithmetic types, pointers, the library class \texttt{complex}, and instantiations of \texttt{valarray} for value types.
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 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` (23.3.6) instead of `valarray` for that class; — end note]

(1.9) — If T is a class, it does not overload unary `operator&`.

2 If any operation on T throws an exception the effects are undefined.

3 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.

4 [Example: It is valid to instantiate `valarray<complex>`, but `operator()` will not be successfully instantiated for `valarray<complex>` operands, since `complex` does not have any ordering operators. — end example]

26.3 The floating-point environment [cfenv]

26.3.1 Header `<cfenv>` synopsis [cfenv.syn]

```cpp
namespace std {
    // types
    typedef object type fenv_t;
    typedef integer type fexcept_t;

    // 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(void);
    int fesetround(int mode);

    int fegetenv(fenv_t* penv);
    int feholdexcept(fenv_t* penv);
    int fesetenv(const fenv_t* penv);
    int feupdateenv(const fenv_t* penv);
}
```

1 The header also defines the macros:

```
FE_ALL_EXCEPT
FE_DIVBYZERO
FE_INEXACT
FE_INVALID
FE_OVERFLOW
```
2 The header defines all functions, types, and macros the same as Clause 7.6 of the C standard.

3 The floating-point environment has thread storage duration (3.7.2). 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 `std::thread` object (30.3.1) 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]

4 A separate floating-point environment shall be maintained for each thread. Each function accesses the environment corresponding to its calling thread.

### 26.4 Complex numbers

The header `<complex>` defines a class template, and numerous functions for representing and manipulating complex numbers.

2 The effect of instantiating the template `complex` for any type other than `float`, `double`, or `long double` is unspecified. The specializations `complex<float>`, `complex<double>`, and `complex<long double>` are literal types (3.9).

3 If the result of a function is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

4 If `z` is an lvalue expression of type `cv std::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 std::complex<T>*` and the expression `a[i]` is well-defined for an integer expression `i`, then:

(4.4) `reinterpret_cast<cv T*>(a)[2*i]` shall designate the real part of `a[i]`, and

(4.5) `reinterpret_cast<cv T*>(a)[2*i + 1]` shall designate the imaginary part of `a[i].`

### 26.4.1 Header `<complex>` synopsis

```cpp
namespace std {
    template<class T> class complex;
    template<> class complex<float>;
    template<> class complex<double>;
    template<> class complex<long double>;

    // 26.4.6, operators:
    template<class T>
        complex<T> operator+(const complex<T>&, const complex<T>&);
    template<class T> complex<T> operator+(const complex<T>&, const T&);
    template<class T> complex<T> operator+(const T&, const complex<T>&);
}
```

§ 26.4.1
template<class T> complex<T> operator-(
    const complex<T>&, const complex<T>&);
template<class T> complex<T> operator-(const complex<T>&, const T&);
template<class T> complex<T> operator-(const T&, const complex<T>&);

template<class T> complex<T> operator*(
    const complex<T>&, const complex<T>&);
template<class T> complex<T> operator*(const complex<T>&, const T&);
template<class T> complex<T> operator*(const T&, const complex<T>&);

template<class T> complex<T> operator/(const complex<T>&, const complex<T>&);
template<class T> complex<T> operator/(const complex<T>&, const T&);
template<class T> complex<T> operator/(const T&, const complex<T>&);

template<class T> complex<T> operator+(const complex<T>&);
template<class T> 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>&);

// 26.4.7, values:
template<class T> constexpr T real(const complex<T>&);
template<class T> constexpr T imag(const complex<T>&);

template<class T> T abs(const complex<T>&);
template<class T> T arg(const complex<T>&);
template<class T> T norm(const complex<T>&);

template<class T> complex<T> conj(const complex<T>&);
template <class T> complex<T> proj(const complex<T>&);

// 26.4.8, transcendentals:
template<class T> complex<T> acos(const complex<T>&);
template<class T> complex<T> asin(const complex<T>&);
template<class T> complex<T> atan(const complex<T>&);
template<class T> complex<T> acosh(const complex<T>&);
template<class T> complex<T> asinh(const complex<T>&);
template<class T> complex<T> atanh(const complex<T>&);
template<class T> complex<T> cos (const complex<T>&);
template<class T> complex<T> cosh (const complex<T>&);
template<class T> complex<T> exp (const complex<T>&);
template<class T> complex<T> log (const complex<T>&);
template<class T> complex<T> log10(const complex<T>&);

template<class T> complex<T> pow(const complex<T>&, const T&);
template<class T> complex<T> pow(const complex<T>&, const complex<T>&);

template<class T> complex<T> pow(const T&, const complex<T>&);

template<class T> complex<T> sin (const complex<T>&);
template<class T> complex<T> sinh (const complex<T>&);

template<class T> complex<T> sqrt (const complex<T>&);
template<class T> complex<T> tan (const complex<T>&);
template<class T> complex<T> tanh (const complex<T>&);

// 26.4.10, complex literals:
inline namespace literals {
    inline namespace complex_literals {
        constexpr complex<long double> operator"il(long double);
        constexpr complex<long double> operator"il(unsigned long long);
        constexpr complex<double> operator"i(long double);
        constexpr complex<double> operator"i(unsigned long long);
        constexpr complex<float> operator"if(long double);
        constexpr complex<float> operator"if(unsigned long long);
    }
}

26.4.2 Class template complex

namespace std {
    template<class T>
    class complex {
    public:
        typedef T value_type;
        constexpr complex(const T& re = T(), const T& im = T());
        constexpr complex(const complex&);
        template<class X> constexpr complex(const complex<X>&);

        constexpr T real() const;
        void real(T);
        constexpr T imag() const;
        void imag(T);

        complex<T>& operator= (const T&);
        complex<T>& operator+=(const T&);
        complex<T>& operator-=(const T&);
        complex<T>& operator*=(const T&);
        complex<T>& operator/=(const T&);
        complex& operator=(const complex&);
        template<class X> complex<T>& operator= (const complex<X>&);
        template<class X> complex<T>& operator+=(const complex<X>&);
    }
The class `complex` describes an object that can store the Cartesian components, `real()` and `imag()`, of a complex number.

### 26.4.3 complex specializations

```cpp
namespace std {
    template<> class complex<float> {
        public:
            typedef float value_type;

            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;
            void real(float);
            constexpr float imag() const;
            void imag(float);

            complex<float>& operator= (float);
            complex<float>& operator+=(float);
            complex<float>& operator-=(float);
            complex<float>& operator*=(float);
            complex<float>& operator/=(float);
    }

    template<> class complex<double> {
        public:
            typedef double value_type;

            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;
            void real(double);
            constexpr double imag() const;
            void imag(double);

            complex<double>& operator= (double);
            complex<double>& operator+=(double);
            complex<double>& operator-=(double);
            complex<double>& operator*=(double);
            complex<double>& operator/=(double);
    }
```

§ 26.4.3
complex<double>& operator/=(double);
complex<double>& operator=(const complex<double>&);
template<class X> complex<double>& operator=(const complex<X>&);
template<class X> complex<double>& operator+=(const complex<X>&);
template<class X> complex<double>& operator-=(const complex<X>&);
template<class X> complex<double>& operator*=(const complex<X>&);
};

template<> class complex<long double> {
public:
    typedef long double value_type;
    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;
    void real(long double);
    constexpr long double imag() const;
    void imag(long double);
    complex<long double>& operator=(const complex<long double>&);
    complex<long double>& operator=(long double);
    complex<long double>& operator+=(long double);
    complex<long double>& operator-=(long double);
    complex<long double>& operator*=(long double);
    complex<long double>& operator/=(long double);
};

26.4.4 complex member functions

template<class T> constexpr complex(const T& re = T(), const T& im = T());

Effects: Constructs an object of class complex.
Postcondition: real() == re && imag() == im.
constexpr T real() const;

Returns: The value of the real component.

void real(T val);

Effects: Assigns val to the real component.

constexpr T imag() const;

Returns: The value of the imaginary component.
void imag(T val);

    *Effects: Assigns val to the imaginary component.

26.4.5 complex member operators [complex.member.ops]

complex<T>& operator+=(const T& rhs);

    *Effects: Adds the scalar value rhs to the real part of the complex value *this and stores the result in the real part of *this, leaving the imaginary part unchanged.
    *Returns: *this.

complex<T>& operator-=(const T& rhs);

    *Effects: Subtracts the scalar value rhs from the real part of the complex value *this and stores the result in the real part of *this, leaving the imaginary part unchanged.
    *Returns: *this.

complex<T>& operator*=(const T& rhs);

    *Effects: Multiplies the scalar value rhs by the complex value *this and stores the result in *this.
    *Returns: *this.

complex<T>& operator/=(const T& rhs);

    *Effects: Divides the scalar value rhs into the complex value *this and stores the result in *this.
    *Returns: *this.

template<class X> complex<T>& operator+=(const complex<X>& rhs);

    *Effects: Adds the complex value rhs to the complex value *this and stores the sum in *this.
    *Returns: *this.

template<class X> complex<T>& operator-=(const complex<X>& rhs);

    *Effects: Subtracts the complex value rhs from the complex value *this and stores the difference in *this.
    *Returns: *this.

template<class X> complex<T>& operator*=(const complex<X>& rhs);

    *Effects: Multiplies the complex value rhs by the complex value *this and stores the product in *this.
    *Returns: *this.

template<class X> complex<T>& operator/=(const complex<X>& rhs);

    *Effects: Divides the complex value rhs into the complex value *this and stores the quotient in *this.
    *Returns: *this.
26.4.6  complex non-member operations  

template<class T> complex<T> operator+(const complex<T>& lhs);

Remarks: unary operator.

Returns: complex<T>(lhs).

template<class T>
    complex<T> operator+(const complex<T>& lhs, const complex<T>& rhs);
    template<class T> complex<T> operator+(const complex<T>& lhs, const T& rhs);
    template<class T> complex<T> operator+(const T& lhs, const complex<T>& rhs);

Returns: complex<T>(lhs) += rhs.

template<class T> complex<T> operator+(const complex<T>& lhs, const T& rhs);

template<class T> complex<T> operator+(const T& lhs, const complex<T>& rhs);

Returns: complex<T>(lhs) += rhs.

Remarks: unary operator.

Returns: complex<T>(-lhs.real(),-lhs.imag()).

template<class T>
    complex<T> operator-(const complex<T>& lhs);

Remarks: unary operator.

Returns: complex<T>(-lhs.real(),-lhs.imag()).

template<class T> complex<T> operator-(const complex<T>& lhs, const complex<T>& rhs);
    template<class T> complex<T> operator-(const complex<T>& lhs, const T& rhs);
    template<class T> complex<T> operator-(const T& lhs, const complex<T>& rhs);

Returns: complex<T>(lhs) -= rhs.

template<class T>
    complex<T> operator*(const complex<T>& lhs, const complex<T>& rhs);
    template<class T> complex<T> operator*(const complex<T>& lhs, const T& rhs);
    template<class T> complex<T> operator*(const T& lhs, const complex<T>& rhs);

Returns: complex<T>(lhs) *= rhs.

template<class T, class charT, class traits>
    basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>& is, complex<T>& x);
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 (27.7.2.2).

Requires: The input values shall be convertible to \( T \).

If bad input is encountered, calls \( \text{is.setstate}(\text{ios_base::failbit}) \) (which may throw \( \text{ios::failure} \) (27.5.5.4)).

Returns: \( \text{is} \).

Remarks: This extraction is performed as a series of simpler extractions. Therefore, the skipping of whitespace is specified to be the same for each of the simpler extractions.

```
template<class T, class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& o, const complex<T>& x);
```

Effects: inserts the complex number \( x \) onto the stream \( o \) as if it were implemented as follows:

```
template<class T, class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& o, const complex<T>& x) {
  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 \( \text{std::showpoint} \) 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.

### 26.4.7 complex value operations

```
template<class T> constexpr T real(const complex<T>& x);
Returns: \( x.\text{real}() \).
```

```
template<class T> constexpr T imag(const complex<T>& x);
Returns: \( x.\text{imag}() \).
```

```
template<class T> T abs(const complex<T>& x);
Returns: The magnitude of \( x \).
```

```
template<class T> T arg(const complex<T>& x);
Returns: The phase angle of \( x \), or \( \text{atan2}(\text{imag}(x), \text{real}(x)) \).
```

```
template<class T> T norm(const complex<T>& x);
Returns: The squared magnitude of \( x \).
```

```
template<class T> complex<T> conj(const complex<T>& x);
Returns: The complex conjugate of \( x \).
```
template<class T> complex<T> proj(const complex<T>& x);

Returns: The projection of x onto the Riemann sphere.

Remarks: Behaves the same as the C function cproj, defined in 7.3.9.4.

template<class T> complex<T> polar(const T& rho, const T& theta = 0);

Returns: The complex value corresponding to a complex number whose magnitude is rho and whose phase angle is theta.

26.4.8 complex transcendentals

template<class T> complex<T> acos(const complex<T>& x);

Returns: The complex arc cosine of x.

Remarks: Behaves the same as C function cacos, defined in 7.3.5.1.

template<class T> complex<T> asin(const complex<T>& x);

Returns: The complex arc sine of x.

Remarks: Behaves the same as C function casin, defined in 7.3.5.2.

template<class T> complex<T> atan(const complex<T>& x);

Returns: The complex arc tangent of x.

Remarks: Behaves the same as C function catan, defined in 7.3.5.3.

template<class T> complex<T> acosh(const complex<T>& x);

Returns: The complex arc hyperbolic cosine of x.

Remarks: Behaves the same as C function cacosh, defined in 7.3.6.1.

template<class T> complex<T> asinh(const complex<T>& x);

Returns: The complex arc hyperbolic sine of x.

Remarks: Behaves the same as C function casinh, defined in 7.3.6.2.

template<class T> complex<T> atanh(const complex<T>& x);

Returns: The complex arc hyperbolic tangent of x.

Remarks: Behaves the same as C function catanh, defined in 7.3.6.3.

template<class T> complex<T> cos(const complex<T>& x);

Returns: The complex cosine of x.

template<class T> complex<T> cosh(const complex<T>& x);

Returns: The complex hyperbolic cosine of x.

template<class T> complex<T> exp(const complex<T>& x);

Returns: The complex base e exponential of x.

template<class T> complex<T> log(const complex<T>& x);
16 Remarks: the branch cuts are along the negative real axis.
17
18 Returns: The complex natural (base e) logarithm of \( x \), in the range of a strip mathematically unbounded along the real axis and in the interval \([-i \times \pi, i \times \pi]\) along the imaginary axis. When \( x \) is a negative real number, \( \text{imag}(\log(x)) = \pi \).

```cpp
template<class T> complex<T> log10(const complex<T>& x);
```

18 Remarks: the branch cuts are along the negative real axis.
19
20 Returns: The complex common (base 10) logarithm of \( x \), defined as \( \log(x)/\log(10) \).

```cpp
template<class T>
complex<T> pow(const complex<T>& x, const complex<T>& y);
```

21 Remarks: the branch cuts are along the negative real axis.
22
23 Returns: The complex power of base \( x \) raised to the \( y \)-th power, defined as \( \exp(y \cdot \log(x)) \). The value returned for \( \text{pow}(0,0) \) is implementation-defined.

```cpp
template<class T> complex<T> sin (const complex<T>& x);
```

24 Returns: The complex sine of \( x \).

```cpp
template<class T> complex<T> sinh (const complex<T>& x);
```

25 Returns: The complex hyperbolic sine of \( x \).

```cpp
template<class T> complex<T> sqrt (const complex<T>& x);
```

26 Remarks: the branch cuts are along the negative real axis.
27
28 Returns: The complex square root of \( x \), in the range of the right half-plane. If the argument is a negative real number, the value returned lies on the positive imaginary axis.

```cpp
template<class T> complex<T> tan (const complex<T>& x);
```

29 Returns: The complex tangent of \( x \).

```cpp
template<class T> complex<T> tanh (const complex<T>& x);
```

27 Returns: The complex hyperbolic tangent of \( x \).

### 26.4.9 Additional overloads

The following function templates shall have additional overloads:

```cpp

1. If the argument has type \texttt{long double}, then it is effectively cast to \texttt{complex<long double>}.
2. Otherwise, if the argument has type \texttt{double} or an integer type, then it is effectively cast to \texttt{complex<double>}.
3. Otherwise, if the argument has type \texttt{float}, then it is effectively cast to \texttt{complex<float>}.

§ 26.4.9
Function template `pow` shall have additional overloads sufficient to ensure, for a call with at least one argument of type `complex<T>`:

1. If either argument has type `complex<long double>` or type `long double`, then both arguments are effectively cast to `complex<long double>`.

2. Otherwise, if either argument has type `complex<double>`, `double`, or an integer type, then both arguments are effectively cast to `complex<double>`.

3. Otherwise, if either argument has type `complex<float>` or `float`, then both arguments are effectively cast to `complex<float>`.

### 26.4.10 Suffixes for complex number literals

This section 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)}`.

### 26.4.11 Header `<ccomplex>`

The header behaves as if it simply includes the header `<complex>`.

### 26.5 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 number 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 number 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 (3.9.1) identified as `result_type`. With `T` as the `result_type` thus associated with such an entity, that entity is characterized:

a) as `boolean` or equivalently as `boolean-valued`, if `T` is `bool`;

b) otherwise as `integral` or equivalently as `integer-valued`, if `numeric_limits<T>::is_integer` is true;

c) otherwise as `floating` or equivalently as `real-valued`. 

§ 26.5
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:

a) the operator `rshift` denotes a bitwise right shift with zero-valued bits appearing in the high bits of the result, and

b) the operator `lshift` denotes a bitwise left shift with zero-valued bits appearing in the low bits of the result, and whose result is always taken modulo $2^w$.

26.5.1 Requirements [rand.req]

26.5.1.1 General requirements [rand.req.genl]

Throughout this subclause 26.5, 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 (26.5.1.2).

b) that has a template type parameter named `URNG` is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of uniform random number generator (26.5.1.3).

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 (26.5.1.4).

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`.

Throughout this subclause 26.5, 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.”

Throughout this subclause 26.5, any constructor that can be called with a single argument and that satisfies a requirement specified in this subclause shall be declared `explicit`.

26.5.1.2 Seed sequence requirements [rand.req.seedseq]

A seed sequence is an object that consumes a sequence of integer-valued data and produces a requested number of unsigned integer values $i$, $0 \leq i < 2^{32}$, based on the consumed data. [Note: Such an object provides a mechanism to avoid replication of streams of random variates. This can be useful, for example, in applications requiring large numbers of random number engines. —end note]

A class $S$ satisfies the requirements of a seed sequence if the expressions shown in Table 115 are valid and have the indicated semantics, and if $S$ also satisfies all other requirements of this section 26.5.1.2. In that Table and throughout this section:

a) $T$ is the type named by $S$’s associated `result_type`;

§ 26.5.1.2
b) \( q \) is a value of \( S \) and \( r \) is a possibly const value of \( S \);

c) \( \text{ib} \) and \( \text{ie} \) are input iterators with an unsigned integer \text{value_type} of at least 32 bits;

d) \( \text{rb} \) and \( \text{re} \) are mutable random access iterators with an unsigned integer \text{value_type} of at least 32 bits;

e) \( \text{ob} \) is an output iterator; and

f) \( \text{il} \) is a value of \text{initializer_list}\langle T\rangle.

Table 115 — Seed sequence requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S::\text{result_type} )</td>
<td>( T )</td>
<td>( T ) is an unsigned integer type ((3.9.1)) 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(\text{ib},\text{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 ([\text{ib},\text{ie})).</td>
<td>( O(\text{ie} - \text{ib}) )</td>
</tr>
<tr>
<td>( S(\text{il}) )</td>
<td></td>
<td>Same as ( S(\text{il}.\text{begin()}, \text{il}.\text{end()}).</td>
<td>same as ( S(\text{il}.\text{begin()}, \text{il}.\text{end()})</td>
</tr>
<tr>
<td>( q.\text{generate}(\text{rb},\text{re}) )</td>
<td>\text{void}</td>
<td>Does nothing if ( \text{rb} == \text{re} ). Otherwise, fills the supplied sequence ([\text{rb},\text{re})) with 32-bit quantities that depend on the sequence supplied to the constructor and possibly also depend on the history of \text{generate}'s previous invocations.</td>
<td>( O(\text{re} - \text{rb}) )</td>
</tr>
<tr>
<td>( r.\text{size()} )</td>
<td>\text{size_t}</td>
<td>The number of 32-bit units that would be copied by a call to ( r.\text{param} ).</td>
<td>constant</td>
</tr>
<tr>
<td>( r.\text{param}(\text{ob}) )</td>
<td>\text{void}</td>
<td>Copies to the given destination a sequence of 32-bit units that can be provided to the constructor of a second object of type ( S ), and that would reproduce in that second object a state indistinguishable from the state of the first object.</td>
<td>( O(r.\text{size()} )</td>
</tr>
</tbody>
</table>
26.5.1.3 Uniform random number generator requirements

A uniform random number 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]

A class \( G \) satisfies the requirements of a uniform random number generator if the expressions shown in Table 116 are valid and have the indicated semantics, and if \( G \) also satisfies all other requirements of this section 26.5.1.3. In that Table and throughout this section:

a) \( T \) is the type named by \( G \)'s associated \texttt{result}\_\texttt{type}, and

b) \( g \) is a value of \( G \).

Table 116 — Uniform random number 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::\text{result}_\text{type} )</td>
<td>( T )</td>
<td>( T ) is an unsigned integer type (3.9.1).</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::\text{min}() )</td>
<td>( T )</td>
<td>Denotes the least value potentially returned by ( \text{operator}() ).</td>
<td>compile-time</td>
</tr>
<tr>
<td>( G::\text{max}() )</td>
<td>( T )</td>
<td>Denotes the greatest value potentially returned by ( \text{operator}() ).</td>
<td>compile-time</td>
</tr>
</tbody>
</table>

The following relation shall hold: \( G::\text{min}() < G::\text{max}() \).

26.5.1.4 Random number engine requirements

A random number engine (commonly shortened to engine) \( e \) of type \( E \) is a uniform random number generator that additionally meets the requirements (e.g., for seeding and for input/output) specified in this section.

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 \( \text{seed} \) function, assignment, or a suitable \( \text{operator}>> \).

E’s specification shall define:

a) the size of \( E \)'s state in multiples of the size of \texttt{result}\_\texttt{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 \texttt{result}\_\texttt{type}.

A class \( E \) that satisfies the requirements of a uniform random number generator (26.5.1.3) also satisfies the requirements of a random number engine if the expressions shown in Table 117 are valid and have the indicated semantics, and if \( E \) also satisfies all other requirements of this section 26.5.1.4. In that Table and throughout this section:

a) \( T \) is the type named by \( E \)'s associated \texttt{result}\_\texttt{type};

b) \( e \) is a value of \( E \), \( v \) is an lvalue of \( E \), \( x \) and \( y \) are (possibly \texttt{const}) values of \( E \);
c) \( z \) is a value of \( T \);

d) \( q \) is an lvalue satisfying the requirements of a seed sequence (26.5.1.2);

e) \( z \) is a value of type \texttt{unsigned long long};

f) \texttt{os} is an lvalue of the type of some class template specialization \texttt{basic_ostream<charT, traits>}; and

g) \texttt{is} is an lvalue of the type of some class template specialization \texttt{basic_istream<charT, traits>};

where \texttt{charT} and \texttt{traits} are constrained according to Clause 21 and Clause 27.

Table 117 — Random number engine requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E() )</td>
<td></td>
<td>Creates an engine with the same initial state as all other default-constructed engines of type ( E ).</td>
<td>( O(\text{size of state}) )</td>
</tr>
<tr>
<td>( E(x) )</td>
<td></td>
<td>Creates an engine that compares equal to ( x ).</td>
<td>( O(\text{size of state}) )</td>
</tr>
<tr>
<td>( E(s) )</td>
<td></td>
<td>Creates an engine with initial state determined by ( s ).</td>
<td>( O(\text{size of state}) )</td>
</tr>
<tr>
<td>( E(q) )</td>
<td></td>
<td>Creates an engine with an initial state that depends on a sequence produced by one call to ( q _\text{generate} ).</td>
<td>same as complexity of ( q _\text{generate} ) called on a sequence whose length is size of state</td>
</tr>
<tr>
<td>( e.\text{seed()} )</td>
<td>\texttt{void}</td>
<td>post: ( e == E() ).</td>
<td>same as ( E() )</td>
</tr>
<tr>
<td>( e.\text{seed}(s) )</td>
<td>\texttt{void}</td>
<td>post: ( e == E(s) ).</td>
<td>same as ( E(s) )</td>
</tr>
<tr>
<td>( e.\text{seed}(q) )</td>
<td>\texttt{void}</td>
<td>post: ( 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} = T\text{A}(e_i) ) and returns ( G\text{A}(e_i) ).</td>
<td>per Table 116</td>
</tr>
<tr>
<td>( e.\text{discard}(z) )</td>
<td>\texttt{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>
<tr>
<td>( x == y )</td>
<td>\texttt{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 \texttt{true} if ( S_x = S_y ); else returns \texttt{false}.</td>
<td>( O(\text{size of state}) )</td>
</tr>
<tr>
<td>( x !!= y )</td>
<td>\texttt{bool}</td>
<td>(! (x == y) ).</td>
<td>( O(\text{size of state}) )</td>
</tr>
</tbody>
</table>

274) This constructor (as well as the subsequent corresponding \texttt{seed()} function) may be particularly useful to applications requiring a large number of independent random sequences.

275) This operation is common in user code, and can often be implemented in an engine-specific manner so as to provide significant performance improvements over an equivalent naive loop that makes \( z \) consecutive calls \( e() \).
<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>os &lt;&lt; x</code></td>
<td>reference to the type of</td>
<td>With <code>os.fmtflags</code> set to `ios_base::dec</td>
<td>ios_base::left<code>and the fill character set to the space character, writes to</code>os<code>the textual representation of</code>x<code>'s current state. In the output, adjacent numbers are separated by one or more space characters. post: The </code>os.fmtflags` and fill character are unchanged.</td>
</tr>
<tr>
<td><code>is &gt;&gt; v</code></td>
<td>reference to the type of</td>
<td>With <code>is.fmtflags</code> set to <code>ios_base::dec</code>, sets <code>v</code>'s state as determined by reading its textual representation from <code>is</code>. If bad input is encountered, ensures that <code>v</code>'s state is unchanged by the operation and calls <code>is.setstate(ios::failbit)</code> (which may throw <code>ios::failure</code> [27.5.5.4]). If a textual representation written via <code>os &lt;&lt; x</code> was subsequently read via <code>is &gt;&gt; v</code>, then <code>x == v</code> provided that there have been no intervening invocations of <code>x</code> or of <code>v</code>. pre: <code>is</code> provides a textual representation that was previously written using an output stream whose imbued locale was the same as that of <code>is</code>, and whose type's template specialization arguments <code>charT</code> and <code>traits</code> were respectively the same as those of <code>is</code>. post: The <code>is.fmtflags</code> are unchanged.</td>
<td>( O(\text{size of state}) )</td>
</tr>
</tbody>
</table>

5 E shall meet the requirements of `CopyConstructible` (Table 21) and `CopyAssignable` (Table 23) types. These operations shall each be of complexity no worse than \( O(\text{size of state}) \).

26.5.1.5 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.

```cpp
A::A();
```

**Effects:** The base engine is initialized as if by its default constructor.

```cpp
bool operator==(const A& a1, const A& a2);
```

**Returns:** `true` if `a1`'s base engine is equal to `a2`'s base engine. Otherwise returns `false`.

```cpp
A::A(result_type s);
```

**Effects:** The base engine is initialized with `s`.

```cpp
template<class Sseq> void A::A(Sseq& q);
```

**Effects:** The base engine is initialized with `q`.

```cpp
void seed();
```

**Effects:** With `b` as the base engine, invokes `b.seed()`.

```cpp
void seed(result_type s);
```

**Effects:** With `b` as the base engine, invokes `b.seed(s)`.

```cpp
template<class Sseq> void seed(Sseq& q);
```

**Effects:** With `b` as the base engine, invokes `b.seed(q)`.

**A** shall also 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.

### 26.5.1.6 Random number distribution requirements

A *random number distribution* (commonly shortened to *distribution*) of type `D` is a function object returning values that are distributed according to an associated mathematical *probability density function* `p(z)` or according to an associated *discrete probability function* `P(z_i)`. A distribution's specification identifies its associated probability function `p(z)` or `P(z_i)`.

An associated probability function is typically expressed using certain externally-supplied quantities known as the *parameters of the distribution*. Such distribution parameters are identified in this context by writing, for example, `p(z | a, b)` or `P(z_i | a, b)`, to name specific parameters, or by writing, for example, `p(z | {p})` or `P(z_i | {p})`, to denote a distribution's parameters `p` taken as a whole.

A class `D` satisfies the requirements of a *random number distribution* if the expressions shown in Table 118 are valid and have the indicated semantics, and if `D` and its associated types also satisfy all other requirements of this section 26.5.1.6. In that Table and throughout this section,

a) `T` is the type named by `D`'s associated `result_type`
b) \( P \) is the type named by \( D \)'s associated \texttt{param\_type};

c) \( d \) is a value of \( D \), and \( x \) and \( y \) are (possibly \texttt{const}) values of \( D \);

d) \( \text{glb} \) and \( \text{lub} \) are values of \( T \) respectively corresponding to the greatest lower bound and the least upper bound on the values potentially returned by \( d \)'s \texttt{operator()}\, as determined by the current values of \( d \)'s parameters;

e) \( p \) is a (possibly \texttt{const}) value of \( P \);

f) \( g, g_1, \) and \( g_2 \) are lvalues of a type satisfying the requirements of a uniform random number generator \[26.5.1.3\];

g) \( \text{os} \) is an lvalue of the type of some class template specialization \texttt{basic\_ostream<\texttt{charT}, traits>;} and

h) \( \text{is} \) is an lvalue of the type of some class template specialization \texttt{basic\_istream<\texttt{charT}, traits>};

where \texttt{charT} and \texttt{traits} are constrained according to Clauses 21 and 27.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{D::result_type} )</td>
<td>( T )</td>
<td>( T ) is an arithmetic type (3.9.1).</td>
<td>compile-time</td>
</tr>
<tr>
<td>( \text{D::param_type} )</td>
<td>( P )</td>
<td></td>
<td>compile-time</td>
</tr>
<tr>
<td>( \text{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>( \text{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>( \text{d.reset()} )</td>
<td>( \text{void} )</td>
<td>Subsequent uses of ( d ) do not depend on values produced by any engine prior to invoking \texttt{reset}.</td>
<td>constant</td>
</tr>
<tr>
<td>( \text{x.param()} )</td>
<td>( P )</td>
<td>Returns a value ( p ) such that ( \text{D(p).param()} == p ).</td>
<td>no worse than the complexity of ( \text{D(p)} )</td>
</tr>
<tr>
<td>( \text{d.param(p)} )</td>
<td>( \text{void} )</td>
<td>post: ( \text{d.param()} == p ).</td>
<td>no worse than the complexity of ( \text{D(p)} )</td>
</tr>
<tr>
<td>( \text{d(g)} )</td>
<td>( T )</td>
<td>With ( p = \text{d.param()} ), the sequence of numbers returned by successive invocations with the same object ( g ) is randomly distributed according to the associated ( p(z</td>
<td>{p}) ) or ( P(z_i</td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Pre/post-condition</td>
<td>Complexity</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>--------------------</td>
<td>------------</td>
</tr>
<tr>
<td>d(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 $p(z \mid {p})$ or $P(z_i \mid {p})$ function.</td>
<td>amortized constant number of invocations of g</td>
</tr>
<tr>
<td>x.min()</td>
<td>T</td>
<td>Returns $\text{glb}$.</td>
<td>constant</td>
</tr>
<tr>
<td>x.max()</td>
<td>T</td>
<td>Returns $\text{lub}$.</td>
<td>constant</td>
</tr>
<tr>
<td>x == y</td>
<td>bool</td>
<td>This operator is an equivalence relation. Returns true if $x\text{.param()} == y\text{.param()}$ and $S_1 = S_2$, where $S_1$ and $S_2$ are the infinite sequences of values that would be generated, respectively, by repeated future calls to $x(g1)$ and $y(g2)$ whenever $g1 == g2$. Otherwise returns false.</td>
<td>constant</td>
</tr>
<tr>
<td>x != y</td>
<td>bool</td>
<td>!(x == y).</td>
<td>same as x == y</td>
</tr>
<tr>
<td>os &lt;&lt; x</td>
<td>reference to the type of os</td>
<td>Writes to os a textual representation for the parameters and the additional internal data of x. post: 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 [27.5.5.4]). pre: 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. post: The is.fmtflags are unchanged.</td>
<td></td>
</tr>
</tbody>
</table>

---

4 D shall satisfy the requirements of CopyConstructible (Table 21) and CopyAssignable (Table 23) 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 26.5, 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 21), CopyAssignable (Table 23), and EqualityComparable (Table 17) 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

    typedef D distribution_type;

26.5.2 Header <random> synopsis

#include <initializer_list>

namespace std {

    // 26.5.3.1, class template linear_congruential_engine
    template<class UIntType, UIntType a, UIntType c, UIntType m>
        class linear_congruential_engine;

    // 26.5.3.2, class template mersenne_twister_engine
    template<class UIntType, size_t w, size_t n, size_t m, size_t r,
            UIntType a, size_t u, UIntType d, size_t s,
            UIntType b, size_t t,
            UIntType c, size_t l, UIntType f>
        class mersenne_twister_engine;

    // 26.5.3.3, class template subtract_with_carry_engine
    template<class UIntType, size_t w, size_t s, size_t r>
        class subtract_with_carry_engine;

    // 26.5.4.2, class template discard_block_engine
    template<class Engine, size_t p, size_t r>
        class discard_block_engine;

    // 26.5.4.3, class template independent_bits_engine
    template<class Engine, size_t w, class UIntType>
        class independent_bits_engine;

    // 26.5.4.4, class template shuffle_order_engine
    template<class Engine, size_t k>
        class shuffle_order_engine;

    // 26.5.5, engines and engine adaptors with predefined parameters
    typedef see below minstd_rand0;
    typedef see below minstd_rand;
typedef see below mt19937;
typedef see below mt19937_64;
typedef see below ranlux24_base;
typedef see below ranlux48_base;
typedef see below ranlux24;
typedef see below ranlux48;
typedef see below knuth_b;
typedef see below default_random_engine;

// 26.5.6, class random_device
class random_device;

// 26.5.7.1, class seed_seq
class seed_seq;

// 26.5.7.2, function template generate_canonical
template<class RealType, size_t bits, class URNG>
  RealType generate_canonical(URNG& g);

// 26.5.8.2.1, class template uniform_int_distribution
template<class IntType = int>
  class uniform_int_distribution;

// 26.5.8.2.2, class template uniform_real_distribution
template<class RealType = double>
  class uniform_real_distribution;

// 26.5.8.3.1, class bernoulli_distribution
class bernoulli_distribution;

// 26.5.8.3.2, class template binomial_distribution
template<class IntType = int>
  class binomial_distribution;

// 26.5.8.3.3, class template geometric_distribution
template<class IntType = int>
  class geometric_distribution;

// 26.5.8.3.4, class template negative_binomial_distribution
template<class IntType = int>
  class negative_binomial_distribution;

// 26.5.8.4.1, class template poisson_distribution
template<class IntType = int>
  class poisson_distribution;

// 26.5.8.4.2, class template exponential_distribution
template<class RealType = double>
  class exponential_distribution;

// 26.5.8.4.3, class template gamma_distribution
template<class RealType = double>
  class gamma_distribution;

// 26.5.8.4.4, class template weibull_distribution
template<class RealType = double>
class weibull_distribution;

// 26.5.8.4.5, class template extreme_value_distribution
template<class RealType = double>
class extreme_value_distribution;

// 26.5.8.5.1, class template normal_distribution
template<class RealType = double>
class normal_distribution;

// 26.5.8.5.2, class template lognormal_distribution
template<class RealType = double>
class lognormal_distribution;

// 26.5.8.5.3, class template chi_squared_distribution
template<class RealType = double>
class chi_squared_distribution;

// 26.5.8.5.4, class template cauchy_distribution
template<class RealType = double>
class cauchy_distribution;

// 26.5.8.5.5, class template fisher_f_distribution
template<class RealType = double>
class fisher_f_distribution;

// 26.5.8.5.6, class template student_t_distribution
template<class RealType = double>
class student_t_distribution;

// 26.5.8.6.1, class template discrete_distribution
template<class IntType = int>
class discrete_distribution;

// 26.5.8.6.2, class template piecewise_constant_distribution
template<class RealType = double>
class piecewise_constant_distribution;

// 26.5.8.6.3, class template piecewise_linear_distribution
template<class RealType = double>
class piecewise_linear_distribution;

} // namespace std

26.5.3 Random number engine class templates

Each type instantiated from a class template specified in this section 26.5.3 satisfies the requirements of a random number engine (26.5.1.4) type.

Except where specified otherwise, the complexity of each function specified in this section 26.5.3 is constant.

Except where specified otherwise, no function described in this section 26.5.3 throws an exception.

Descriptions are provided in this section 26.5.3 only for engine operations that are not described in 26.5.1.4 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.

5 Each template specified in this section 26.5.3 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.

6 For every random number engine and for every random number engine adaptor \( X \) defined in this subclause (26.5.3) and in sub-clause 26.5.4:

(6.1) — if the constructor

\[
\text{template} \ <\text{class} \ Sseq> \ \text{explicit} \ X(Sseq& \ q);
\]

is called with a type \( Sseq \) that does not qualify as a seed sequence, then this constructor shall not participate in overload resolution;

(6.2) — if the member function

\[
\text{template} \ <\text{class} \ Sseq> \ \text{void} \ \text{seed}(Sseq& \ q);
\]

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 \( X::\text{result\_type} \).

26.5.3.1 Class template \texttt{linear\_congruential\_engine} \[\text{rand.eng.lcong}\]

1 A \texttt{linear\_congruential\_engine} random number engine produces unsigned integer random numbers. The state \( x_i \) of a \texttt{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 \( GA(x_i) = x_i + 1 \).

\[
\text{template}<\text{class} \ \texttt{UIntType}, \ \texttt{UIntType} \ a, \ \texttt{UIntType} \ c, \ \texttt{UIntType} \ m>
\]

\[
\text{class} \ \text{linear\_congruential\_engine}
\]

\{
public:
// types
typedef \texttt{UIntType} \ result\_type;

// engine characteristics
static constexpr \texttt{result\_type} \ multiplier = a;
static constexpr \texttt{result\_type} \ increment = c;
static constexpr \texttt{result\_type} \ modulus = m;
static constexpr \texttt{result\_type} \ min() \ { return c == 0u ? 1u: 0u; }
static constexpr \texttt{result\_type} \ max() \ { return m - 1u; }
static constexpr \texttt{result\_type} \ default\_seed = 1u;

// constructors and seeding functions
explicit \text{linear\_congruential\_engine}(\text{result\_type} \ s = \text{default\_seed});
\text{template}<\text{class} \ Sseq> \ \text{explicit} \ \text{linear\_congruential\_engine}(Sseq& \ q);
void \text{seed}(\text{result\_type} \ s = \text{default\_seed});
\text{template}<\text{class} \ Sseq> \ \text{void} \ \text{seed}(Sseq& \ q);

// generating functions
\text{result\_type} \ \text{operator}();
\text{void} \ \text{discard}(\text{unsigned} \ \text{long} \ \text{long} \ z);
\}

\section*{§ 26.5.3.1}
If the template parameter \( m \) is 0, the modulus \( m \) used throughout this section 26.5.3.1 is \( \text{numeric_limits<result_type>::max()} \) plus 1. [Note: \( m \) need not be representable as a value of type \( \text{result_type} \). — 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 \( x_i \).

```cpp
explicit linear_congruential_engine(result_type s = default_seed);
```

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 \( s \mod m \).

```cpp
template<class Sseq> explicit linear_congruential_engine(Sseq& q);
```

Effects: Constructs a `linear_congruential_engine` object. With \( k = \lceil \log_2 m \cdot 32 \rceil \) and \( a \) an array (or equivalent) of length \( k + 3 \), invokes \( q\text{.generate}(a + 0, a + k + 3) \) and then computes \( S = \left( \sum_{j=0}^{k-1} a_{j+3} \cdot 2^{32j} \right) \mod m \). If \( c \mod m \) is 0 and \( S \) is 0, sets the engine’s state to 1, else sets the engine’s state to \( S \).

### 26.5.3.2 Class template `mersenne_twister_engine`

A `mersenne_twister_engine` random number engine\(^{276} \) produces unsigned integer random numbers in the closed interval \([0, 2^w - 1]\). The state \( x_i \) of a `mersenne_twister_engine` object \( x \) is of size \( n \) and consists of a sequence \( X \) of \( n \) values of the type delivered by \( x \); all subscripts applied to \( X \) are to be taken modulo \( n \).

The transition algorithm employs a twisted generalized feedback shift register defined by shift values \( n \) and \( m \), a twist value \( r \), and a conditional xor-mask \( a \). To improve the uniformity of the result, the bits of the raw shift register are additionally tempered (i.e., scrambled) according to a bit-scrambling matrix defined by values \( u, d, s, b, t, c, \) and \( \ell \).

The state transition is performed as follows:

a) 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 \( Y \).

b) 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:

a) Let \( z_1 = X_i \text{xor} ((X_i \text{rshift} w) \text{bitand} d) \).

b) Let \( z_2 = z_1 \text{xor} ((z_1 \text{lshift} w) \text{bitand} b) \).

c) Let \( z_3 = z_2 \text{xor} ((z_2 \text{lshift} w) \text{bitand} c) \).

d) Let \( z_4 = z_3 \text{xor} (z_3 \text{rshift} \ell) \).

```cpp
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,
```

\(^{276}\) 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.
class mersenne_twister_engine
{
public:
// types
typedef UIntType result_type;

// engine characteristics
static constexpr size_t word_size = w;
static constexpr size_t state_size = n;
static constexpr size_t shift_size = m;
static constexpr size_t mask_bits = r;
static constexpr UIntType xor_mask = a;
static constexpr size_t tempering_u = u;
static constexpr UIntType tempering_d = d;
static constexpr size_t tempering_s = s;
static constexpr UIntType tempering_b = b;
static constexpr size_t tempering_t = t;
static constexpr UIntType tempering_c = c;
static constexpr size_t tempering_l = l;
static constexpr UIntType initialization_multiplier = f;
static constexpr result_type min() { return 0; }
static constexpr result_type max() { return 2^w − 1; }
static constexpr result_type default_seed = 5489u;

// constructors and seeding functions
explicit mersenne_twister_engine(result_type value = default_seed);
template<class Sseq> explicit mersenne_twister_engine(Sseq& q);
void seed(result_type value = default_seed);
template<class Sseq> void seed(Sseq& q);

// generating functions
result_type operator()();
void discard(unsigned long long z);
};
26.5.3.3 Class template subtract_with_carry_engine

1 A **subtract_with_carry_engine** random number engine produces unsigned integer random numbers.

2 The state $x_i$ of a **subtract_with_carry_engine** object $x$ is of size $\mathcal{O}(r)$, and consists of a sequence $X$ of $r$ integer values $0 \leq X_i < m = 2^w$; all subscripts applied to $X$ are to be taken modulo $r$. The state $x_i$ additionally consists of an integer $c$ (known as the *carry*) whose value is either 0 or 1.

3 The state transition is performed as follows:

   a) Let $Y = X_{i-s} - X_{i-r} - c$.

   b) Set $X_i$ to $y = Y \mod m$. Set $c$ to 1 if $Y < 0$, otherwise set $c$ to 0.

   [Note: This algorithm corresponds to a modular linear function of the form $TA(x_i) = (a \cdot x_i) \mod b$, where $b$ is of the form $m' - m^2 + 1$ 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.

```cpp
template<class UIntType, size_t w, size_t s, size_t r>
class subtract_with_carry_engine
{
public:
  // types
  typedef UIntType result_type;

  // 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);
};
```

5 The following relations shall hold: $0u < s, s < r, 0 < w,$ and $w \leq \text{numeric_limits<UIntType>::digits}$.

6 The textual representation consists of the values of $X_{i-r}, \ldots, X_{i-1}$, in that order, followed by $c$.

```cpp
explicit subtract_with_carry_engine(result_type value = default_seed);
```

7 **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:

```cpp
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 $(\sum_{j=0}^{n-1} z_j \cdot 2^{32}) \mod m$.

Complexity: Exactly $n \cdot r$ invocations of $e$.

template<class Sseq> explicit subtract_with_carry_engine(Sseq& q);

Effects: Constructs a subtract_with_carry_engine object. With $k = \lceil w/32 \rceil$ and $a$ an array (or equivalent) of length $r \cdot k$, invokes $q$.generate($a+0$, $a+r \cdot k$) and then, iteratively for $i = -r, \ldots, -1$, sets $X_i$ to $(\sum_{j=0}^{k-1} a_{k(i+r)+j} \cdot 2^{32}) \mod m$. If $X_{-1}$ is then 0, sets $c$ to 1; otherwise sets $c$ to 0.

26.5.4 Random number engine adaptor class templates

26.5.4.1 In general

1 Each type instantiated from a class template specified in this section 26.5.3 satisfies the requirements of a random number engine adaptor (26.5.1.5) type.

2 Except where specified otherwise, the complexity of each function specified in this section 26.5.4 is constant.

3 Except where specified otherwise, no function described in this section 26.5.4 throws an exception.

4 Descriptions are provided in this section 26.5.4 only for adaptor operations that are not described in section 26.5.1.5 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.

5 Each template specified in this section 26.5.4 requires one or more relationships, involving the value(s) of its non-type template parameter(s), to hold. A program instantiating any of these templates is ill-formed if any such required relationship fails to hold.

26.5.4.2 Class template discard_block_engine

1 A discard_block_engine random number engine adaptor produces random numbers selected from those produced by some base engine $e$. The state $x_i$ of a discard_block_engine engine adaptor object $x$ consists of the state $e_i$ of its base engine $e$ and an additional integer $n$. The size of the state is the size of $e$’s state plus 1.

2 The transition algorithm discards all but $r > 0$ values from each block of $p \geq r$ values delivered by $e$. The state transition is performed as follows: If $n \geq r$, advance the state of $e$ from $e_i$ to $e_{i+p-r}$ and set $n$ to 0. In any case, then increment $n$ and advance $e$’s then-current state $e_j$ to $e_{j+1}$.

3 The generation algorithm yields the value returned by the last invocation of $e()$ while advancing $e$’s state as described above.

template<class Engine, size_t p, size_t r>
  class discard_block_engine
  {
    public:
      // types
      typedef typename Engine::result_type result_type;

      // engine characteristics
      static constexpr size_t block_size = p;
      static constexpr size_t used_block = r;
      static result_type min() { return Engine::min(); }
      static 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 <= p.
The textual representation consists of the textual representation of e followed by the value of n.
In addition to its behavior pursuant to section 26.5.1.5, each constructor that is not a copy constructor sets n to 0.

26.5.4.3 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 \( R = e.\max() - e.\min() + 1 \) 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 \( w = n_0w_0 + (n - n_0)(w_0 + 1) \) always holds. — end note]

The transition algorithm is carried out by invoking \( e() \) as often as needed to obtain \( n_0 \) values less than \( y_0 + e.\min() \) and \( n - n_0 \) values less than \( y_1 + e.\min() \).

The generation algorithm uses the values produced while advancing the state as described above to yield a quantity \( S \) obtained as if by the following algorithm:

\[
S = 0;
\text{for } (k = 0; k \neq n_0; k += 1) \{ \\
    \text{do } u = e() - e.\min(); \text{ while } (u \geq y_0);
    S = 2^{w_0} \cdot S + u \mod 2^{w_0};
\}
\text{for } (k = n_0; k \neq n; k += 1) \{ \\
\]
do \( u = e() - e\text{.min}; \) while \( (u \geq y_1); \)
\( S = 2^{w_0+1} \cdot S + u \mod 2^{w_0+1}; \)

```cpp
template<class Engine, size_t w, class UIntType>
class independent_bits_engine
{
public:
    // types
    typedef UIntType result_type;

    // 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 \( w \leq \texttt{numeric\_limits<result\_type>::digits}. \)

6 The textual representation consists of the textual representation of \( e. \)

### 26.5.4.4 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 \( k + 1. \)

2 The transition algorithm permutes the values produced by \( e. \) The state transition is performed as follows:

a) Calculate an integer \( j = \left\lfloor \frac{k(Y-e\text{.min})}{e\text{.max}-e\text{.min}+1} \right\rfloor. \)

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.
template<class Engine, size_t k>
class shuffle_order_engine
{
public:
   // types
typedef typename Engine::result_type 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 Y; // exposition only
   result_type V[k]; // 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 section 26.5.1.5, each constructor that is not a copy constructor initializes V[0],...,V[k-1] and Y, in that order, with values returned by successive invocations of e().

26.5.5 Engines and engine adaptors with predefined parameters [rand.predef]

typedef linear_congruential_engine<uint_fast32_t, 16807, 0, 2147483647> minstd_rand0;

1 Required behavior: The 10000th consecutive invocation of a default-constructed object of type minstd_rand0 shall produce the value 1043618065.

typedef linear_congruential_engine<uint_fast32_t, 48271, 0, 2147483647> minstd_rand;

2 Required behavior: The 10000th consecutive invocation of a default-constructed object of type minstd_rand shall produce the value 399268537.
typedef mersenne_twister_engine<uint_fast32_t, 32, 624, 397, 31, 0x9908b0df, 11, 0xffffffff, 7, 0x9d2c5680, 15, 0xefc60000, 18, 1812433253>
mt19937;

3  Required behavior: The 10000th consecutive invocation of a default-constructed object of type mt19937 shall produce the value 4123659995.

typedef mersenne_twister_engine<uint_fast64_t, 64, 312, 156, 31, 0xb5026f5aa96619e9, 29, 0x5555555555555555, 17, 0x71d67ffffffeda60000, 37, 0xfff7eee000000000, 43, 6364136223846793005>
mt19937_64;

4  Required behavior: The 10000th consecutive invocation of a default-constructed object of type mt19937_64 shall produce the value 9981545732273789042.

typedef subtract_with_carry_engine<uint_fast32_t, 24, 10, 24>
ranlux24_base;

5  Required behavior: The 10000th consecutive invocation of a default-constructed object of type ranlux24_base shall produce the value 7937952.

typedef subtract_with_carry_engine<uint_fast64_t, 48, 5, 12>
ranlux48_base;

6  Required behavior: The 10000th consecutive invocation of a default-constructed object of type ranlux48_base shall produce the value 61839128582725.

typedef discard_block_engine<ranlux24_base, 223, 23>
ranlux24;

7  Required behavior: The 10000th consecutive invocation of a default-constructed object of type ranlux24 shall produce the value 9901578.

typedef discard_block_engine<ranlux48_base, 389, 11>
ranlux48;

8  Required behavior: The 10000th consecutive invocation of a default-constructed object of type ranlux48 shall produce the value 249142670248501.

typedef shuffle_order_engine<minstd_rand0, 256>
knuth_b;

9  Required behavior: The 10000th consecutive invocation of a default-constructed object of type knuth_b shall produce the value 1112339016.

typedef implementation-defined
default_random_engine;

10  Remark: The choice of engine type named by this typedef is implementation-defined. [ Note: The implementation may select this type on the basis of performance, size, quality, or any combination of such factors, so as to provide at least acceptable engine behavior for relatively casual, inexpert, and/or lightweight use. Because different implementations may select different underlying engine types, code that uses this typedef need not generate identical sequences across implementations. — end note ]
26.5.6 Class random_device

A `random_device` uniform random number generator produces non-deterministic random numbers.

If implementation limitations prevent generating non-deterministic random numbers, the implementation may employ a random number engine.

```cpp
class random_device
{
public:
    // types
    typedef unsigned int result_type;

    // 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` non-deterministic uniform random number generator object. The semantics and default value of the `token` parameter are implementation-defined.\(^{277}\)

**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\(^{278}\) for the random numbers returned by `operator()`, in the range `min()` to \(\log_2(max() + 1)\).

`result_type operator()();`

**Returns:** A non-deterministic 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.

---

\(^{277}\) The parameter is intended to allow an implementation to differentiate between different sources of randomness.

\(^{278}\) If a device has \(n\) states whose respective probabilities are \(P_0, \ldots, P_{n-1}\), the device entropy \(S\) is defined as

\[
S = - \sum_{i=0}^{n-1} P_i \cdot \log P_i.
\]
26.5.7 Utilities

26.5.7.1 Class seed_seq

```cpp
class seed_seq {
public:
    // types
    typedef uint_least32_t result_type;

    // 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;
    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
};
```

1. **Effects**: Constructs a `seed_seq` object as if by default-constructing its member v.

2. **Throws**: Nothing.

```cpp
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())`.

```cpp
template<class InputIterator>
seed_seq(InputIterator begin, InputIterator end);
```

5. **Requires**: `InputIterator` shall satisfy the requirements of an input iterator (Table 107) type. Moreover, `iterator_traits<InputIterator>::value_type` shall denote an integer type.

6. **Effects**: Constructs a `seed_seq` object by the following algorithm:

   ```cpp
   for( InputIterator s = begin; s != end; ++s)
       v.push_back((*s)%32);
   ```

```cpp
template<class RandomAccessIterator>
void generate(RandomAccessIterator begin, RandomAccessIterator end);
```
Requires: RandomAccessIterator shall meet the requirements of a mutable random access iterator (Table 111) type. Moreover, iterator_traits<RandomAccessIterator>::value_type shall denote an unsigned integer type capable of accommodating 32-bit quantities.

Effects: Does nothing if begin == end. Otherwise, with s = v.size() and n = end - begin, fills the supplied range [begin, end) according to the following algorithm in which each operation is to be carried out modulo \(2^{32}\), each indexing operator applied to begin is to be taken modulo \(n\), and \(T(x)\) is defined as \(x \ xor \ (x \ rshift 27)\):

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

\[
s = (n \geq 623) \ ? 11 : (n \geq 68) \ ? 7 : (n \geq 39) \ ? 5 : (n \geq 7) \ ? 3 : (n - 1)/2;
\]

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

\[
\begin{align*}
r_1 &= 1664525 \cdot T(begin[k] \ xor \ begin[k + p] \ xor \ begin[k - 1]) \\
r_2 &= r_1 + \left\{ \begin{array}{ll}
& s \mod n + v[k - 1], \ 0 < k \leq s \\
& k \mod n \ , \ s < k
\end{array} \right.
\end{align*}
\]

and, in order, increment begin\([k + p]\) by \(r_1\), increment begin\([k + q]\) by \(r_2\), and set begin\([k]\) to \(r_2\).

c) Transform the elements of the range again, beginning where the previous step ended: iteratively for \(k = m, \ldots, m+n-1\), calculate values

\[
\begin{align*}
r_3 &= 1566083941 \cdot T(begin[k] + begin[k + p] + begin[k - 1]) \\
r_4 &= r_3 - (k \mod n)
\end{align*}
\]

and, in order, update begin\([k + p]\) by xoring it with \(r_3\), update begin\([k + q]\) by xoring it with \(r_4\), and set begin\([k]\) to \(r_4\).

Throws: What and when RandomAccessIterator operations of begin and end throw.

size_t size() const;  
Returns: The number of 32-bit units that would be returned by a call to param().  
Throws: Nothing.  
Complexity: Constant time.

template<class OutputIterator>  
void param(OutputIterator dest) const;  
Requires: OutputIterator shall satisfy the requirements of an output iterator (Table 108) type. 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:

\[
\text{copy(v.begin(), v.end(), dest);}
\]

Throws: What and when OutputIterator operations of dest throw.
26.5.7.2 Function template generate_canonical

Each function instantiated from the template described in this section 26.5.7.2 maps the result of one or more invocations of a supplied uniform random number 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 number generator into a value that can be delivered by a random number distribution. —end note]

\[
\text{template}<\text{class RealType, size_t bits, class URNG}>
\]

\[
\text{RealType generate_canonical(URNG}& g); \]

Complexity: Exactly \( k = \max(1, \lceil b/\log_2 R \rceil) \) invocations of \( g \), where \( b^{279} \) is the lesser of \( \text{numeric_limits<RealType>::digits} \) and \( \text{bits} \), 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

\[
S = \sum_{i=0}^{k-1} (g_i - g.\min()) \cdot R^i
\]

using arithmetic of type RealType.

Returns: \( S/R^k \).

Throws: What and when \( g \) throws.

26.5.8 Random number distribution class templates

26.5.8.1 In general

Each type instantiated from a class template specified in this section 26.5.8 satisfies the requirements of a random number distribution (26.5.1.6) type.

Descriptions are provided in this section 26.5.8 only for distribution operations that are not described in 26.5.1.6 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.

The algorithms for producing each of the specified distributions are implementation-defined.

The value of each probability density function \( p(z) \) and of each discrete probability function \( P(z_i) \) specified in this section is 0 everywhere outside its stated domain.

26.5.8.2 Uniform distributions

26.5.8.2.1 Class template uniform_int_distribution

A uniform_int_distribution random number distribution produces random integers \( i, a \leq i \leq b \), distributed according to the constant discrete probability function

\[
P(i \mid a, b) = 1/(b - a + 1) .
\]

\[
\text{template<class IntType = int>}
\]

\[
\text{class uniform_int_distribution}
\]

\[
\{
\text{public:}
// types
\]
typedef IntType result_type;
typedef unspecified param_type;

// 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 URNG>
result_type operator()(URNG& g);
template<class URNG>
result_type operator()(URNG& 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());

2 Requires: a \leq b.
3 Effects: Constructs a uniform_int_distribution object; a and b correspond to the respective parameters of the distribution.

result_type a() const;
4 Returns: The value of the a parameter with which the object was constructed.

result_type b() const;
5 Returns: The value of the b parameter with which the object was constructed.

26.5.8.2.2 Class template uniform_real_distribution

A uniform_real_distribution random number distribution produces random numbers \( x, a \leq x < b \), distributed according to the constant probability density function

\[
p(x \mid a,b) = 1/(b-a) .
\]

template<class RealType = double>
class uniform_real_distribution
{
public:
    // types
    typedef RealType result_type;
typedef unspecified param_type;

    // 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();

§ 26.5.8.2.2
// generating functions
template<class URNG>
    result_type operator()(URNG& g);

// 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);

2 Requires: a ≤ b and b − a ≤ numeric_limits<RealType>::max().
3 Effects: Constructs a uniform_real_distribution object; a and b correspond to the respective parameters of the distribution.
result_type a() const;
4 Returns: The value of the a parameter with which the object was constructed.
result_type b() const;
5 Returns: The value of the b parameter with which the object was constructed.

26.5.8.3 Bernoulli distributions
26.5.8.3.1 Class bernoulli_distribution

1 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
    typedef bool result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit bernoulli_distribution(double p = 0.5);
    explicit bernoulli_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URNG>
        result_type operator()(URNG& g);
    template<class URNG>

result_type operator()(URNG& 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.

26.5.8.3.2 Class template binomial_distribution [rand.dist.bern.bin]

A binomial_distribution random number distribution produces integer values i ≥ 0 distributed according to the discrete probability function

\[ P(i|t,p) = \binom{t}{i} \cdot p^i \cdot (1-p)^{t-i} . \]

template<class IntType = int>
class binomial_distribution
{
public:
   // types
   typedef IntType result_type;
   typedef unspecified param_type;

   // 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 URNG>
      result_type operator()(URNG& g);
   template<class URNG>
      result_type operator()(URNG& 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;
};

§ 26.5.8.3.2
explicit binomial_distribution(IntType t = 1, double p = 0.5);

2 Requires: $0 \leq p \leq 1$ and $0 \leq t$.
3 Effects: Constructs a binomial_distribution object; $t$ and $p$ correspond to the respective parameters of the distribution.

IntType t() const;

4 Returns: The value of the $t$ parameter with which the object was constructed.

double p() const;

5 Returns: The value of the $p$ parameter with which the object was constructed.

§ 26.5.8.3.3 Class template geometric_distribution

A geometric_distribution random number distribution produces integer values $i \geq 0$ distributed according to the discrete probability function

$$P(i|p) = p \cdot (1 - p)^i.$$
26.5.8.3.4 Class template negative_binomial_distribution

A `negative_binomial_distribution` random number distribution produces random integers \( i \geq 0 \) distributed according to the discrete probability function
\[
P(i | k, p) = \binom{k + i - 1}{i} \cdot p^k \cdot (1 - p)^i.
\]

```cpp
template<class IntType = int>
class negative_binomial_distribution
{
public:
    // types
    typedef IntType result_type;
    typedef unspecified param_type;

    // 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 URNG>
    result_type operator()(URNG& g);
    template<class URNG>
    result_type operator()(URNG& 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;
};
```

```cpp
effective negative_binomial_distribution(IntType k = 1, double p = 0.5);

- Requires: \( 0 < p \leq 1 \) and \( 0 < k \).
- Effects: Constructs a `negative_binomial_distribution` object; \( k \) and \( p \) correspond to the respective parameters of the distribution.

IntType k() const;
- Returns: The value of the \( k \) parameter with which the object was constructed.

double p() const;
- Returns: The value of the \( p \) parameter with which the object was constructed.
```

26.5.8.4 Poisson distributions

26.5.8.4.1 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 \textit{mean}.

```cpp
template<class IntType = int>
class poisson_distribution
{
public:
    // types
    typedef IntType result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit poisson_distribution(double mean = 1.0);
    explicit poisson_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URNG>
    result_type operator()(URNG& g);
    template<class URNG>
    result_type operator()(URNG& g, const param_type& parm);

    // property functions
    double mean() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};

explicit poisson_distribution(double mean = 1.0);
```

2 \textit{Requires}: $0 < \text{mean}$.

3 \textit{Effects}: Constructs a \texttt{poisson\_distribution} object; \texttt{mean} corresponds to the parameter of the distribution.

4 \textit{Returns}: The value of the \texttt{mean} parameter with which the object was constructed.

\textbf{26.5.8.4.2} Class template \texttt{exponential\_distribution} \[\text{rand.dist.pois.exp}\]

An \texttt{exponential\_distribution} random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$p(x \mid \lambda) = \lambda e^{-\lambda x}.$$ 

```cpp
template<class RealType = double>
class exponential_distribution
{
public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit exponential_distribution(RealType lambda = 1.0);
};
```

\section*{§ 26.5.8.4.2}
explicit exponential_distribution(const param_type& parm);
void reset();

// generating functions
template<class URNG>
  result_type operator()(URNG& g);
template<class URNG>
  result_type operator()(URNG& 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);

2 Requires: \(0 < \lambda\).
3 Effects: Constructs a exponential_distribution object; \(\lambda\) corresponds to the parameter of the distribution.

RealType lambda() const;

4 Returns: The value of the \(\lambda\) parameter with which the object was constructed.

26.5.8.4.3 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
typedef RealType result_type;
typedef unspecified param_type;

    // 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 URNG>
      result_type operator()(URNG& g);
    template<class URNG>
      result_type operator()(URNG& g, const param_type& parm);

    // property functions

\§ 26.5.8.4.3
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.

26.5.8.4.4 Class template weibull_distribution

A weibull_distribution random number distribution produces random numbers \( x \geq 0 \) distributed according to the probability density function

\[
p(x \mid a, b) = \frac{a}{b} \cdot \left(\frac{x}{b}\right)^{a-1} \cdot \exp \left(-\left(\frac{x}{b}\right)^a\right).
\]

template<class RealType = double>
class weibull_distribution
{
  public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    explicit weibull_distribution(RealType a = 1.0, RealType b = 1.0);
    explicit weibull_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URNG>
    result_type operator()(URNG& g);
    template<class URNG>
    result_type operator()(URNG& g, const param_type& parm);

    // property functions
    RealType a() const;
    RealType b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
explicit weibull_distribution(RealType a = 1.0, RealType b = 1.0);

2 Requires: $0 < a$ and $0 < b$.

3 Effects: Constructs a weibull_distribution object; $a$ and $b$ correspond to the respective parameters of the distribution.

4 RealType a() const;

5 Returns: The value of the $a$ parameter with which the object was constructed.

5 RealType b() const;

5 Returns: The value of the $b$ parameter with which the object was constructed.

26.5.8.4.5 Class template extreme_value_distribution

1 An extreme_value_distribution random number distribution produces random numbers $x$ distributed according to the probability density function

$$p(x | a, b) = \frac{1}{b} \cdot \exp \left( \frac{a - x}{b} - \exp \left( \frac{a - x}{b} \right) \right).$$

template<class RealType = double>
class extreme_value_distribution
{
public:

// types
typedef RealType result_type;
typedef unspecified param_type;

// 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 URNG>
result_type operator()(URNG& g);
template<class URNG>
result_type operator()(URNG& 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);

---

280) The distribution corresponding to this probability density function is also known (with a possible change of variable) as the Gumbel Type I, the log-Weibull, or the Fisher-Tippett Type I distribution.
Requires: 0 < b.

Effects: Constructs an extreme_value_distribution object; a and b correspond to the respective parameters of the distribution.

RealType a() const;

Returns: The value of the a parameter with which the object was constructed.

RealType b() const;

Returns: The value of the b parameter with which the object was constructed.

26.5.8.5 Normal distributions

26.5.8.5.1 Class template normal_distribution

A normal_distribution random number distribution produces random numbers x distributed according to the probability density function

\[ p(x | \mu, \sigma) = \frac{1}{\sigma \sqrt{2\pi}} \exp \left( -\frac{(x - \mu)^2}{2\sigma^2} \right) . \]

The distribution parameters \( \mu \) and \( \sigma \) are also known as this distribution's mean and standard deviation.

template<class RealType = double>

class normal_distribution
{
public:

    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // 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 URNG>
        result_type operator()(URNG& g);
    template<class URNG>
        result_type operator()(URNG& 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.

§ 26.5.8.5.1

970
RealType mean() const;

   Returns: The value of the mean parameter with which the object was constructed.

RealType stddev() const;

   Returns: The value of the stddev parameter with which the object was constructed.

26.5.8.5.2 Class template lognormal_distribution [rand.dist.norm.lognormal]

A lognormal_distribution random number distribution produces random numbers \( x > 0 \) distributed according to the probability density function

\[
p(x \mid m, s) = \frac{1}{sx\sqrt{2\pi}} \cdot \exp \left( -\frac{(\ln x - m)^2}{2s^2} \right).
\]

template<class RealType = double>
   class lognormal_distribution
   {
   public:
      // types
      typedef RealType result_type;
      typedef unspecified param_type;

      // 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 URNG>
         result_type operator()(URNG& g);
      template<class URNG>
         result_type operator()(URNG& g, const param_type& parm);

      // property functions
      RealType m() const;
      RealType s() const;
      param_type param() const;
      void param(const param_type& parm);
      result_type min() const;
      result_type max() const;
   };

explicit lognormal_distribution(RealType m = 0.0, RealType s = 1.0);

   Requires: 0 < s.

   Effects: Constructs a lognormal_distribution object; \( m \) and \( s \) correspond to the respective parameters of the distribution.

RealType m() const;

   Returns: The value of the \( m \) parameter with which the object was constructed.

RealType s() const;

   Returns: The value of the \( s \) parameter with which the object was constructed.
26.5.8.5.3 Class template chi_squared_distribution

A \texttt{chi\_squared\_distribution} random number distribution produces random numbers \( x > 0 \) distributed according to the probability density function

\[
p(x \mid n) = \frac{x^{(n/2)-1} e^{-x/2}}{\Gamma(n/2) \cdot 2^{n/2}}.
\]

\[
template<class RealType = double>
\]

\[
\text{class chi\_squared\_distribution}
\]

\[
\{\text{public:}
\]

\[
\text{// types}
\]

\[
typedef RealType result\_type;
\]

\[
typedef unspecified param\_type;
\]

\[
\text{// constructor and reset functions}
\]

\[
\text{explicit chi\_squared\_distribution(RealType n = 1);} ;
\]

\[
\text{explicit chi\_squared\_distribution(const param\_type& parm);} ;
\]

\[
\text{void reset();}
\]

\[
\text{// generating functions}
\]

\[
\text{template<class URNG>}
\]

\[
\text{result\_type operator()}(URNG& g);
\]

\[
\text{template<class URNG>}
\]

\[
\text{result\_type operator()}(URNG& g, const param\_type& parm);
\]

\[
\text{// property functions}
\]

\[
\text{RealType n() const;}
\]

\[
\text{param\_type param() const;}
\]

\[
\text{void param(const param\_type& parm);} ;
\]

\[
\text{result\_type min() const;}
\]

\[
\text{result\_type max() const;}
\]

\[
\}
\]

\[
\text{explicit chi\_squared\_distribution(RealType n = 1);} ;
\]

\[
\text{Requires: } 0 < n.
\]

\[
\text{Effects: } \text{Constructs a } \texttt{chi\_squared\_distribution} \text{ object; } n \text{ corresponds to the parameter of the distribution.}
\]

\[
\text{RealType n() const;}
\]

\[
\text{Returns: } \text{The value of the } n \text{ parameter with which the object was constructed.}
\]

26.5.8.5.4 Class template cauchy_distribution

A \texttt{cauchy\_distribution} random number distribution produces random numbers \( x \) distributed according to the probability density function

\[
p(x \mid a, b) = \left( \frac{\pi b}{1 + \left( \frac{x - a}{b} \right)^2} \right)^{-1}.
\]
template<class RealType = double>
class cauchy_distribution
{
public:
  // types
  typedef RealType result_type;
  typedef unspecified param_type;

  // 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 URNG>
  result_type operator()(URNG& g);
  template<class URNG>
  result_type operator()(URNG& 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.

26.5.8.5.5 Class template fisher_f_distribution
1  A fisher_f_distribution random number distribution produces random numbers \( x \geq 0 \) distributed according to the probability density function

\[
p(x \mid m, n) = \frac{\Gamma((m + n)/2)}{\Gamma(m/2) \Gamma(n/2)} \left( \frac{m}{n} \right)^{m/2} x^{(m/2)-1} \left( 1 + \frac{m x}{n} \right)^{-(m+n)/2}.
\]

template<class RealType = double>
class fisher_f_distribution
{
public:
  // types
  typedef RealType result_type;
typedef unspecified param_type;

// 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 URNG>
result_type operator()(URNG& g);
template<class URNG>
result_type operator()(URNG& g, const param_type& parm);

// property functions
RealType m() const;
RealType n() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;

explicit fisher_f_distribution(RealType m = 1, RealType n = 1);

2 Requires: 0 < m and 0 < n.
3 Effects: Constructs a fisher_f_distribution object; m and n 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 n() const;
5 Returns: The value of the n parameter with which the object was constructed.

26.5.8.5.6 Class template student_t_distribution [rand.dist.norm.t]
1 A student_t_distribution random number distribution produces random numbers x distributed according to the probability density function

\[ p(x | n) = \frac{1}{\sqrt{n\pi}} \cdot \frac{\Gamma((n + 1)/2)}{\Gamma(n/2)} \cdot \left(1 + \frac{x^2}{n}\right)^{-(n+1)/2}. \]

template<class RealType = double>
class student_t_distribution
{
    public:
        // types
        typedef RealType result_type;
        typedef unspecified param_type;

        // constructor and reset functions
        explicit student_t_distribution(RealType n = 1);
        explicit student_t_distribution(const param_type& parm);
        void reset();

§ 26.5.8.5.6
// generating functions
template<class URNG>
result_type operator()(URNG& g);
template<class URNG>
result_type operator()(URNG& 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);

Requires: 0 < n.

Effects: Constructs a student_t_distribution object; n corresponds to the parameter of the distribution.

RealType n() const;

Returns: The value of the n parameter with which the object was constructed.

26.5.8.6 Sampling distributions  [rand.dist.samp]
26.5.8.6.1 Class template discrete_distribution  [rand.dist.samp.discrete]

A discrete_distribution random number distribution produces random integers $i$, $0 \leq i < n$, distributed according to the discrete probability function

$$P(i|p_0, \ldots, p_{n-1}) = p_i.$$  

Unless specified otherwise, the distribution parameters are calculated as: $p_k = w_k/S$ for $k = 0, \ldots, n-1$, in which the values $w_k$, commonly known as the weights, shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold: $0 < S = w_0 + \cdots + w_{n-1}$.

template<class IntType = int>
class discrete_distribution
{
 public:
 // types
typedef IntType result_type;
typedef unspecified param_type;

 // 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();

§ 26.5.8.6.1
// generating functions
template<class URNG>
  result_type operator()(URNG& g);
template<class URNG>
  result_type operator()(URNG& g, const param_type& parm);

// property functions
vector<double> probabilities() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;
};

discrete_distribution();

Effects: Constructs a discrete_distribution object with n = 1 and p0 = 1. [Note: Such an object will always deliver the value 0. — end note]

template<class InputIterator>
  discrete_distribution(InputIterator firstW, InputIterator lastW);

Requires: InputIterator shall satisfy the requirements of an input iterator (Table 107) type. Moreover, iterator_traits<InputIterator>::value_type shall denote a type that is convertible to double. If firstW == lastW, let n = 1 and w0 = 1. Otherwise, (firstW, lastW) shall form a sequence w of length n > 0.

Effects: Constructs a discrete_distribution object with probabilities given by the formula above.

discrete_distribution(initializer_list<double> wl);

Effects: Same as discrete_distribution(wl.begin(), wl.end()).

template<class UnaryOperation>
  discrete_distribution(size_t nw, double xmin, double xmax, UnaryOperation fw);

Requires: Each instance of type UnaryOperation shall be a function object (20.9) whose return type shall be convertible to double. Moreover, double shall be convertible to the type of UnaryOperation’s sole parameter. If nw = 0, let n = 1, otherwise let n = nw. The relation 0 < δ = (xmax − xmin)/n shall hold.

Effects: Constructs a discrete_distribution object with probabilities given by the formula above, using the following values: If nw = 0, let w0 = 1. Otherwise, let wk = fw(xmin + k · δ + δ/2) for k = 0, ..., n − 1.

Complexity: The number of invocations of fw shall not exceed n.

vector<double> probabilities() const;

Returns: A vector<double> whose size member returns n and whose operator[] member returns pk when invoked with argument k for k = 0, ..., n − 1.

26.5.8.6.2 Class template piecewise_constant_distribution [rand.dist.samp.pconst]

A piecewise_constant_distribution random number distribution produces random numbers x, b0 ≤ x < bn, uniformly distributed over each subinterval [bi, bi+1) according to the probability density function

\[ p(x \mid b_0, \ldots, b_n, \rho_0, \ldots, \rho_{n-1}) = \rho_i, \text{ for } b_i \leq x < b_{i+1}. \]
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:

\[
\rho_k = \frac{w_k}{S \cdot (b_{k+1} - b_k)} \quad \text{for} \quad k = 0, \ldots, n-1,
\]

in which the values \( w_k \), commonly known as the "weights", shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold: \( 0 < S = w_0 + \cdots + w_{n-1} \).

```cpp
template<class RealType = double>
class piecewise_constant_distribution
{
public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // 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 URNG>
    result_type operator()(URNG& g);
    template<class URNG>
    result_type operator()(URNG& g, const param_type& parm);

    // property functions
    vector<result_type> intervals() const;
    vector<result_type> densities() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

```
Effects: Constructs a `piecewise_constant_distribution` object with \( n = 1, \rho_0 = 1, b_0 = 0, \) and \( b_1 = 1 \).
```

```
Requires: `InputIteratorB` and `InputIteratorW` shall each satisfy the requirements of an input iterator (Table 107) 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 ≥ n shall be ignored by the distribution.

Effects: Constructs a piecewise_constant_distribution object with parameters as specified above.

template<class UnaryOperation>

piecewise_constant_distribution(initializer_list<RealType> bl, UnaryOperation fw);

Requires: Each instance of type UnaryOperation shall be a function object (20.9) 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, ..., b_n, and let w_k = fw((b_{k+1} + b_k)/2) for k = 0, ..., n-1.

Complexity: The number of invocations of fw shall not exceed n.

template<class UnaryOperation>

piecewise_constant_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);

Requires: Each instance of type UnaryOperation shall be a function object (20.9) whose return type shall be convertible to double. Moreover, double shall be convertible to the type of UnaryOperation's sole parameter. If nw = 0, let n = 1, otherwise let n = nw. The relation 0 < δ = (xmax - xmin)/n shall hold.

Effects: Constructs a piecewise_constant_distribution object with parameters taken or calculated from the following values: Let b_k = xmin + k·δ for k = 0, ..., n, and w_k = fw(b_k + δ/2) for k = 0, ..., n-1.

Complexity: The number of invocations of fw shall not exceed n.

vector<result_type> intervals() const;

Returns: A vector<result_type> whose size member returns n + 1 and whose operator[] member returns b_k when invoked with argument k for k = 0, ..., n.

vector<result_type> densities() const;

Returns: A vector<result_type> whose size member returns n and whose operator[] member returns ρ_k when invoked with argument k for k = 0, ..., n - 1.

### 26.5.8.6.3 Class template piecewise_linear_distribution

A piecewise_linear_distribution random number distribution produces random numbers x, b_0 ≤ x < b_n, distributed over each subinterval [b_i, b_{i+1}) according to the probability density function

\[ p(x \mid b_0, \ldots, b_n, \rho_0, \ldots, \rho_n) = \rho_i \cdot \frac{b_{i+1} - x}{b_{i+1} - b_i} + \rho_{i+1} \cdot \frac{x - b_i}{b_{i+1} - b_i}, \text{ for } b_i ≤ x < b_{i+1}. \]

The n + 1 distribution parameters b_i, also known as this distribution’s interval boundaries, shall satisfy the relation b_i < b_{i+1} for i = 0, ..., n-1. Unless specified otherwise, the remaining n + 1 distribution parameters are calculated as \( \rho_k = w_k/S \) for k = 0, ..., 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
    typedef RealType result_type;
    typedef unspecified param_type;

    // 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 URNG>
    result_type operator()(URNG& g);
    template<class URNG>
    result_type operator()(URNG& 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;
};
Requires: Each instance of type UnaryOperation shall be a function object (20.9) whose return type shall be convertible to double. Moreover, double shall be convertible to the type of UnaryOperation’s sole parameter.

Effects: Constructs a piecewise_linear_distribution object with parameters taken or calculated from the following values: If bl.size() < 2, let n = 1, \( \rho_0 = \rho_1 = 1, b_0 = 0, \) and \( b_1 = 1. \) Otherwise, let \( \{\text{bl.begin()}, \text{bl.end()}\} \) 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 \( n + 1. \)

```cpp
template<class UnaryOperation>
piecewise_linear_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);
```

Requires: Each instance of type UnaryOperation shall be a function object (20.9) whose return type shall be convertible to double. Moreover, double shall be convertible to the type of UnaryOperation’s sole parameter. If \( n w = 0 \), let \( n = 1 \), otherwise let \( n = n w. \) The relation \( 0 < \delta = (x m a x - x m i n)/n \) shall hold.

Effects: Constructs a piecewise_linear_distribution object with parameters taken or calculated from the following values: Let \( b_k = x m i n + k \cdot \delta \) for \( k = 0, \ldots, n, \) and \( w_k = f w(b_k) \) for \( k = 0, \ldots, n. \)

Complexity: The number of invocations of \( f w \) shall not exceed \( n + 1. \)

```cpp
vector<result_type> intervals() const;
```

Returns: A vector<result_type> whose size member returns \( n + 1 \) and whose operator[] member returns \( b_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n. \)

```cpp
vector<result_type> densities() const;
```

Returns: A vector<result_type> whose size member returns \( n \) and whose operator[] member returns \( \rho_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n. \)

### 26.6 Numeric arrays

#### 26.6.1 Header <valarray> synopsis

```cpp
#include <valarray>
```

namespace std {

```cpp
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;

```cpp
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 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>&);
```

§ 26.6.1 980
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<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>&);

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>&);

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>&);

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>&);

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 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>&);

The header `<valarray>` defines five class templates (`valarray`, `slice_array`, `gslice_array`, `mask_array`, and `indirect_array`), two classes (`slice` and `gslice`), and a series of related function templates for representing and manipulating arrays of values.

The `valarray` array classes are defined to be free of certain forms of aliasing, thus allowing operations on these classes to be optimized.

Any function returning a `valarray<T>` is permitted to return an object of another type, provided all the const member functions of `valarray<T>` are also applicable to this type. This return type shall not add more than two levels of template nesting over the most deeply nested argument type.

Clause 18.3.2 recommends a minimum number of recursively nested template instantiations. This requirement thus indirectly suggests a minimum allowable complexity for valarray expressions.
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` (26.6.10), 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 computed 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` (18.6.2.1) exception if there are not sufficient resources available to carry out the operation. Note that the exception is not mandated.

### 26.6.2 Class template `valarray` [template.valarray]

#### 26.6.2.1 Class template `valarray` overview [template.valarray.overview]

```cpp
namespace std {
  template<class T> class valarray {
  public:
    typedef T value_type;

    // 26.6.2.2 construct/destroy:
    valarray();
    explicit valarray(size_t);
    valarray(const T&, size_t);
    valarray(const T*, size_t);
    valarray(const valarray&);
    valarray(valarray&&) noexcept;
    valarray(const slice_array<T>&);
    valarray(const gslice_array<T>&);
    valarray(const mask_array<T>&);
    valarray(const indirect_array<T>&);
    valarray(initializer_list<T>);
    ~valarray();

    // 26.6.2.3 assignment:
    valarray<T>& operator=(const valarray<T>&);
    valarray<T>& operator=(valarray<T>&&) noexcept;
    valarray<T>& operator=(initializer_list<T>);
    valarray<T>& operator=(const T&);
    valarray<T>& operator=(const slice_array<T>&);
    valarray<T>& operator=(const gslice_array<T>&);
    valarray<T>& operator=(const mask_array<T>&);
    valarray<T>& operator=(const indirect_array<T>&);

    // 26.6.2.4 element access:
    const T& operator[](size_t) const;
    T& operator[](size_t);

    // 26.6.2.5 subset operations:
    valarray<T> operator[](slice) const;
    slice_array<T> operator[](slice);
    valarray<T> operator[](const gslice&) const;
```

§ 26.6.2.1
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. 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.\textsuperscript{282}

An implementation is permitted to qualify any of the functions declared in `<valarray>` as \textcode{inline}.

\textbf{26.6.2.2 valarray constructors} \hfill [valarray.cons]

\begin{enumerate}
  \item \textcode{valarray();}
    \begin{itemize}
      \item \textit{Effects:} Constructs an object of class \textcode{valarray<T>\textsuperscript{283}} which has zero length.\textsuperscript{284}
    \end{itemize}
  \item \textcode{explicit valarray(size_t)}
    \begin{itemize}
      \item The array created by this constructor has a length equal to the value of the argument. The elements of the array are value-initialized (8.5).
    \end{itemize}
  \item \textcode{valarray(const T&, size_t)}
    \begin{itemize}
      \item The array created by this constructor has a length equal to the second argument. The elements of the array are initialized with the value of the first argument.
    \end{itemize}
  \item \textcode{valarray(const T*, size_t)}
    \begin{itemize}
      \item The array created by this constructor has a length equal to the second argument \textcode{n}. The values of the elements of the array are initialized with the first \textcode{n} values pointed to by the first argument.\textsuperscript{285} If the value of the second argument is greater than the number of values pointed to by the first argument, the behavior is undefined.
    \end{itemize}
  \item \textcode{valarray(const valarray<T>&)}
    \begin{itemize}
      \item The array created by this constructor has the same length as the argument array. The elements are initialized with the values of the corresponding elements of the argument array.\textsuperscript{286}
    \end{itemize}
  \item \textcode{valarray(valarray<T>&& v) noexcept}
    \begin{itemize}
      \item The array created by this constructor has the same length as the argument array. The elements are initialized with the values of the corresponding elements of the argument array.
      \begin{itemize}
        \item \textit{Complexity:} Constant.
      \end{itemize}
    \end{itemize}
  \item \textcode{valarray(initializer_list<T> il)}
    \begin{itemize}
      \item \textit{Effects:} Same as \textcode{valarray(il.begin(), il.size())}.
    \end{itemize}
  \item \textcode{valarray(const slice_array<T>&)}
  \item \textcode{valarray(const gslice_array<T>&)}
  \item \textcode{valarray(const mask_array<T>&)}
  \item \textcode{valarray(const indirect_array<T>&)}
    \begin{itemize}
      \item These conversion constructors convert one of the four reference templates to a \textcode{valarray}.
    \end{itemize}
  \item \textcode{~valarray();}
    \begin{itemize}
      \item The destructor is applied to every element of \textcode{*this}; an implementation may return all allocated memory.
    \end{itemize}
\end{enumerate}

\textsuperscript{282} The intent is to specify an array template that has the minimum functionality necessary to address aliasing ambiguities and the proliferation of temporaries. Thus, the \textcode{valarray} template is neither a matrix class nor a field class. However, it is a very useful building block for designing such classes.

\textsuperscript{283} For convenience, such objects are referred to as “arrays” throughout the remainder of 26.6.

\textsuperscript{284} This default constructor is essential, since arrays of \textcode{valarray} may be useful. After initialization, the length of an empty array can be increased with the \textcode{resize} member function.

\textsuperscript{285} This constructor is the preferred method for converting a C array to a \textcode{valarray} object.

\textsuperscript{286} 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.
26.6.2.3 \texttt{valarray} assignment

\begin{verbatim}
valarray\langle T\rangle\& \text{ operator\texttt{=}(const\ valarray\langle T\rangle\& v);} \\
1 Each element of the \texttt{*this} array is assigned the value of the corresponding element of the argument array. If the length of \texttt{v} is not equal to the length of \texttt{*this}, resizes \texttt{*this} to make the two arrays the same length, as if by calling \texttt{resize(v.size())}, before performing the assignment.
2 \textit{Postcondition:} size() == v.size().

valarray\langle T\rangle\& \text{ operator\texttt{=}(valarray\langle T\rangle\&\& v) noexcept;} \\
3 Effects: \texttt{*this} obtains the value of \texttt{v}. The value of \texttt{v} after the assignment is not specified.
4 Complexity: Linear.

valarray\& \text{ operator\texttt{=}(initializer\_list\langle T\rangle il);} \\
5 Effects: \texttt{*this} = valarray(il).
6 Returns: \texttt{*this}.

valarray\langle T\rangle\& \text{ operator\texttt{=}(const\ T\&);} \\
7 The scalar assignment operator causes each element of the \texttt{*this} array to be assigned the value of the argument.

valarray\langle T\rangle\& \text{ operator\texttt{=}(const\ slice\_array\langle T\rangle\&);} \\
valarray\langle T\rangle\& \text{ operator\texttt{=}(const\ gslice\_array\langle T\rangle\&);} \\
valarray\langle T\rangle\& \text{ operator\texttt{=}(const\ mask\_array\langle T\rangle\&);} \\
valarray\langle T\rangle\& \text{ operator\texttt{=}(const\ indirect\_array\langle T\rangle\&);} \\
8 Requires: The length of the array to which the argument refers equals size().
9 These operators allow the results of a generalized subscripting operation to be assigned directly to a \texttt{valarray}.
10 If the value of an element in the left-hand side of a \texttt{valarray} assignment operator depends on the value of another element in that left-hand side, the resulting behavior is undefined.
\end{verbatim}

26.6.2.4 \texttt{valarray} element access

\begin{verbatim}
const\ T\& \text{ operator\texttt{[]}(size_t) const;} \\
T\& \text{ operator\texttt{[]}\texttt{(size}_t\texttt{)}};
1 The subscript operator returns a reference to the corresponding element of the array.
2 Thus, the expression \texttt{(a[i] = q, a[i]) == q} evaluates as true for any non-constant \texttt{valarray\langle T\rangle} a, any T q, and for any size_t i such that the value of i is less than the length of a.
3 The expression \texttt{&a[i+j] == &a[i] + j} evaluates as true for all size_t i and size_t j such that i+j is less than the length of the array a.
4 Likewise, the expression \texttt{&a[i] \&!= &b[j]} evaluates as true for any two arrays a and b and for any size_t i and size_t j such that i is less than the length of a and j is less than the length of b. This property indicates an absence of aliasing and may be used to advantage by optimizing compilers.\footnote{Compilers may take advantage of inlining, constant propagation, loop fusion, tracking of pointers obtained from \texttt{operator new}, and other techniques to generate efficient \texttt{valarrays.}}
5 The reference returned by the subscript operator for an array shall be valid until the member function \texttt{resize(size}_t\texttt{, T)} (26.6.2.8) is called for that array or until the lifetime of that array ends, whichever happens first.
6 If the subscript operator is invoked with a size_t argument whose value is not less than the length of the array, the behavior is undefined.
\end{verbatim}
26.6.2.5 valarray subset operations

The member operator[] is overloaded to provide several ways to select sequences of elements from among those controlled by *this. Each of these operations returns a subset of the array. The const-qualified versions return this subset as a new valarray object. The non-const versions return a class template object which has reference semantics to the original array, working in conjunction with various overloads of operator= and other assigning operators to allow selective replacement (slicing) of the controlled sequence. In each case the selected element(s) must exist.

valarray<T> operator[](slice slicearr) const;

Returns: An object of class valarray<T> containing those elements of the controlled sequence designated by slicearr. [Example:
   const valarray<char> v0("abcdefhijklmnop", 16);
   // v0[slice(2, 5, 3)] returns valarray<char>("cfilo", 5)
   — end example]

slice_array<T> operator[](slice slicearr);

Returns: An object that holds references to elements of the controlled sequence selected by slicearr. [Example:
   valarray<char> v0("abcdefhijklmnop", 16);
   valarray<char> v1("ABCDE", 5);
   v0[slice(2, 5, 3)] = v1;
   // v0 == valarray<char>("abAdeBghCjkDmnEp", 16);
   — end example]

valarray<T> operator[](const gslice& gslicearr) const;

Returns: An object of class valarray<T> containing those elements of the controlled sequence designated by gslicearr. [Example:
   const valarray<char> v0("abcdefhijklmnop", 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. [Example:
   valarray<char> v0("abcdefhijklmnop", 16);
   valarray<char> v1("ABCDE", 5);
   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>("abcAeBghCijDlEnFp", 16)
   — end example]
valarray<T> operator[](const valarray<bool>& boolarr) const;

Returns: An object of class valarray<T> containing those elements of the controlled sequence designated by boolarr. [Example:

const valarray<char> v0("abcdefgijklmnop", 16);
const bool vb[] = { false, false, true, true, false, true };  // returns
// v0[valarray<bool>(vb, 6)] returns
// v0 = 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<char>("abABeCghijklmnop", 16)
// v0[valarray<bool>(vb, 6)] = v1;
// v0 == valarray<char>("abABeCghijklmnop", 16)

— end example]

valarray<T> operator[](const valarray<size_t>& indarr) const;

Returns: An object of class valarray<T> 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 };  // returns
// v0[valarray<size_t>(vi, 5)] returns
// v0 = 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<char>("abCDeBgAEjklmnop", 16)
// v0[valarray<size_t>(vi, 5)] = v1;
// v0 == valarray<char>("abCDeBgAEjklmnop", 16)

— end example]

26.6.2.6 valarray unary operators [valarray.unary]

valarray<T> operator+() const;
valarray<T> operator-() const;
valarray<T> operator-() const;
valarray<bool> operator!() const;

§ 26.6.2.6 988
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 \( T \) (\texttt{bool} for \texttt{operator!}) or which may be unambiguously implicitly converted to type \( T \) (\texttt{bool} for \texttt{operator!}).

Each of these operators returns an array whose length is equal to the length of the array. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array.

### 26.6.2.7 valarray computed assignment

[\texttt{valarray.cassign}]

```cpp
valarray<T>& operator*= (const valarray<T>&);  
valarray<T>& operator/= (const valarray<T>&);  
valarray<T>& operator%= (const valarray<T>&);  
valarray<T>& operator+= (const valarray<T>&);  
valarray<T>& operator-= (const valarray<T>&);  
valarray<T>& operator^= (const valarray<T>&);  
valarray<T>& operator&= (const valarray<T>&);  
valarray<T>& operator|= (const valarray<T>&);  
valarray<T>& operator<<=(const valarray<T>&);  
valarray<T>& operator>>=(const valarray<T>&);  
```

Each of these operators may only be instantiated for a type \( T \) to which the indicated operator can be applied. Each of these operators performs the indicated operation on each of its elements and the corresponding element of the argument array.

The array is then returned by reference.

If the array and the argument array do not have the same length, the behavior is undefined. The appearance of an array on the left-hand side of a computed assignment does \textit{not} invalidate references or pointers.

If the value of an element in the left-hand side of a valarray computed assignment operator depends on the value of another element in that left hand side, the resulting behavior is undefined.

```cpp
valarray<T>& operator*= (const T&);  
valarray<T>& operator/= (const T&);  
valarray<T>& operator%= (const T&);  
valarray<T>& operator+= (const T&);  
valarray<T>& operator-= (const T&);  
valarray<T>& operator^= (const T&);  
valarray<T>& operator&= (const T&);  
valarray<T>& operator|= (const T&);  
valarray<T>& operator<<=(const T&);  
valarray<T>& operator>>=(const T&);  
```

Each of these operators may only be instantiated for a type \( T \) to which the indicated operator can be applied.

Each of these operators applies the indicated operation to each element of the array and the non-array argument.

The array is then returned by reference.

The appearance of an array on the left-hand side of a computed assignment does \textit{not} invalidate references or pointers to the elements of the array.

### 26.6.2.8 valarray member functions

[\texttt{valarray.members}]

```cpp
void swap(valarray& v) noexcept;
```
Effects: \*this obtains the value of v. v obtains the value of \*this.

Complexity: Constant.

size_t size() const;

Returns: The number of elements in the array.

Complexity: Constant time.

T sum() const;

This function may only be instantiated for a type T to which operator\+= can be applied. This function returns the sum of all the elements of the array.

If the array has length 0, the behavior is undefined. If the array has length 1, sum() returns the value of element 0. Otherwise, the returned value is calculated by applying operator\+= to a copy of an element of the array and all other elements of the array in an unspecified order.

T min() const;

This function returns the minimum value contained in \*this. The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using operator<.

T max() const;

This function returns the maximum value contained in \*this. The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using operator<.

valarray<T> shift(int n) const;

This function returns an object of class valarray<T> of length size(), each of whose elements I is (\*this)[I + n] if I + n is non-negative and less than size(), otherwise T(). Thus if element zero is taken as the leftmost element, a positive value of n shifts the elements left n places, with zero fill.

[Example: If the argument has the value -2, the first two elements of the result will be value-initialized (8.5); the third element of the result will be assigned the value of the first element of the argument; etc. — end example]

valarray<T> cshift(int n) const;

This function returns an object of class valarray<T> 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<T> apply(T func(T)) const;
valarray<T> apply(T func(const T&)) const;

These functions return an array whose length is equal to the array. Each element of the returned array is assigned the value returned by applying the argument function to the corresponding element of the array.

void resize(size_t sz, T c = T());

This member function changes the length of the \*this array to sz and then assigns to each element the value of the second argument. Resizing invalidates all pointers and references to elements in the array.
26.6.3  valarray non-member operations

26.6.3.1  valarray binary operators

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 \( T \).

Each of these operators returns an array 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.

If the argument arrays do not have the same length, the behavior is undefined.

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
Each of these operators returns an array whose length is equal to the length of the array argument. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array argument and the non-array argument.

26.6.3.2 valarray logical operators

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.

Each of these operators returns a bool array 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.

If the two array arguments do not have the same length, the behavior is undefined.
26.6.3.3 valarray transcendental functions

*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>&);*

1 Each of these functions may only be instantiated for a type T to which a unique function with the indicated name can be applied (unqualified). This function shall return a value which is of type T or which can be unambiguously implicitly converted to type T.

26.6.3.4 valarray specialized algorithms

*template<class T> void swap(valarray<T>& x, valarray<T>& y) noexcept;

Effects: x.swap(y).

1

26.6.4 Class slice

26.6.4.1 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.288

26.6.4.2 slice constructors

288) 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.
slice();
slice(size_t start, size_t length, size_t stride);
slice(const slice&);

1 The default constructor is equivalent to slice(0, 0, 0). A default constructor is provided only to permit the declaration of arrays of slices. The constructor with arguments for a slice takes a start, length, and stride parameter.

2 [Example: slice(3, 8, 2) constructs a slice which selects elements 3, 5, 7, ... 17 from an array. —end example]

26.6.4.3 slice access functions [slice.access]

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.

26.6.5 Class template slice_array [template.slice.array]

26.6.5.1 Class template slice_array overview [template.slice.array.overview]

namespace std {
    template <class T> class slice_array {
        public:
            typedef T value_type;

            void operator= (const valarray<T>&) const;
            void operator*= (const valarray<T>&) const;
            void operator/= (const valarray<T>&) const;
            void operator%= (const valarray<T>&) const;
            void operator+= (const valarray<T>&) const;
            void operator-= (const valarray<T>&) const;
            void operator^= (const valarray<T>&) const;
            void operator&= (const valarray<T>&) const;
            void operator|= (const valarray<T>&) const;
            void operator<<=(const valarray<T>&) const;
            void operator>>=(const valarray<T>&) const;

            slice_array(const slice_array&);
            ~slice_array();
            const slice_array& operator=(const slice_array&) const;
            void operator=(const T&) const;

            slice_array() = delete; // as implied by declaring copy constructor above
    };
}

1 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.

2 [Example: The expression a[slice(1, 5, 3)] = b; has the effect of assigning the elements of b to a slice of the elements in a. For the slice shown, the elements selected from a are 1, 4, ..., 13. —end example]

26.6.5.2 slice_array assignment [slice.arr.assign]

§ 26.6.5.2
void operator=(const valarray<T>&) const;
const slice_array& operator=(const slice_array&) const;

1 These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the \texttt{valarray<T>} object to which the \texttt{slice_array} object refers.

26.6.5.3 \texttt{slice_array} computed 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;

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the \texttt{valarray<T>} object to which the \texttt{slice_array} object refers.

26.6.5.4 \texttt{slice_array} fill function

void operator=(const T&) const;

1 This function has reference semantics, assigning the value of its argument to the elements of the \texttt{valarray<T>} object to which the \texttt{slice_array} object refers.

26.6.6 The \texttt{gslice} class

26.6.6.1 The \texttt{gslice} class overview

namespace std {
    class gslice {
    public:
        gslice();
        gslice(size_t s, const valarray<size_t>& l, const valarray<size_t>& d);

        size_t     start() const;
        valarray<size_t> size() const;
        valarray<size_t> stride() const;
    };
}

1 This class represents a generalized slice out of an array. A \texttt{gslice} is defined by a starting offset \((s)\), a set of lengths \((l_j)\), and a set of strides \((d_j)\). The number of lengths shall equal the number of strides.

2 A \texttt{gslice} represents a mapping from a set of indices \((i_j)\), equal in number to the number of strides, to a single index \(k\). It is useful for building multidimensional array classes using the \texttt{valarray} template, which is one-dimensional. The set of one-dimensional index values specified by a \texttt{gslice} are

\[
k = s + \sum_j i_j d_j
\]

where the multidimensional indices \(i_j\) range in value from 0 to \(l_{ij} - 1\).

3 [\textit{Example: The \texttt{gslice} specification}]

§ 26.6.6.1
start = 3
length = \{2, 4, 3\}
stride = \{19, 4, 1\}

yields the sequence of one-dimensional indices

\[ k = 3 + (0, 1) \times 19 + (0, 1, 2, 3) \times 4 + (0, 1, 2) \times 1 \]

which are ordered as shown in the following table:

<table>
<thead>
<tr>
<th>(i_0)</th>
<th>(i_1)</th>
<th>(i_2)</th>
<th>(k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>36</td>
</tr>
</tbody>
</table>

That is, the highest-ordered index turns fastest. —end example]  

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

<table>
<thead>
<tr>
<th>(i_0)</th>
<th>(i_1)</th>
<th>(i_2)</th>
<th>(k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

—end example]

If a degenerate slice is used as the argument to the non-const version of `operator[]`(const gslice&), the resulting behavior is undefined.

26.6.6.2 gslice constructors

```cpp
gslice();
gslice(size_t start, const valarray<size_t>& lengths,
    const valarray<size_t>& strides);
gslice(const gslice&);
```
The default constructor is equivalent to `gslice(0, valarray<size_t>(), valarray<size_t>())`. The constructor with arguments builds a `gslice` based on a specification of start, lengths, and strides, as explained in the previous section.

### 26.6.6.3 `gslice` access functions

```cpp
size_t start() const;
valarray<size_t> size() const;
valarray<size_t> stride() const;
```

1. **Returns:** The representation of the start, lengths, or strides specified for the `gslice`.
2. **Complexity:** `start()` is constant time. `size()` and `stride()` are linear in the number of strides.

### 26.6.7 Class template `gslice_array`

#### 26.6.7.1 Class template `gslice_array` overview

```cpp
namespace std {
    template <class T> class gslice_array {
        public:
            typedef T value_type;

            void operator= (const valarray<T>&) const;
            void operator*= (const valarray<T>&) const;
            void operator/= (const valarray<T>&) const;
            void operator%= (const valarray<T>&) const;
            void operator+= (const valarray<T>&) const;
            void operator-= (const valarray<T>&) const;
            void operator^= (const valarray<T>&) const;
            void operator&= (const valarray<T>&) const;
            void operator|= (const valarray<T>&) const;
            void operator<<=(const valarray<T>&) const;
            void operator>>=(const valarray<T>&) const;

            gslice_array(const gslice_array&);
            ~gslice_array();
            const gslice_array& operator=(const gslice_array&) const;
            void operator=(const T&) const;
            gslice_array() = delete;  // as implied by declaring copy constructor above
        }
    }
```

1. This template is a helper template used by the `slice` subscript operator

```cpp
gslice_array<T> valarray<T>::operator[](const gslice&);
```

2. It has reference semantics to a subset of an array specified by a `gslice` object.
3. Thus, the expression `a[gslice(1, length, stride)] = b` has the effect of assigning the elements of `b` to a generalized slice of the elements in `a`.

#### 26.6.7.2 `gslice_array` assignment

```cpp
void operator=(const valarray<T>&) const;
const gslice_array& operator=(const gslice_array&) const;
```

1. These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which the `gslice_array` refers.
26.6.7.3 gslice_array

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 computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the valarray<T> object to which the gslice_array object refers.

26.6.7.4 gslice_array fill function

void operator=(const T&) const;

1 This function has reference semantics, assigning the value of its argument to the elements of the valarray<T> object to which the gslice_array object refers.

26.6.8 Class template mask_array

26.6.8.1 Class template mask_array overview

namespace std {
    template <class T> class mask_array {
    public:
        typedef T value_type;

        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
    };
}

1 This template is a helper template used by the mask subscript operator:

mask_array<T> valarray<T>::operator[] (const valarray<bool>&).

§ 26.6.8.1
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 \textit{true}).

26.6.8.2 \textbf{mask_array assignment} \[\text{mask.array.assign}\]

```cpp
void operator=(const valarray<T>&) const;
const mask_array& operator=(const mask_array&) const;
```

These assignment operators have reference semantics, assigning the values of the argument array
elements to selected elements of the \texttt{valarray<T>} object to which it refers.

26.6.8.3 \textbf{mask_array computed assignment} \[\text{mask.array.comp.assign}\]

```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;
```

These computed assignments have reference semantics, applying the indicated operation to the elements
of the argument array and selected elements of the \texttt{valarray<T>} object to which the mask object refers.

26.6.8.4 \textbf{mask_array fill function} \[\text{mask.array.fill}\]

```cpp
void operator=(const T&) const;
```

This function has reference semantics, assigning the value of its argument to the elements of the
\texttt{valarray<T>} object to which the \texttt{mask_array} object refers.

26.6.9 \textbf{Class template indirect_array} \[\text{template.indirect.array}\]

26.6.9.1 \textbf{Class template indirect_array overview} \[\text{template.indirect.array.overview}\]

```cpp
namespace std {
    template <class T> class indirect_array {
    public:
        typedef T value_type;

        void operator=(const valarray<T>&) const;
        void operator**= (const valarray<T>&) const;
        void operator/= (const valarray<T>&) const;
        void operator%= (const valarray<T>&) const;
        void operator+= (const valarray<T>&) const;
        void operator-= (const valarray<T>&) const;
        void operator^= (const valarray<T>&) const;
        void operator&= (const valarray<T>&) const;
        void operator|= (const valarray<T>&) const;
        void operator<<=(const valarray<T>&) const;
        void operator>>=(const valarray<T>&) const;

        indirect_array(const indirect_array&);
        ~indirect_array();
        const indirect_array& operator=(const indirect_array&) const;
    }

§ 26.6.9.1
void operator=(const T&) const;

indirect_array() = delete;  // as implied by declaring copy constructor above
};

1 This template is a helper template used by the indirect subscript operator

indirect_array<T> valarray<T>::operator[](const valarray<size_t>&).

2 It has reference semantics to a subset of an array specified by an indirect_array. Thus the expression

a[indirect] = b;

has the effect of assigning the elements of b to the elements in a whose indices appear in indirect.

26.6.9.2 indirect_array assignment [indirect.array.assign]

void operator=(const valarray<T>&) const;
const indirect_array& operator=(const indirect_array&) const;

1 These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the valarray<T> object to which it refers.

2 If the indirect_array specifies an element in the valarray<T> object to which it refers more than once, the behavior is undefined.

3 [Example:

    int addr[] = {2, 3, 1, 4, 4};
    valarray<size_t> indirect(addr, 5);
    valarray<double> a(0., 10), b(1., 5);
    a[indirect] = b;

    results in undefined behavior since element 4 is specified twice in the indirection. — end example]

26.6.9.3 indirect_array computed assignment [indirect.array.comp.assign]

void operator*=(const valarray<T>&) const;
void operator/=(const valarray<T>&) const;
void operator%=(const valarray<T>&) const;
void operator^=(const valarray<T>&) const;
void operator&=(const valarray<T>&) const;
void operator|=(const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the valarray<T> object to which the indirect_array object refers.

2 If the indirect_array specifies an element in the valarray<T> object to which it refers more than once, the behavior is undefined.

26.6.9.4 indirect_array fill function [indirect.array.fill]

void operator=(const T&) const;

1 This function has reference semantics, assigning the value of its argument to the elements of the valarray<T> object to which the indirect_array object refers.
26.6.10 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 (24.2.7) and of a contiguous iterator (24.2.1) 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 (24.2.7) and of a contiguous iterator (24.2.1) 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)` (26.6.2.8) is called for that array or until the lifetime of that array ends, whichever happens first.

```cpp
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 numeric array.
```

```cpp
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 numeric array.
```

26.7 Generalized numeric operations

26.7.1 Header `<numeric>` synopsis

```cpp
namespace std {
    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);

    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);

    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);

    template <class InputIterator, class OutputIterator>
    OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                         OutputIterator result);
```

§ 26.7.1
template <class InputIterator, class OutputIterator,
class BinaryOperation>
OutputIterator adjacent_difference(InputIterator first,
InputIterator last,
OutputIterator result,
BinaryOperation binary_op);

template <class ForwardIterator, class T>
void iota(ForwardIterator first, ForwardIterator last, T value);
}

The requirements on the types of algorithms’ arguments that are described in the introduction to Clause 25 also apply to the following algorithms.

### 26.7.2 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);

**Effects:** Computes its result by initializing the accumulator acc with the initial value init and then modifies it with acc = acc + *i or acc = binary_op(acc, *i) for every iterator i in the range [first,last) in order.\(^{289}\)

**Requires:** T shall meet the requirements of CopyConstructible (Table 21) and CopyAssignable (Table 23) types. In the range [first,last], binary_op shall neither modify elements nor invalidate iterators or subranges.\(^{290}\)

### 26.7.3 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);

**Effects:** Computes its result by initializing the accumulator acc with the initial value init and then modifying it with acc = acc + (*i1) * (*i2) or acc = binary_op1(acc, binary_op2(*i1, *i2)) for every iterator i1 in the range [first1,last1) and iterator i2 in the range [first2,first2 + (last1 - first1)) in order.

**Requires:** T shall meet the requirements of CopyConstructible (Table 21) and CopyAssignable (Table 23) types. In the ranges [first1,last1) and [first2,first2 + (last1 - first1)] binary_op1 and binary_op2 shall neither modify elements nor invalidate iterators or subranges.\(^{291}\)

### 26.7.4 Partial sum

\(^{289}\) 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.

\(^{290}\) The use of fully closed ranges is intentional

\(^{291}\) The use of fully closed ranges is intentional
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);

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 = acc + *i or acc = binary_op(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.

Requires: InputIterator's value type shall be constructible from the type of *first. The result of the expression acc + *i or binary_op(acc, *i) shall be implicitly convertible to InputIterator's value type. acc shall be writable 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.\(^{292}\)

Remarks: result may be equal to first.

26.7.5 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);

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, creates an object val whose type is InputIterator's value type, initializes it with *i, computes val - acc or binary_op(val, acc), assigns the result to *(result + (i - first)), and move assigns from val to acc.

Requires: InputIterator's value type shall be MoveAssignable (Table 22) and shall be constructible from the type of *first. acc shall be writable to the result output iterator. The result of the expression val - acc or binary_op(val, acc) shall be writable 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.\(^{293}\)

Remarks: result may be equal to first.

Returns: result + (last - first).

Complexity: Exactly (last - first) - 1 applications of the binary operation.

\(^{292}\) The use of fully closed ranges is intentional.

\(^{293}\) The use of fully closed ranges is intentional.
26.7.6 Iota

template <class ForwardIterator, class T>
void iota(ForwardIterator first, ForwardIterator last, T value);

1 Requires: T shall be convertible to ForwardIterator’s value type. The expression ++val, where val has type T, shall be well formed.

2 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.

3 Complexity: Exactly last - first increments and assignments.

26.8 C library

1 The header <ctgmath> simply includes the headers <ccomplex> and <cmath>.

2 [Note: The overloads provided in C by type-generic macros are already provided in <ccomplex> and <cmath> by “sufficient” additional overloads. —end note]

3 Tables 119 and 120 describe headers <cmath> and <cstdlib>, respectively.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td></td>
</tr>
<tr>
<td>FP_FAST_FMA</td>
<td>FP_ILOGBNAN</td>
</tr>
<tr>
<td>FP_FAST_FMAF</td>
<td>FP_INFINITE</td>
</tr>
<tr>
<td>FP_FAST_FMAL</td>
<td>FP_NORMAL</td>
</tr>
<tr>
<td>FP_ILOGBO</td>
<td></td>
</tr>
<tr>
<td>Types:</td>
<td></td>
</tr>
<tr>
<td>double_t</td>
<td>float_t</td>
</tr>
<tr>
<td>Math Functions:</td>
<td></td>
</tr>
<tr>
<td>abs</td>
<td>cosh</td>
</tr>
<tr>
<td>acos</td>
<td>erf</td>
</tr>
<tr>
<td>acosh</td>
<td>erfc</td>
</tr>
<tr>
<td>asin</td>
<td>exp2</td>
</tr>
<tr>
<td>asinh</td>
<td>exp</td>
</tr>
<tr>
<td>atan</td>
<td>expm1</td>
</tr>
<tr>
<td>atan2</td>
<td>fabs</td>
</tr>
<tr>
<td>atanh</td>
<td>fdim</td>
</tr>
<tr>
<td>cbrt</td>
<td>floor</td>
</tr>
<tr>
<td>ceil</td>
<td>fma</td>
</tr>
<tr>
<td>copy sign</td>
<td>fmax</td>
</tr>
<tr>
<td>cos</td>
<td>fmin</td>
</tr>
<tr>
<td>Classification/comparison Functions:</td>
<td></td>
</tr>
<tr>
<td>fpclassify</td>
<td>isgreater equal</td>
</tr>
<tr>
<td>isfinite</td>
<td>isinf</td>
</tr>
<tr>
<td>isgreater</td>
<td>isless</td>
</tr>
</tbody>
</table>

4 The contents of these headers are the same as the Standard C library headers <math.h> and <stdlib.h> respectively, with the following changes:

5 The rand function has the semantics specified in the C standard, except that the implementation may specify that particular library functions may call rand. It is implementation-defined whether the rand function may introduce data races (17.6.5.9). [Note: The random number generation (26.5) facilities in this standard are
Table 120 — Header `<cstdlib>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro</td>
<td>RAND_MAX</td>
</tr>
<tr>
<td>Types</td>
<td></td>
</tr>
<tr>
<td>div_t</td>
<td>ldiv_t</td>
</tr>
<tr>
<td>lldiv_t</td>
<td></td>
</tr>
<tr>
<td>Functions</td>
<td></td>
</tr>
<tr>
<td>abs</td>
<td>ldiv, rand</td>
</tr>
<tr>
<td>div</td>
<td>llabs, srand</td>
</tr>
<tr>
<td>labs</td>
<td>lldiv</td>
</tr>
</tbody>
</table>

often preferable to `rand`, because `rand`'s underlying algorithm is unspecified. Use of `rand` therefore continues to be nonportable, with unpredictable and oft-questionable quality and performance. —end note]

6 In addition to the `int` versions of certain math functions in `<cstdlib>`, C++ adds `long` and `long long` overloaded versions of these functions, with the same semantics.

7 The added signatures are:

```c
long abs(long); // labs()
long long abs(long long); // llabs()
ldiv_t div(long, long); // ldiv()
lldiv_t div(long long, long long); // lldiv()
```

8 In addition to the `double` versions of the math functions in `<cmath>`, C++ adds `float` and `long double` overloaded versions of these functions, with the same semantics.

9 The added signatures are:

```c
float abs(float);
float acos(float);
float acosh(float);
float asin(float);
float asinh(float);
float atan(float);
float atan2(float, float);
float atanh(float);
float cbrt(float);
float ceil(float);
float copysign(float, float);
float cos(float);
float cosh(float);
float erf(float);
float erfc(float);
float exp(float);
float exp2(float);
float expm1(float);
float fabs(float);
float fdim(float, float);
float floor(float);
float fma(float, float, float);
float fmax(float, float);
float fmin(float, float);
float fmod(float, float);
float frexp(float, int*);
float hypot(float, float);
```
int ilogb(float);
float ldexp(float, int);
float lgamma(float);
long long llrint(float);
long long llround(float);
float log(float);
float log10(float);
float log1p(float);
float log2(float);
float logb(float);
long lrint(float);
long lround(float);
float modf(float, float*);
float nearbyint(float);
float nextafter(float, float);
float nexttoward(float, long double);
float pow(float, float);
float remainder(float, float);
float remquo(float, float, int *);
float rint(float);
float round(float);
float scalbln(float, long);
float scalbn(float, int);
float sin(float);
float sinh(float);
float sqrt(float);
float tan(float);
float tanh(float);
float tgamma(float);
float trunc(float);

double abs(double); // fabs()

long double abs(long double);
long double acos(long double);
long double acosh(long double);
long double asin(long double);
long double asinh(long double);
long double atan(long double);
long double atan2(long double, long double);
long double atanh(long double);
long double cbrt(long double);
long double ceil(long double);
long double copysign(long double, long double);
long double cos(long double);
long double cosh(long double);
long double erf(long double);
long double erfc(long double);
long double exp(long double);
long double exp2(long double);
long double expm1(long double);
long double fabs(long double);
long double fdim(long double, long double);
long double floor(long double);
long double fma(long double, long double, long double);
long double fmax(long double, long double);
long double fmin(long double, long double);
long double fmod(long double, long double);
long double frexp(long double, int*);
long double hypot(long double, long double);
int ilogb(long double);
long double ldexp(long double, int);
long double lgamma(long double);
long double llrint(long double);
long double llround(long double);
long double log(long double);
long double log10(long double);
long double log1p(long double);
long double log2(long double);
long double logb(long double);
long double lrint(long double);
long double lround(long double);
long double modf(long double, long double*);
long double nearbyint(long double);
long double nextafter(long double, long double);
long double nexttoward(long double, long double);
long double pow(long double, long double);
long double remainder(long double, long double);
long double remquo(long double, long double, int *);
long double round(long double);
long double scalbln(long double, long);
long double scalbn(long double, int);
long double sin(long double);
long double sinh(long double);
long double sqrt(long double);
long double tan(long double);
long double tanh(long double);
long double tgamma(long double);
long double trunc(long double);

10 The classification/comparison functions behave the same as the C macros with the corresponding names
declared in 7.12.3, Classification macros, and 7.12.14, Comparison macros in the C Standard. Each function
is overloaded for the three floating-point types, as follows:

int fpclassify(float x);
bool isfinite(float x);
bool isinf(float x);
bool isnan(float x);
bool isnormal(float x);
bool signbit(float x);
bool isgreater(float x, float y);
bool isgreaterequal(float x, float y);
bool isless(float x, float y);
bool islessequal(float x, float y);
bool islessgreater(float x, float y);
bool isunordered(float x, float y);

int fpclassify(double x);
bool isfinite(double x);

§ 26.8
bool isinf(double x);
bool isnan(double x);
bool isnormal(double x);
bool signbit(double x);

bool isgreater(double x, double y);
bool isgreaterequal(double x, double y);
bool isless(double x, double y);
bool islessequal(double x, double y);
bool islessgreater(double x, double y);
bool isunordered(double x, double y);

int fpclassify(long double x);
bool isfinite(long double x);
bool isinf(long double x);
bool isnan(long double x);
bool isnormal(long double x);
bool signbit(long double x);

bool isgreater(long double x, long double y);
bool isgreaterequal(long double x, long double y);
bool isless(long double x, long double y);
bool islessequal(long double x, long double y);
bool islessgreater(long double x, long double y);
bool isunordered(long double x, long double y);

Moreover, there shall be additional overloads sufficient to ensure:

1. If any arithmetic argument corresponding to a `double` parameter has type `long double`, then all arithmetic arguments corresponding to `double` parameters are effectively cast to `long double`.

2. Otherwise, if any arithmetic argument corresponding to a `double` parameter has type `double` or an integer type, then all arithmetic arguments corresponding to `double` parameters are effectively cast to `double`.

3. Otherwise, all arithmetic arguments corresponding to `double` parameters have type `float`.

See also: ISO C 7.5, 7.10.2, 7.10.6.
27 Input/output library [input.output]

27.1 General [input.output.general]

This Clause describes components that C++ programs may use to perform input/output operations.

The following subclauses describe requirements for stream parameters, and components for forward declarations of iostreams, predefined iostreams objects, base iostreams classes, stream buffering, stream formatting and manipulators, string streams, and file streams, as summarized in Table 121.

Table 121 — Input/output library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.2 Requirements</td>
<td></td>
</tr>
<tr>
<td>27.3 Forward declarations</td>
<td>&lt;iosfwd&gt;</td>
</tr>
<tr>
<td>27.4 Standard iostream objects</td>
<td>&lt;iostream&gt;</td>
</tr>
<tr>
<td>27.5 Iostreams base classes</td>
<td>&lt;ios&gt;</td>
</tr>
<tr>
<td>27.6 Stream buffers</td>
<td>&lt;streambuf&gt;</td>
</tr>
<tr>
<td>27.7 Formatting and manipulators</td>
<td>&lt;istream&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;ostream&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;iomanip&gt;</td>
</tr>
<tr>
<td>27.8 String streams</td>
<td>&lt;sstream&gt;</td>
</tr>
<tr>
<td>27.9 File streams</td>
<td>&lt;fstream&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cstdio&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cinttypes&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.

Figure 7 — Stream position, offset, and size types [non-normative]
27.2 Iostreams requirements

27.2.1 Imbue limitations

No function described in Clause 27 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 27 or as an overriding virtual function of any class declared in Clause 27 calls `imbue`, the behavior is undefined.

27.2.2 Positioning type limitations

The classes of Clause 27 with template arguments `charT` and `traits` behave as described if `traits::pos_type` and `traits::off_type` are `streampos` and `streamoff` respectively. Except as noted explicitly below, their behavior when `traits::pos_type` and `traits::off_type` are other types is implementation-defined.

In the classes of Clause 27, 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.

27.2.3 Thread safety

Concurrent access to a stream object (27.8, 27.9), stream buffer object (27.6), or C Library stream (27.9.2) by multiple threads may result in a data race (1.10) unless otherwise specified (27.4). [Note: Data races result in undefined behavior (1.10). — end note]

If one thread makes a library call `a` that writes a value to a stream and, as a result, another thread reads this value from the stream through a library call `b` such that this does not result in a data race, then `a`'s write synchronizes with `b`'s read.

27.3 Forward declarations

Header `<iosfwd>` synopsis

```cpp
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 Allocator = allocator<charT> >
    class basic_iostream;

    template <class charT, class traits = char_traits<charT>,
              class Allocator = allocator<charT> >
    class basic_stringbuf;
    template <class charT, class traits = char_traits<charT>,
              class Allocator = allocator<charT> >
    class basic_istringstream;
}
```
template <class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_ostringstream;

template <class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_stringstream;

template <class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_filebuf;

template <class charT, class traits = char_traits<charT> >
class 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 istreambuf_iterator;

template <class charT, class traits = char_traits<charT> >
class ostreambuf_iterator;

typedef basic_ios<char> ios;

typedef basic_ios<wchar_t> wios;

typedef basic_streambuf<char> streambuf;

typedef basic_istream<char> istream;

typedef basic_ostringstream<char> ostringstream;

typedef basic_filebuf<char> filebuf;

typedef basic_ifstream<char> ifstream;

typedef basic_ofstream<char> ofstream;

typedef basic_fstream<char> fstream;

typedef basic_streambuf<wchar_t> wstreambuf;

typedef basic_istream<wchar_t> wistream;

typedef basic_ostringstream<wchar_t> wostringstream;

typedef basic_filebuf<wchar_t> wfilebuf;

typedef basic_ifstream<wchar_t> wifstream;

typedef basic_ofstream<wchar_t> wofstream;

typedef basic_fstream<wchar_t> wfstream;

typedef basic_streambuf<wchar_t> wstringbuf;

typedef basic_istream<wchar_t> wistream;

typedef basic_ostringstream<wchar_t> wostream;

typedef basic_filebuf<wchar_t> wfilebuf;

typedef basic_ifstream<wchar_t> wifstream;

typedef basic_ofstream<wchar_t> wofstream;

typedef basic_fstream<wchar_t> wfstream;

typedef fpos<char_traits<char>::state_type> streampos;
typedef fpos<char_traits<wchar_t>::state_type> wstreampos;

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.294

2 [Note: The class template specialization `basic_ios<charT,traits>` serves as a virtual base class for the
class templates `basic_istream`, `basic_ostream`, and class templates derived from them. `basic_iostream`
is a class template derived from both `basic_istream<charT,traits>` and `basic_ostream<charT,traits>`.]

3 The class template specialization `basic_streambuf<charT,traits>` serves as a base class for class templates
`basic_stringbuf` and `basic_filebuf`.

4 The class template specialization `basic_istream<charT,traits>` serves as a base class for class templates
`basic_istreamstream` and `basic_ifstream`.

5 The class template specialization `basic_ostream<charT,traits>` serves as a base class for class templates
`basic_ostringstream` and `basic_ofstream`.

6 The class template specialization `basic_iostream<charT,traits>` serves as a base class for class templates
`basic_stringstream` and `basic_fstream`.

7 Other typedefs define instances of class templates specialized for `char` or `wchar_t` types.

8 Specializations of the class template `fpos` are used for specifying file position information.

9 The types `streampos` and `wstreampos` are used for positioning streams specialized on `char` and `wchar_t`
respectively.

10 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

```
template<class stateT> class fpos {
  ...;
}  // depends on nothing
typedef ... _STATE;  // implementation private declaration of stateT

typedef fpos<_STATE> streampos;

template<> struct char_traits<char> {
  typedef streampos
  pos_type;
}
```

—end note]}

27.4 Standard iostream objects

27.4.1 Overview

Header `<iostream>` synopsis

```
#include <ios>
#include <streambuf>
#include <istream>
#include <ostream>

namespace std {
  extern istream cin;
}
```

294) 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.
extern ostream cout;
extern ostream cerr;
extern ostream clog;

extern wistream wcin;
extern wostream wcout;
extern wostream wcerr;
extern wostream wclog;
}

1 The header `<iostream>` declares objects that associate objects with the standard C streams provided for by the functions declared in `<cstdio>` (27.9.2), and includes all the headers necessary to use these objects.

2 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` begins execution.295 The objects are not destroyed during program execution.296 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. Similarly, the entire program shall behave as if there were at least one instance of `ios_base::Init` with static storage duration.

3 Mixing operations on corresponding wide- and narrow-character streams follows the same semantics as mixing such operations on `FILE`s, as specified in Amendment 1 of the ISO C standard.

4 Concurrent access to a synchronized (27.5.3.4) standard iostream object’s formatted and unformatted input (27.7.2.1) and output (27.7.3.1) functions or a standard C stream by multiple threads shall not result in a data race (1.10). [Note: Users must still synchronize concurrent use of these objects and streams by multiple threads if they wish to avoid interleaved characters. —end note]

#### 27.4.2 Narrow stream objects

**istream cin;**

1 The object `cin` controls input from a stream buffer associated with the object `stdin`, declared in `<cstdio>`.

2 After the object `cin` is initialized, `cin.tie()` returns `&cout`. Its state is otherwise the same as required for `basic_ios<char>::init` (27.5.5.2).

**ostream cout;**

3 The object `cout` controls output to a stream buffer associated with the object `stdout`, declared in `<cstdio>` (27.9.2).

**ostream cerr;**

4 The object `cerr` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (27.9.2).

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` (27.5.5.2).

**ostream clog;**

6 The object `clog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (27.9.2).

---

295) If it is possible for them to do so, implementations are encouraged to initialize the objects earlier than required.
296) Constructors and destructors for static objects can access these objects to read input from `stdin` or write output to `stdout` or `stderr`. 

§ 27.4.2
27.4.3 Wide stream objects

wistream wcin;

1 The object wcin controls input from a stream buffer associated with the object stdin, declared in <cstdio>.

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 (27.5.5.2).

wostream wcout;

3 The object wcout controls output to a stream buffer associated with the object stdout, declared in <cstdio> (27.9.2).

wostream wcerr;

4 The object wcerr controls output to a stream buffer associated with the object stderr, declared in <cstdio> (27.9.2).

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 (27.5.5.2).

wostream wclog;

6 The object wclog controls output to a stream buffer associated with the object stderr, declared in <cstdio> (27.9.2).

27.5 Iostreams base classes

27.5.1 Overview

Header <ios> synopsis

#include <iosfwd>

namespace std {
     typedef implementation-defined streamoff;
     typedef implementation-defined streamsize;
     template <class stateT> class fpos;
     class ios_base;
     template <class charT, class traits = char_traits<charT> >
     class basic_ios;

     // 27.5.6, 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);

§ 27.5.1
ios_base& uppercase (ios_base& str);
ios_base& nouppercase(ios_base& str);

ios_base& unitbuf (ios_base& str);
ios_base& nounitbuf (ios_base& str);

// 27.5.6.2 adjustfield:
ios_base& internal (ios_base& str);
ios_base& left (ios_base& str);
ios_base& right (ios_base& str);

// 27.5.6.3 basefield:
ios_base& dec (ios_base& str);
ios_base& hex (ios_base& str);
ios_base& oct (ios_base& str);

// 27.5.6.4 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);

// 27.5.6.5 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;
}

27.5.2 Types [stream.types]

typedef implementation-defined streamoff;

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.²⁹⁷

typedef implementation-defined streamsize;

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.²⁹⁸

27.5.3 Class ios_base [ios.base]

namespace std {
  class ios_base {
    public:
      class failure;

²⁹⁷ Typically long long.
²⁹⁸ 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 strstreambuf constructors, which require negative values. It should probably be the signed type corresponding
to size_t (which is what Posix.2 calls ssize_t).
// 27.5.3.1.2 fmtflags
typedef T1 fmtflags;
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;

// 27.5.3.1.3 iostate
typedef T2 iostate;
static constexpr iostate badbit = unspecified;
static constexpr iostate eofbit = unspecified;
static constexpr iostate failbit = unspecified;
static constexpr iostate goodbit = see below;

// 27.5.3.1.4 openmode
typedef T3 openmode;
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;

// 27.5.3.1.5 seekdir
typedef T4 seekdir;
static constexpr seekdir beg = unspecified;
static constexpr seekdir cur = unspecified;
static constexpr seekdir end = unspecified;

class Init;

// 27.5.3.2 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);

// 27.5.3.3 locales:
locale imbue(const locale& loc);
locale getloc() const;

// 27.5.3.5 storage:
static int xalloc();
long& iword(int index);
void*& pword(int index);

// destructor
virtual ~ios_base();

// 27.5.3.6 callbacks;
enum event { erase_event, imbue_event, copyfmt_event };
typedef void (*event_callback)(event, ios_base&, int index);
void register_callback(event_callback fn, int index);

ios_base(const ios_base&) = delete;
ios_base& operator=(const ios_base&) = delete;

static bool sync_with_stdio(bool sync = true);

protected:
  ios_base();

private:
  static int index; // exposition only
  long* iarray; // exposition only
  void** parray; // exposition only
};

1 ios_base defines several member types:
(1.1) — a class failure derived from system_error;
(1.2) — a class Init;
(1.3) — three bitmask types, fmtflags, iostate, and openmode;
(1.4) — an enumerated type, seekdir.

2 It maintains several kinds of data:
(2.1) — state information that reflects the integrity of the stream buffer;
(2.2) — control information that influences how to interpret (format) input sequences and how to generate (format) output sequences;
(2.3) — additional information that is stored by the program for its private use.

3 [Note: For the sake of exposition, the maintained data is presented here as:
(3.1) static int index, specifies the next available unique index for the integer or pointer arrays maintained for the private use of the program, initialized to an unspecified value;

(3.2) long* iarray, points to the first element of an arbitrary-length long array maintained for the private use of the program;

(3.3) void** parray, points to the first element of an arbitrary-length pointer array maintained for the private use of the program. — end note |

27.5.3.1 Types  

27.5.3.1.1 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 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, iostrm_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.

27.5.3.1.2 Type ios_base::fmtflags

typedef T1 fmtflags;

1 The type fmtflags is a bitmask type (17.5.2.1.3). Setting its elements has the effects indicated in Table 122.

2 Type fmtflags also defines the constants indicated in Table 123.

27.5.3.1.3 Type ios_base::iostate

typedef T2 iostate;

1 The type iostate is a bitmask type (17.5.2.1.3) that contains the elements indicated in Table 124.

2 Type iostate also defines the constant:

(2.1) — goodbit, the value zero.

27.5.3.1.4 Type ios_base::openmode

typedef T3 openmode;

1 The type openmode is a bitmask type (17.5.2.1.3). It contains the elements indicated in Table 125.
Table 122 — `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 <code>right</code> if no such point is designated</td>
</tr>
<tr>
<td>left</td>
<td>adds fill characters on the right (final positions) of certain generated output</td>
</tr>
<tr>
<td>oct</td>
<td>converts integer input or generates integer output in octal base</td>
</tr>
<tr>
<td>right</td>
<td>adds fill characters on the left (initial positions) of certain generated output</td>
</tr>
<tr>
<td>scientific</td>
<td>generates floating-point output in scientific notation</td>
</tr>
<tr>
<td>showbase</td>
<td>generates a prefix indicating the numeric base of generated integer output</td>
</tr>
<tr>
<td>showpoint</td>
<td>generates a decimal-point character unconditionally in generated floating-point output</td>
</tr>
<tr>
<td>showpos</td>
<td>generates a + sign in non-negative generated numeric output</td>
</tr>
<tr>
<td>skipws</td>
<td>skips 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 123 — `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 124 — `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 125 — `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>
27.5.3.1.5 Type `ios_base::seekdir`  

```cpp
typedef T4 seekdir;
```

The type `seekdir` is an enumerated type (17.5.2.1.2) that contains the elements indicated in Table 126.

<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>

27.5.3.1.6 Class `ios_base::Init`  

```cpp
namespace std {
    class ios_base::Init {
    public:
        Init();
        ~Init();
    private:
        static int init_cnt; // exposition only
    }
}
```

1 The class `Init` describes an object whose construction ensures the construction of the eight objects declared in `<iostream>` (27.4) that associate file stream buffers with the standard C streams provided for by the functions declared in `<cstdio>` (27.9.2).

2 For the sake of exposition, the maintained data is presented here as:

(2.1) — `static int init_cnt`, counts the number of constructor and destructor calls for class `Init`, initialized to zero.

```cpp
Init();
```

3 **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.

```cpp
~Init();
```

4 **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()`, and `wclog.flush()`.

27.5.3.2 `ios_base` state functions  

```cpp
fmtflags flags() const;
```

1 **Returns:** The format control information for both input and output.

```cpp
fmtflags flags(fmtflags fmtfl);
```

2 **Postcondition:** `fmtfl == flags()`.

3 **Returns:** The previous value of `flags()`.

```cpp
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);
```

Postcondition: 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);
```

Postcondition: wide == width().

Returns: The previous value of width().

### 27.5.3.3 ios_base functions

locale imbue(const locale& loc);

Effects: Calls each registered callback pair (fn, index) at such a time that a call to \texttt{ios\_base::getloc()} from within \texttt{fn} returns the new locale value \texttt{loc}.

Returns: The previous value of \texttt{getloc()}.  

Postcondition: loc == getloc().

locale getloc() const;

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.

### 27.5.3.4 ios_base static members

bool sync_with_stdio(bool sync = true);

Returns: \texttt{true} if the previous state of the standard iostream objects (27.4) was synchronized and otherwise returns \texttt{false}. The first time it is called, the function returns \texttt{true}.

Effects: If any input or output operation has occurred using the standard streams prior to the call, the effect is implementation-defined. Otherwise, called with a false argument, it allows the standard streams to operate independently of the standard C streams.

When a standard iostream object \texttt{str} is \texttt{synchronized} with a standard stdio stream \texttt{f}, the effect of inserting a character \texttt{c} by

§ 27.5.3.4
fputc(f, c);
  
is the same as the effect of
  
  str.rdbuf()->sputc(c);

for any sequences of characters; the effect of extracting a character c by
  
  c = fgetc(f);

is the same as the effect of
  
  c = str.rdbuf()->sbumpc();

for any sequences of characters; and the effect of pushing back a character c by
  
  ungetc(c, f);

is the same as the effect of
  
  str.rdbuf()->sputbackc(c);

for any sequence of characters.299

27.5.3.5 ios_base storage functions

static int xalloc();

Returns: index ++.

Remarks: Concurrent access to this function by multiple threads shall not result in a data race (1.10).

long& iword(int idx);

Effects: If iarray is a null pointer, allocates an array of long of unspecified size and stores a pointer to its first element in iarray. The function then extends the array pointed at by iarray as necessary to include the element iarray[idx]. Each newly allocated element of the array is initialized to zero. The reference returned is invalid after any other operations on the object.300 However, the value of the storage referred to is retained, so that until the next call to copyfmt, calling iword with the same index yields another reference to the same value. If the function fails and *this is a base 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.

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 parray[idx]. Each newly allocated element of the array is initialized to a null pointer. The reference returned is invalid after any other operations on the object. However, the value of the storage referred to is retained, so that until the next call to copyfmt, calling pword with the same index yields another reference to the same value. If the function fails,302 for example, because it cannot allocate space.

299) 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.

300) 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.

301) for example, because it cannot allocate space.

302) for example, because it cannot allocate space.

§ 27.5.3.5
and *this is a base 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.

### 27.5.3.6 ios_base callbacks

```
void register_callback(event_callback fn, int index);
```

**Effects:** Registers the pair (fn, index) such that during calls to imbue() (27.5.3.3), copyfmt(), or ~ios_base() (27.5.3.7), 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.

### 27.5.3.7 ios_base constructors/destructor

```
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.

```~ios_base();
```

**Effects:** Destroys an object of class ios_base. Calls each registered callback pair (fn, index) (27.5.3.6) as (*fn)(erase_event, *this, index) at such time that any ios_base member function called from within fn has well defined results.

### 27.5.4 Class template fpos

```
namespace std {
    template <class stateT> class fpos {
public:
    // 27.5.4.1 Members
    stateT state() const;
    void state(stateT);
private:
    stateT st; // exposition only
    
    }
}
```

#### 27.5.4.1 fpos members

```void state(stateT s);
```

**Effects:** Assign s to st.

```stateT state() const;
```

**Returns:** Current value of st.
27.5.4.2 fpos requirements [fpos.operations]

Operations specified in Table 127 are permitted. In that table,

1. (1.1) \( P \) refers to an instance of fpos,
2. (1.2) \( p \) and \( q \) refer to values of type \( P \),
3. (1.3) \( O \) refers to type streamoff,
4. (1.4) \( o \) refers to a value of type streamoff,
5. (1.5) \( sz \) refers to a value of type streamsize and
6. (1.6) \( i \) refers to a value of type int.

Table 127 — Position type requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P(i) )</td>
<td>( P )</td>
<td>( p == P(i) )</td>
<td>post: ( p == P(i) ).</td>
</tr>
<tr>
<td>( P \ p(i); ) ( P \ p = i; )</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>( P(0(p)) == p )</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 ) ( p += o )</td>
<td>fpos</td>
<td>+ offset</td>
<td>( q - o == p )</td>
</tr>
<tr>
<td>( q = p - o ) ( p -= o )</td>
<td>fpos</td>
<td>- offset</td>
<td>( q + o == p )</td>
</tr>
<tr>
<td>( o = p - q )</td>
<td>streamoff</td>
<td>distance</td>
<td>( q + o == p )</td>
</tr>
<tr>
<td>( streamsize(o) )</td>
<td>streamsize</td>
<td>converts</td>
<td>( streamsize(0(sz)) == sz )</td>
</tr>
<tr>
<td>( 0(sz) )</td>
<td>streamoff</td>
<td>converts</td>
<td>( streamsize(0(sz)) == sz )</td>
</tr>
</tbody>
</table>

2 [Note: Every implementation is required to supply overloaded operators on fpos objects to satisfy the requirements of 27.5.4.2. 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 \( P(0(-1)) \) as an invalid value to signal an error. If this value is used as an argument to any istream, ostream, or streambuf member that accepts a value of type traits::pos_type then the behavior of that function is undefined.

27.5.5 Class template basic_ios [ios]

27.5.5.1 Overview [ios.overview]

namespace std {

    template <class charT, class traits = char_traits<charT> >
    class basic_ios : public ios_base {
    public:

        // types:

    }

§ 27.5.5.1
typedef charT char_type;
typedef typename traits::int_type int_type;
typedef typename traits::pos_type pos_type;
typedef typename traits::off_type off_type;
typedef traits traits_type;

explicit operator bool() const;
bool operator!() const;
iosate rdstate() const;
void clear(iostate state = goodbit);
void setstate(iostate state);
bool good() const;
bool eof() const;
bool fail() const;
bool bad() const;

iosate exceptions() const;
void exceptions(iostate except);

// 27.5.5.2 Constructor/destructor:
explicit basic_ios(basic_streambuf<charT,traits>* sb);
virtual ~basic_ios();

// 27.5.5.3 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);

};

}
explicit basic_ios(basic_streambuf<charT,traits>* sb);

Effects: Constructs an object of class basic_ios, assigning initial values to its member objects by calling init(sb).

basic_ios();

Effects: Constructs an object of class basic_ios (27.5.3.7) 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();

Remarks: The destructor does not destroy rdbuf().

void init(basic_streambuf<charT,traits>* sb);

Postconditions: The postconditions of this function are indicated in Table 128.

Table 128 — 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>

27.5.5.3 Member functions [basic.ios.members]

basic_ostream<charT,traits>* tie() const;

Returns: An output sequence that is tied to (synchronized with) the sequence controlled by the stream buffer.

basic_ostream<charT,traits>* tie(basic_ostream<charT,traits>* tiestr);

Requires: If tiestr is not null, tiestr must not be reachable by traversing the linked list of tied stream objects starting from tiestr->tie().

Postcondition: 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);
Postcondition: \( \text{sb} == \text{rdbuf()} \).

Effects: Calls \( \text{clear()} \).

Returns: The previous value of \( \text{rdbuf()} \).

locale imbue(const locale& loc);

Effects: Calls \( \text{ios\_base::imbue(loc)} \) (27.5.3.3) and if \( \text{rdbuf()}! = 0 \) then \( \text{rdbuf()}\rightarrow\text{pubimbue(loc)} \) (27.6.3.2.1).

Returns: The prior value of \( \text{ios\_base::imbue()} \).

char narrow(char_type c, char dfault) const;

Returns: \( \text{use\_facet< ctype<char\_type> >(getloc()).narrow(c,dfault)} \)

char_type widen(char c) const;

Returns: \( \text{use\_facet< ctype<char\_type> >(getloc()).widen(c)} \)

char_type fill() const;

Returns: The character used to pad (fill) an output conversion to the specified field width.

char_type fill(char_type fillch);

Postcondition: \( \text{traits::eq(fillch, fill())} \)

Returns: The previous value of \( \text{fill()} \).

basic_ios& copyfmt(const basic_ios& rhs);

Effects: If \( \text{this} == \&\text{rhs} \) does nothing. Otherwise assigns to the member objects of \( *\text{this} \) the corresponding member objects of \( \text{rhs} \) as follows:
1. calls each registered callback pair \( (\text{fn}, \text{index}) \) as \( (*\text{fn})(\text{erase\_event, *this, index}) \);
2. assigns to the member objects of \( *\text{this} \) the corresponding member objects of \( \text{rhs} \), except that
   (16.1) \( \text{rdstate()}, \text{rdbuf()}, \text{and exceptions()} \) are left unchanged;
   (16.2) the contents of arrays pointed at by \( \text{pword} \) and \( \text{iword} \) are copied, not the pointers themselves;\(^{303}\) and
   (16.3) if any newly stored pointer values in \( *\text{this} \) point at objects stored outside the object \( \text{rhs} \) and those objects are destroyed when \( \text{rhs} \) is destroyed, the newly stored pointer values are altered to point at newly constructed copies of the objects;
3. calls each callback pair that was copied from \( \text{rhs} \) as \( (*\text{fn})(\text{copyfmt\_event, *this, index}) \);
4. calls \( \text{exceptions(rhs.except())} \).

Note: The second pass through the callback pairs permits a copied \( \text{pword} \) value to be zeroed, or to have its referent deep copied or reference counted, or to have other special action taken.

Postconditions: The postconditions of this function are indicated in Table 129.

Returns: \( *\text{this} \).

void move(basic_ios& rhs);
void move(basic_ios&& rhs);

\(^{303}\) This suggests an infinite amount of copying, but the implementation can keep track of the maximum element of the arrays that is non-zero.

§ 27.5.5.3
Table 129 — 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>

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.

void swap(basic_ios& rhs) noexcept;

Effects: The states of *this and rhs shall be exchanged, except that rdbuf() shall return the same value as it returned before the function call, and rhs.rdbuf() shall return the same value as it returned before the function call.

void set_rdbuf(basic_streambuf<charT, traits>* sb);

Requires: sb != nullptr.

Effects: Associates the basic_streambuf object pointed to by sb with this stream without calling clear().

Postconditions: rdbuf() == sb.

Throws: Nothing.

27.5.5.4 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);

Postcondition: 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 fail of class basic_ios::failure (27.5.3.1.1), constructed with implementation-defined argument values.

void setstate(iostate state);

§ 27.5.5.4
Effects: Calls clear(rdstate() | state) (which may throw basic_ios::failure (27.5.3.1.1)).

bool good() const;

Returns: rdstate() == 0

bool eof() const;

Returns: true if eofbit is set in rdstate().

bool fail() const;

Returns: true if failbit or badbit is set in rdstate().

bool bad() const;

Returns: true if badbit is set in rdstate().

iostate exceptions() const;

Returns: A mask that determines what elements set in rdstate() cause exceptions to be thrown.

void exceptions(iostate except);

Postcondition: except == exceptions().

Effects: Calls clear(rdstate()).

27.5.6 ios_base manipulators

27.5.6.1 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.

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: Checking badbit also for fail() is historical practice.
11  Effects: Calls `str.unsetf(ios_base::showpoint)`.
12  Returns: `str`.

io_base& showpos(ios_base& str);
13  Effects: Calls `str.setf(ios_base::showpos)`.
14  Returns: `str`.

15  Effects: Calls `str.unsetf(ios_base::showpos)`.
16  Returns: `str`.

17  Effects: Calls `str.setf(ios_base::skipws)`.
18  Returns: `str`.

19  Effects: Calls `str.unsetf(ios_base::skipws)`.
20  Returns: `str`.

21  Effects: Calls `str.setf(ios_base::uppercase)`.
22  Returns: `str`.

23  Effects: Calls `str.unsetf(ios_base::uppercase)`.
24  Returns: `str`.

25  Effects: Calls `str.setf(ios_base::unitbuf)`.
26  Returns: `str`.

27  Effects: Calls `str.unsetf(ios_base::unitbuf)`.
28  Returns: `str`.

27.5.6.2 adjustfield manipulators

29  Effects: Calls `str.setf(ios_base::internal, ios_base::adjustfield)`.
30  Returns: `str`.

31  Effects: Calls `str.setf(ios_base::left, ios_base::adjustfield)`.
32  Returns: `str`.

33  Effects: Calls `str.setf(ios_base::right, ios_base::adjustfield)`.
34  Returns: `str`.
27.5.6.3 basefield manipulators [basefield.manip]

ios_base& dec(ios_base& str);
1  
Effects: Calls str.setf(ios_base::dec, ios_base::basefield).
2  
Returns: str
3

ios_base& hex(ios_base& str);
4  
Effects: Calls str.setf(ios_base::hex, ios_base::basefield).
5  
Returns: str.
6

ios_base& oct(ios_base& str);
7  
Effects: Calls str.setf(ios_base::oct, ios_base::basefield).
8  
Returns: str.
9

27.5.6.4 floatfield manipulators [floatfield.manip]

ios_base& fixed(ios_base& str);
1  
Effects: Calls str.setf(ios_base::fixed, ios_base::floatfield).
2  
Returns: str.
3

ios_base& scientific(ios_base& str);
4  
Effects: Calls str.setf(ios_base::scientific, ios_base::floatfield).
5  
Returns: str.
6

ios_base& hexfloat(ios_base& str);
7  
Effects: Calls str.setf(ios_base::fixed | ios_base::scientific, ios_base::floatfield).
8  
Returns: str.
9

[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);
10  
Effects: Calls str.unsetf(ios_base::floatfield).
11  
Returns: str.
12

27.5.6.5 Error reporting [error.reporting]

error_code make_error_code(io_errc e) noexcept;
1  
Returns: error_code(static_cast<int>(e), iostream_category()).
2

error_condition make_error_condition(io_errc e) noexcept;
3  
Returns: error_condition(static_cast<int>(e), iostream_category()).
4

cost error_category& iostream_category() noexcept;
5

305) The function signature dec(ios_base&) can be called by the function signature basic_ostream& stream::operator<<(ios_base& (*)(ios_base&)) to permit expressions of the form cout <<dec to change the format flags stored in cout.
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".

27.6 Stream buffers

27.6.1 Overview

Header <streambuf> synopsis

namespace std {
    template <class charT, class traits = char_traits<charT> >
    class basic_streambuf;
    typedef basic_streambuf<char> streambuf;
    typedef basic_streambuf<wchar_t> wstreambuf;
}

1 The header <streambuf> defines types that control input from and output to character sequences.

27.6.2 Stream buffer requirements

1 Stream buffers can impose various constraints on the sequences they control. Some constraints are:

   (1.1) — The controlled input sequence can be not readable.
   (1.2) — The controlled output sequence can be not writable.
   (1.3) — The controlled sequences can be associated with the contents of other representations for character sequences, such as external files.
   (1.4) — The controlled sequences can support operations directly to or from associated sequences.
   (1.5) — The controlled sequences can impose limitations on how the program can read characters from a sequence, write characters to a sequence, put characters back into an input sequence, or alter the stream position.

2 Each sequence is characterized by three pointers which, if non-null, all point into the same charT array object. The array object represents, at any moment, a (sub)sequence of characters from the sequence. Operations performed on a sequence alter the values stored in these pointers, perform reads and writes directly to or from associated sequences, and alter “the stream position” and conversion state as needed to maintain this subsequence relationship. The three pointers are:

   (2.1) — the beginning pointer, or lowest element address in the array (called xbeg here);
   (2.2) — the next pointer, or next element address that is a current candidate for reading or writing (called xnext here);
   (2.3) — the end pointer, or first element address beyond the end of the array (called xend here).

3 The following semantic constraints shall always apply for any set of three pointers for a sequence, using the pointer names given immediately above:

   (3.1) — If xnext is not a null pointer, then xbeg and xend shall also be non-null pointers into the same charT array, as described above; otherwise, xbeg and xend shall also be null.

§ 27.6.2
If \( x_{\text{next}} \) is not a null pointer and \( x_{\text{next}} < x_{\text{end}} \) for an output sequence, then a write position is available. In this case, \( *x_{\text{next}} \) shall be assignable as the next element to write (to put, or to store a character value, into the sequence).

If \( x_{\text{next}} \) is not a null pointer and \( x_{\text{beg}} < x_{\text{next}} \) for an input sequence, then a putback position is available. In this case, \( x_{\text{next}}[-1] \) shall have a defined value and is the next (preceding) element to store a character that is put back into the input sequence.

If \( x_{\text{next}} \) is not a null pointer and \( x_{\text{next}} < x_{\text{end}} \) for an input sequence, then a read position is available. In this case, \( *x_{\text{next}} \) shall have a defined value and is the next element to read (to get, or to obtain a character value, from the sequence).

### 27.6.3 Class template basic_streambuf<\texttt{charT,traits}>

```cpp
namespace std {
    template <class charT, class traits = \texttt{char_traits<charT>}> 
    class basic_streambuf {
    public:
        // types:
        typedef charT char_type;
        typedef typename traits::int_type int_type;
        typedef typename traits::pos_type pos_type;
        typedef typename traits::off_type off_type;
        typedef traits traits_type;
        virtual ~basic_streambuf();

        // 27.6.3.2.1 locales:
        locale pubimbue(const locale& loc);
        locale getloc() const;

        // 27.6.3.2.2 buffer and positioning:
        basic_streambuf<char_type,traits>* 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:
        // 27.6.3.2.3 Get area:
        streamsize in_avail();
        int_type snextc();
        int_type sbumpc();
        int_type sgetc();
        streamsize sgetn(char_type* s, streamsize n);

        // 27.6.3.2.4 Putback:
        int_type sputbackc(char_type c);
        int_type sungetc();

        // 27.6.3.2.5 Put area:
    }
}
```
The class template `basic_streambuf<charT,traits>` serves as an abstract base class for deriving various stream buffers whose objects each control two character sequences:

---

(1.1) — a character input sequence;
— a character output sequence.

### 27.6.3.1 basic_streambuf constructors

**basic_streambuf();**

Effects: Constructs an object of class `basic_streambuf<charT,traits>` and initializes:\n
1. all its pointer member objects to null pointers,
2. the `getloc()` member to a copy the global locale, `locale()`, at the time of construction.

Remarks: Once the `getloc()` member is initialized, results of calling locale member functions, and of members of facets so obtained, can safely be cached until the next time the member `imbue` is called.

```cpp
basic_streambuf(const basic_streambuf& rhs);
```

Effects: Constructs a copy of `rhs`.

Postconditions:
1. `eback() == rhs.eback()`
2. `gptr() == rhs.gptr()`
3. `egptr() == rhs.egptr()`
4. `pbase() == rhs.pbase()`
5. `pptr() == rhs.pptr()`
6. `eptr() == rhs.eptr()`
7. `getloc() == rhs.getloc()`

```cpp
~basic_streambuf();
```

Effects: None.

### 27.6.3.2 basic_streambuf public member functions

#### 27.6.3.2.1 Locales

**locale pubimbue(const locale& loc);**

Postcondition: `loc == getloc()`.

Effects: Calls `imbue(loc)`.

Returns: Previous value of `getloc()`.

locale getloc() const;

Returns: If `pubimbue()` has ever been called, then the last value of `loc` supplied, otherwise the current global locale, `locale()`, in effect at the time of construction. If called after `pubimbue()` has been called but before `pubimbue()` has returned (i.e., from within the call of `imbue()`) then it returns the previous value.

---

306) The default constructor is protected for class `basic_streambuf` to assure that only objects for classes derived from this class may be constructed.
27.6.3.2.2 Buffer management and positioning

basic_streambuf<char_type,traits>* pubsetbuf(char_type* s, streamsize n);
1
Returns: setbuf(s, n).

pos_type pubseekoff(off_type off, ios_base::seekdir way,
   ios_base::openmode which = ios_base::in | ios_base::out);
2
Returns: seekoff(off, way, which).

pos_type pubseekpos(pos_type sp,
   ios_base::openmode which = ios_base::in | ios_base::out);
3
Returns: seekpos(sp, which).

int pubsync();
4
Returns: sync().

27.6.3.2.3 Get area

streamsize in_avail();
1
Returns: If a read position is available, returns egptr() - gptr(). Otherwise returns showmanyc()
   (27.6.3.4.3).

int_type snextc();
2
Effects: Calls sbumpc().

Returns: If that function returns traits::eof(), returns traits::eof(). Otherwise, returns sgetc().

int_type sbumpc();
3
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();
4
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);
5
Returns: xsgetn(s, n).

27.6.3.2.4 Putback

int_type sputbackc(char_type c);
1
Returns: If the input sequence putback position is not available, or if traits::eq(c,gptr()[-1]) is
   false, returns pbackfail(traits::to_int_type(c)). Otherwise, decrements the next pointer for the
   input sequence and returns traits::to_int_type(*gptr()).

int_type sungetc();
2
Returns: If the input sequence putback position is not available, returns pbackfail(). Otherwise,
   decrements the next pointer for the input sequence and returns traits::to_int_type(*gptr()).
27.6.3.2.5 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).

27.6.3.3 basic_streambuf protected member functions

27.6.3.3.1 Assignment

basic_streambuf& operator=(const basic_streambuf& rhs);

Effects: Assigns the data members of rhs to *this.

Postconditions:
1. eback() == rhs.eback()
2. gptr() == rhs.gptr()
3. egptr() == rhs.egptr()
4. pbase() == rhs.pbase()
5. pptr() == rhs.pptr()
6. epptr() == rhs.epptr()
7. getloc() == rhs.getloc()

Returns: *this.

void swap(basic_streambuf& rhs);

Effects: Swaps the data members of rhs and *this.

27.6.3.3.2 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().
27.6.3.3 Put area access

```cpp
char_type* pbase() const;

Returns: The beginning pointer for the output sequence.
```

```cpp
char_type* pptr() const;

Returns: The next pointer for the output sequence.
```

```cpp
char_type* epptr() const;

Returns: The end pointer for the output sequence.
```

```cpp
void pbump(int n);

Effects: Adds n to the next pointer for the output sequence.
```

```cpp
void setp(char_type* pbeg, char_type* pend);

Postconditions: pbeg == pbase(), pbeg == pptr(), and pend == epptr().
```

27.6.3.4 basic_streambuf virtual functions

27.6.3.4.1 Locales

```cpp
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.

27.6.3.4.2 Buffer management and positioning

```cpp
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 (27.8.2.4, 27.9.1.5).
```

Default behavior: Does nothing. Returns this.

```cpp
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 (27.8.2.4, 27.9.1.5).
```

Default behavior: Returns pos_type(off_type(-1)).

```cpp
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 (27.8.2, 27.9.1.1).
```

Default behavior: Returns pos_type(off_type(-1)).

```cpp
int sync();
```

§ 27.6.3.4.2
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 (27.9.1.5).

Default behavior: Returns zero.

27.6.3.4.3 Get area

```
streamsize showmanyc();
```

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.

Default behavior: Returns zero.

Remarks: Uses `traits::eof()`.

```
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.

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() >= 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:

a) If `gptr()` is non-null, then the `egptr() - gptr()` characters starting at `gptr()`, otherwise the empty sequence.

b) Some sequence (possibly empty) of characters read from the input sequence.

The result character is

a) If the pending sequence is non-empty, the first character of the sequence.

b) If the pending sequence is empty then the next character that would be read from the input sequence.

The backup sequence is defined as the concatenation of:

---

307) The morphemes of `showmanyc` are “es-how-many-see”, not “show-manic”.

308) `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.”

309) Classes derived from `basic_streambuf` can provide more efficient ways to implement `xsgetn()` and `xsputn()` by overriding these definitions from the base class.
a) If `eback()` is null then empty,
b) Otherwise the `gptr() - eback()` characters beginning at `eback()`.

**Effects:** The function sets up the `gptr()` and `egptr()` satisfying one of:

a) If the pending sequence is non-empty, `egptr()` is non-null and `egptr() - gptr()` characters starting at `gptr()` are the characters in the pending sequence
b) If the pending sequence is empty, either `gptr()` is null or `gptr()` and `egptr()` are set to the same non-null pointer value.

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:

a) If the backup sequence contains at least `gptr() - eback()` characters, then the `gptr() - eback()` characters starting at `eback()` agree with the last `gptr() - eback()` characters of the backup sequence.
b) Or the `n` characters starting at `gptr() - n` agree with the backup sequence (where `n` is the length of the backup sequence)

**Default behavior:** Returns `traits::eof()`.

```cpp
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.

### 27.6.3.4.4 Putback

```cpp
int_type pbackfail(int_type c = traits::eof());
```

**Remarks:** The public functions of `basic_streambuf` call this virtual function only when `gptr()` is null, `gptr() == eback()`, or `traits::eq(traits::to_char_type(c),gptr()[-1])` returns `false`. Other calls shall also satisfy that constraint.

The *pending sequence* is defined as for `underflow()`, with the modifications that

(1.1) If `traits::eq_int_type(c,traits::eof())` returns `true`, then the input sequence is backed up one character before the pending sequence is determined.
(1.2) If `traits::eq_int_type(c,traits::eof())` returns `false`, then `c` is prepended. Whether the input sequence is backed up or modified in any other way is unspecified.

**Postcondition:** 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()`.

§ 27.6.3.4.4
27.6.3.4.5 Put area

```
streamsize xmputn(const char_type* s, streamsize n);
```

**Effects:** Writes up to \( n \) characters to the output sequence as if by repeated calls to `sputc(c)`. The characters written are obtained from successive elements of the array whose first element is designated by \( s \). Writing stops when either \( n \) characters have been written or a call to `sputc(c)` would return `traits::eof()`. Is unspecified whether the function calls `overflow()` when `pptr()` becomes true or whether it achieves the same effects by other means.

**Returns:** The number of characters written.

~~~
int_type overflow(int_type c = traits::eof());
~~~

**Effects:** Consumes some initial subsequence of the characters of the pending sequence. The pending sequence is defined as the concatenation of

- a) if \( \text{pbase()} \) is null then the empty sequence otherwise, \( \text{pptr()} - \text{pbase()} \) characters beginning at \( \text{pbase()} \).
- b) if `traits::eq_int_type(c,traits::eof())` returns \true, then the empty sequence otherwise, the sequence consisting of \( c \).

**Remarks:** The member functions `sputc()` and `sputn()` call this function in case that no room can be found in the put buffer enough to accommodate the argument character sequence.

**Requires:** Every overriding definition of this virtual function shall obey the following constraints:

1) The effect of consuming a character on the associated output sequence is specified\(^{310}\)

2) Let \( r \) be the number of characters in the pending sequence not consumed. If \( r \) is non-zero then \( \text{pbase()} \) and \( \text{pptr()} \) shall be set so that: \( \text{pptr()} - \text{pbase()} == r \) and the \( r \) characters starting at \( \text{pbase()} \) are the associated output stream. In case \( r \) is zero (all characters of the pending sequence have been consumed) then either \( \text{pbase()} \) is set to `nullptr`, or \( \text{pbase()} \) and \( \text{pptr()} \) are both set to the same non-null value.

3) The function may fail if either appending some character to the associated output stream fails or if it is unable to establish \( \text{pbase()} \) and \( \text{pptr()} \) according to the above rules.

**Returns:** `traits::eof()` or throws an exception if the function fails.

Otherwise, returns some value other than `traits::eof()` to indicate success.\(^{311}\)

**Default behavior:** Returns `traits::eof()`.

### 27.7 Formatting and manipulators

#### 27.7.1 Overview

**Header `<istream>` synopsis**

```cpp
namespace std {
    template <class charT, class traits = char_traits<charT> >
    class basic_istream;

    typedef basic_istream<char> istream;
    typedef basic_istream<wchar_t> wistream;
}
```

\(^{310}\) That is, for each class derived from an instance of `basic_streambuf` in this Clause (27.8.2, 27.9.1.1), a specification of how consuming a character effects the associated output sequence is given. There is no requirement on a program-defined class.

\(^{311}\) 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)`.

§ 27.7.1
template <class charT, class traits = char_traits<charT> >
  class basic_ostream;
typedef basic_ostream<char>      ostream;
typedef basic_ostream<wchar_t>   wostream;

template <class charT, class traits>
  basic_ostream<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);

Header <iostream> synopsis

namespace std {
  template <class charT, class traits = char_traits<charT> >
    class basic_ostream;
typedef basic_ostream<char>      ostream;
typedef basic_ostream<wchar_t>   wostream;

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);
}

Header <iomanip> synopsis

namespace std {
  // types T1, T2, ... are unspecified implementation types
  T1 resetiosflags(ios_base::fmtflags mask);
  T2 setiosflags (ios_base::fmtflags mask);
  T3 setbase(int base);
  template<charT> T4 setfill(charT c);
  T5 setprecision(int n);
  T6 setw(int n);
  template <class moneyT>
    T7 get_money(moneyT& mon, bool intl = false);
  template <class moneyT>
    T8 put_money(const moneyT& mon, bool intl = false);
  template <class charT>
    T9 get_time(struct tm* tmb, const charT* fmt);
  template <class charT>
    T10 put_time(const struct tm* tmb, const charT* fmt);

template <class charT>
  T11 quoted(const charT* s, charT delim=charT('"'), charT escape=charT('\\'));

template <class charT, class traits, class Allocator>
  T12 quoted(const basic_string<charT, traits, Allocator>& s,
             charT delim=charT('"'), charT escape=charT('\\'));

template <class charT, class traits, class Allocator>
27.7.2 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.

27.7.2.1 Class template basic_istream

```cpp
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_istream : virtual public basic_ios<charT,traits> { 
    public:
      // types (inherited from basic_ios (27.5.5)):
      typedef charT char_type;
      typedef typename traits::int_type int_type;
      typedef typename traits::pos_type pos_type;
      typedef typename traits::off_type off_type;
      typedef traits traits_type;

      // 27.7.2.2.1 Constructor/destructor:
      explicit basic_istream(basic_streambuf<charT,traits>* sb);
      virtual ~basic_istream();

      // 27.7.2.2.3 Prefix/suffix:
      class sentry;

      // 27.7.2.2 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);

      // 27.7.2.3 Unformatted input:
      streamsize gcount() const;

    } // class basic_istream
}
```
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);

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:
    basic_istream(const basic_istream& rhs) = delete;
basic_istream(basic_istream& rhs);

    // 27.7.2.1.2 Assign/swap:
basic_istream& operator=(const basic_istream& rhs) = delete;
basic_istream& operator=(basic_istream& rhs);
    void swap(basic_istream& rhs);
};

    // 27.7.2.2.3 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*);
The class `basic_istream` defines a number of member function signatures that assist in reading and interpreting input from sequences controlled by a stream buffer.

Two groups of member function signatures share common properties: the formatted input functions (or extractors) and the unformatted input functions. Both groups of input functions are described as if they obtain (or extract) input characters by calling `rdbuf()->sbumpc()` or `rdbuf()->sgetc()`. They may use other public members of `istream`.

If `rdbuf()->sbumpc()` or `rdbuf()->sgetc()` returns `traits::eof()`, then the input function, except as explicitly noted otherwise, completes its actions and does `setstate(eofbit)`, which may throw `ios_base::failure` (27.5.5.4), before returning.

If one of these called functions throws an exception, then unless explicitly noted otherwise, the input function sets `badbit` in error state. If `badbit` is on in `exceptions()`, the input function rethrows the exception without completing its actions, otherwise it does not throw anything and proceeds as if the called function had returned a failure indication.

27.7.2.1.1 `basic_istream` constructors

```cpp
explicit basic_istream(basic_streambuf<charT,traits>* sb);
```

1. **Effects:** Constructs an object of class `basic_istream`, assigning initial values to the base class by calling `basic_ios::init(sb)` (27.5.5.2).
2. **Postcondition:** `gcount() == 0`

```cpp
basic_istream(basic_istream&& rhs);
```

1. **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.

```cpp
virtual ~basic_istream();
```

1. **Effects:** Destroys an object of class `basic_istream`.
2. **Remarks:** Does not perform any operations of `rdbuf()`.

27.7.2.1.2 Class `basic_istream` assign and swap

```cpp
basic_istream& operator=(basic_istream&& rhs);
```

1. **Effects:** `swap(rhs);`
2. **Returns:** `*this`.

```cpp
void swap(basic_istream& rhs);
```

1. **Effects:** Calls `basic_ios<charT, traits>::swap(rhs)`. Exchanges the values returned by `gcount()` and `rhs.gcount()`.

27.7.2.1.3 Class `basic_istream::sentry`

```cpp
namespace std {
    template <class charT,class traits = char_traits<charT> >
    class basic_istream<charT,traits>::sentry {
        typedef traits traits_type;
        bool ok_; // exposition only
    }
}
```
The class `sentry` defines a class that is responsible for doing exception safe prefix and suffix operations.

```cpp
explicit sentry(basic_istream<charT,traits>& is, bool noskipws = false);
```

**Effects:** If `is.good()` is `false`, calls `is.setstate(failbit)`. Otherwise, prepares for formatted or unformatted input. First, if `is.tie()` is not a null pointer, the function calls `is.tie()->flush()` to synchronize the output sequence with any associated external C stream. Except that this call can be suppressed if the put area of `is.tie()` is empty. Further an implementation is allowed to defer the call to `flush` until a call of `is.rdbuf()->underflow()` occurs. If no such call occurs before the `sentry` object is destroyed, the call to `flush` may be eliminated entirely. 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 `explicit sentry(basic_istream<charT,traits>& is, bool noskipws = false)` uses the currently imbued locale in `is`, to determine whether the next input character is whitespace or not.

To decide if the character `c` is a whitespace character, the constructor performs as if it executes the following code fragment:

```cpp
const ctype<charT>& ctype = use_facet<ctype<charT>>(is.getloc());
if (ctype.is(ctype.space,c)!=0)
    // c is a whitespace character.
```

If, after any preparation is completed, `is.good()` is `true`, `ok_ != false` otherwise, `ok_ == false`. During preparation, the constructor may call `setstate(failbit)` (which may throw `ios_base::failure`) (27.5.5.4).

```cpp
~sentry();
```

**Effects:** None.

```cpp
explicit operator bool() const;
```

**Effects:** Returns `ok_`.

### 27.7.2.2 Formatted input functions

**27.7.2.2.1 Common requirements**

Each formatted input function begins execution by constructing an object of class `sentry` with the `noskipws` (second) argument `false`. If the `sentry` object returns `true`, when converted to a value of type `bool`, the function endeavors to obtain the requested input. If an exception is thrown during input then `ios::badbit` will be set.

---

312) 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.

313) The sentry constructor and destructor can also perform additional implementation-dependent operations.
is turned on in *this's error state. If (exceptions() & badbit) != 0 then the exception is rethrown. In any case, the formatted input function destroys the sentry object. If no exception has been thrown, it returns *this.

### 27.7.2.2.2 Arithmetic extractors

[istream.formatted.arithmetic]

```cpp
operator>>(unsigned short& val);
operator>>(unsigned int& val);
operator>>(long& val);
operator>>(unsigned long& val);
operator>>(long long& val);
operator>>(unsigned long long& val);
operator>>(float& val);
operator>>(double& val);
operator>>(long double& val);
operator>>(bool& val);
operator>>(void*& val);
```

1. As in the case of the inserters, these extractors depend on the locale’s `num_get<>` (22.4.2.1) object to perform parsing the input stream data. These extractors behave as formatted input functions (as described in 27.7.2.2.1). After a sentry object is constructed, the conversion occurs as if performed by the following code fragment:

   ```cpp
typedef num_get< charT, istreambuf_iterator<charT, traits> > numget;
   iostate err = iostate::goodbit;
   use_facet<numget>(loc).get(*this, 0, *this, err, val);
   setstate(err);
```

   In the above fragment, `loc` stands for the private member of the `basic_ios` class. [Note: The first argument provides an object of the `istreambuf_iterator` class which is an iterator pointed to an input stream. It bypasses istreams and uses streambufs directly. — end note] Class `locale` relies on this type as its interface to `istream`, so that it does not need to depend directly on `istream`.

2. The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

   ```cpp
typedef num_get< charT, istreambuf_iterator<charT, traits> > numget;
   iostate err = iostate::goodbit;
   long lval;
   use_facet<numget>(loc).get(*this, 0, *this, err, lval);
   if (lval < numeric_limits<short>::min()) {
       err |= iostate::failbit;
       val = numeric_limits<short>::min();
   } else if (numeric_limits<short>::max() < lval) {
       err |= iostate::failbit;
       val = numeric_limits<short>::max();
   } else
       val = static_cast<short>(lval);
   setstate(err);
```

```cpp
operator>>(int& val);
```
The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

```c++
typedef num_get<charT, istreambuf_iterator<charT, traits>> numget;
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);
```

\[27.7.2.2.3\] basic_istream::operator\[\texttt{>>}\]

basic_istream<charT, traits>& operator\[\texttt{>>}\]
(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 \[27.7.2.2.1\].

2 Returns: \(\texttt{pf(*this)}\).\(^{315}\)

basic_istream<charT, traits>& operator\[\texttt{>>}\]
(basic_ios<charT, traits>& (*pf)(basic_ios<charT, traits>&&));

3 Effects: Calls \(\texttt{pf(*this)}\). This extractor does not behave as a formatted input function (as described in \[27.7.2.2.1\].

4 Returns: \(\texttt{*this}\).

basic_istream<charT, traits>& operator\[\texttt{>>}\]
(ios_base& (*pf)(ios_base&));

5 Effects: Calls \(\texttt{pf(*this)}\).\(^{316}\) This extractor does not behave as a formatted input function (as described in \[27.7.2.2.1\].

6 Returns: \(\texttt{*this}\).

template<class charT, class traits>
    basic_istream<charT, traits>& operator\[\texttt{>>}\](basic_istream<charT, traits>& in,
        charT* s);

template<class traits>
    basic_istream<char, traits>& operator\[\texttt{>>}\](basic_istream<char, traits>& in,
        unsigned char* s);

template<class traits>
    basic_istream<char, traits>& operator\[\texttt{>>}\](basic_istream<char, traits>& in,
        signed char* s);

7 Effects: Behaves like a formatted input member (as described in \[27.7.2.2.1\]) of \(\texttt{in}\). After a \texttt{sentry} object is constructed, \texttt{operator\[\texttt{>>}\]} extracts characters and stores them into successive locations of an array whose first element is designated by \(s\). If \(\texttt{width()}\) is greater than zero, \(n\) is \(\texttt{width()}\). Otherwise

\(^{315}\) See, for example, the function signature \(\texttt{ws(basic_istream&)}\) (\[27.7.2.4\]).

\(^{316}\) See, for example, the function signature \(\texttt{dec(ios_base&)}\) (\[27.5.6.3\]).
n is the number of elements of the largest array of char_type that can store a terminating charT().

Characters are extracted and stored until any of the following occurs:

- n-1 characters are stored;
- end of file occurs on the input sequence;
- ct_is(ct.space,c) is true for the next available input character c, where ct is use_facet<ctype<
  charT()>(in.getloc()).

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 (27.5.5.4).

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 27.7.2.2.1) of in. After a
sentry object is constructed a character is extracted from in, if one is available, and stored in c. Otherwise,
the function calls in.setstate(failbit).

Returns: in.

```
basic_istream<charT,traits>& operator>>(
  basic_streambuf<charT,traits>* sb);
```

Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1). If sb is null,
calls setstate(failbit), which may throw ios_base::failure (27.5.5.4). 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:

- 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 (in which case the exception is caught).

If the function inserts no characters, it calls setstate(failbit), which may throw ios_base::
failure (27.5.5.4). If it inserted no characters because it caught an exception thrown while extracting
characters from *this and failbit is on in exceptions() (27.5.5.4), then the caught exception is
rethrown.

Returns: *this.
27.7.2.3 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 non-zero 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\(^{317}\) in `*this`'s error state. (Exceptions thrown from `basic_ios<>::clear()` are not caught or rethrown.) If `(exceptions()&badbit) != 0` then the exception is rethrown. It also counts the number of characters extracted. If no exception has been thrown it ends by storing the count in a member object and returning the value specified. In any event the `sentry` object is destroyed before leaving the unformatted input function.

```cpp
streamsize gcount() const;
```

**Effects:** None. This member function does not behave as an unformatted input function (as described in 27.7.2.3, paragraph 1).

**Returns:** The number of characters extracted by the last unformatted input member function called for the object.

```cpp
int_type get();
```

**Effects:** Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1). 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 (27.5.5.4)`,

**Returns:** `c` if available, otherwise `traits::eof()`.

```cpp
basic_istream<charT,traits>& get(char_type& c);
```

**Effects:** Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1). After constructing a sentry object, extracts a character, if one is available, and assigns it to `c`.\(^{318}\) Otherwise, the function calls `setstate(failbit)` (which may throw `ios_base::failure (27.5.5.4)`).

**Returns:** `*this`.

```cpp
basic_istream<charT,traits>& get(char_type* s, streamsize n,
       char_type delim );
```

**Effects:** Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1). After constructing a sentry object, extracts characters and stores them into successive locations of an array whose first element is designated by `s`.\(^{319}\) Characters are extracted and stored until any of the following occurs:

- `n` is less than one or `n - 1` characters are stored;
- end-of-file occurs on the input sequence (in which case the function calls `setstate(eofbit)`);
- `traits::eq(c, delim)` for the next available input character `c` (in which case `c` is not extracted).

If the function stores no characters, it calls `setstate(failbit)` (which may throw `ios_base::failure (27.5.5.4)`). 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`.

\(^{317}\) This is done without causing an `ios::failure` to be thrown.

\(^{318}\) Note that this function is not overloaded on types `signed char` and `unsigned char`.

\(^{319}\) Note that this function is not overloaded on types `signed char` and `unsigned char`.
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 in 27.7.2.3, paragraph 1). 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:

   (13.1) — end-of-file occurs on the input sequence;
   (13.2) — inserting in the output sequence fails (in which case the character to be inserted is not extracted);
   (13.3) — traits::eq(c, delim) for the next available input character c (in which case c is not extracted);
   (13.4) — 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 (27.5.5.4).

   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 in 27.7.2.3, paragraph 1). After constructing a sentry object, extracts characters and stores them into successive locations of an array whose first element is designated by s. Characters are extracted and stored until one of the following occurs:

   1. end-of-file occurs on the input sequence (in which case the function calls setstate(eofbit));
   2. traits::eq(c, delim) for the next available input character c (in which case the input character is extracted but not stored);\(^\text{(321)}\)
   3. n is less than one or n - 1 characters are stored (in which case the function calls setstate( failbit)).

   These conditions are tested in the order shown.\(^\text{(322)}\)
   
   If the function extracts no characters, it calls setstate(failbit) (which may throw ios_base::failure (27.5.5.4)).\(^\text{(323)}\)
   
   In any case, if n is greater than zero, it then stores a null character (using charT()) into the next successive location of the array.

   Returns: *this.

[Example:

\(^{320}\) Note that this function is not overloaded on types signed char and unsigned char.
\(^{321}\) Since the final input character is “extracted,” it is counted in the gcount(), even though it is not stored.
\(^{322}\) This allows an input line which exactly fills the buffer, without setting failbit. This is different behavior than the historical AT&T implementation.
\(^{323}\) This implies an empty input line will not cause failbit to be set.

§ 27.7.2.3
```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;
  }
  — end example]

  basic_istream<charT,traits>& getline(char_type* s, streamsize n);

  Returns: getline(s,n,widen('\n'))

  basic_istream<charT,traits>&
  ignore(streamsize n = 1, int_type delim = traits::eof());

  Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1). After constructing a sentry object, extracts characters and discards them. Characters are extracted until any of the following occurs:

  (25.1) — n != numeric_limits<streamsize>::max() (18.3.2) and n characters have been extracted so far

  (25.2) — end-of-file occurs on the input sequence (in which case the function calls setstate(eofbit), which may throw ios_base::failure (27.5.5.4));

  (25.3) — traits::eq_int_type(traits::to_int_type(c), delim) for the next available input character c (in which case c is extracted).

  Remarks: The last condition will never occur if traits::eq_int_type(delim, traits::eof()).

  Returns: *this.

  int_type peek();

  Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1). 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);

  § 27.7.2.3
```
30 Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1). 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 (27.5.5.4)).

Returns: *this.

31 streamsize readsome(char_type* s, streamsize n);

Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1). 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 (27.5.5.4)), 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.

32 basic_istream<charT,traits>& putback(char_type c);

Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1), 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 (27.5.5.4)). [Note: This function extracts no characters, so the value returned by the next call to gcount() is 0. —end note]

Returns: *this.

33 basic_istream<charT,traits>& unget();

Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1), 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 (27.5.5.4)). [Note: This function extracts no characters, so the value returned by the next call to gcount() is 0. —end note]

Returns: *this.

34 int sync();

Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1), 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 (27.5.5.4), and returns -1. Otherwise, returns zero.

35 pos_type tellg();

324) Note that this function is not overloaded on types signed char and unsigned char.
Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1), 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)`.

```cpp
basic_istream<charT,traits>& seekg(pos_type pos);
```

Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1), 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`.

```cpp
basic_istream<charT,traits>& seekg(off_type off, ios_base::seekdir dir);
```

Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1), 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 `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`.

### 27.7.2.4 Standard basic_istream manipulators

```cpp
namespace std {
    template <class charT, class traits>
    basic_istream<charT,traits>& ws(basic_istream<charT,traits>& is);
}
```

Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1), 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` (27.7.2.1.3). If `ws` stops extracting characters because there are no more available it sets `eofbit`, but not `failbit`.

Returns: `is`.

### 27.7.2.5 Class template basic_iostream

```cpp
namespace std {
    template <class charT, class traits = char_traits<charT> >
    class basic_iostream :
        public basic_istream<charT,traits>,
        public basic_ostream<charT,traits> {
        public:
            // types:
            typedef charT char_type;
            typedef typename traits::int_type int_type;
            typedef typename traits::pos_type pos_type;
            typedef typename traits::off_type off_type;
            typedef traits traits_type;

            // constructor/destructor

§ 27.7.2.5
The class `basic_iostream` inherits a number of functions that allow reading input and writing output to sequences controlled by a stream buffer.

### 27.7.2.5.1 basic_iostream constructors

```cpp
explicit basic_iostream(basic_streambuf<charT, traits>** sb);
```

**Effects:** Constructs an object of class `basic_iostream`, assigning initial values to the base classes by calling `basic_istream<charT, traits>(sb)` (27.7.2.1) and `basic_ostream<charT, traits>(sb)` (27.7.3.1)

**Postcondition:** `rdbuf()==sb` and `gcount()==0`.

### 27.7.2.5.2 basic_iostream destructor

```cpp
virtual basic_iostream();
```

**Effects:** Destroys an object of class `basic_iostream`.

**Remarks:** Does not perform any operations on `rdbuf()`.

### 27.7.2.5.3 basic_iostream assign and swap

```cpp
void swap(basic_iostream& rhs);
```

**Effects:** Calls `basic_istream<charT, traits>::swap(rhs)`.

### 27.7.2.6 Rvalue stream extraction

```cpp
template <class charT, class traits, class T>
basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>&& is, T& x);
```

**Effects:** `is >> x`

**Returns:** `is`
27.7.3 Output streams [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.

27.7.3.1 Class template basic_ostream [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 (27.5.5)):
            typedef charT char_type;
            typedef typename traits::int_type int_type;
            typedef typename traits::pos_type pos_type;
            typedef typename traits::off_type off_type;
            typedef traits traits_type;

            // Constructor/destructor:
            explicit basic_ostream(basic_streambuf<char_type,traits>* sb);
            virtual ~basic_ostream();

            // Prefix/suffix:
            class sentry;

            // Formatted output:
            basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& (*pf)(basic_ostream<charT,traits>&));
            basic_ostream<charT,traits>& operator<<(basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&));
            basic_ostream<charT,traits>& operator<<(ios_base& (*pf)(ios_base&));
            basic_ostream<charT,traits>& operator<<(bool n);
            basic_ostream<charT,traits>& operator<<(short n);
            basic_ostream<charT,traits>& operator<<(unsigned short n);
            basic_ostream<charT,traits>& operator<<(int n);
            basic_ostream<charT,traits>& operator<<(unsigned int n);
            basic_ostream<charT,traits>& operator<<(long n);
            basic_ostream<charT,traits>& operator<<(unsigned long n);
            basic_ostream<charT,traits>& operator<<(long long n);
            basic_ostream<charT,traits>& operator<<(unsigned long long n);
            basic_ostream<charT,traits>& operator<<(float f);
            basic_ostream<charT,traits>& operator<<(double f);
            basic_ostream<charT,traits>& operator<<(const void* p);

            // 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();

            // seeks:
```
The class `basic_ostream` defines a number of member function signatures that assist in formatting and writing output to output sequences controlled by a stream buffer.

Two groups of member function signatures share common properties: the formatted output functions (or *inserters*) and the unformatted output functions. Both groups of output functions generate (or *insert*)
output characters by actions equivalent to calling `rdbuf()->sputc(int_type)`. They may use other public members of `basic_ostream` except that they shall not invoke any virtual members of `rdbuf()` except `overflow()`, `xsputn()`, and `sync()`.

3 If one of these called functions throws an exception, then unless explicitly noted otherwise the output function sets `badbit` in error state. If `badbit` is on in `exceptions()`, the output function rethrows the exception without completing its actions, otherwise it does not throw anything and treat as an error.

### 27.7.3.2 basic_ostream constructors

#### explicit basic_ostream(basic_streambuf<charT,traits>* sb);

1 **Effects:** Constructs an object of class `basic_ostream`, assigning initial values to the base class by calling `basic_ios<charT,traits>::init(sb)` (27.5.5.2).

2 **Postcondition:** `rdbuf()==sb`.

3 **Virtual**

4 **Effects:** Destroys an object of class `basic_ostream`.

5 **Remarks:** Does not perform any operations on `rdbuf()`.

#### basic_ostream(basic_ostream&& rhs);

5 **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.

### 27.7.3.3 Class basic_ostream assign and swap

#### basic_ostream& operator=(basic_ostream&& rhs);

1 **Effects:** `swap(rhs)`.

2 **Returns:** *this.

#### void swap(basic_ostream& rhs);

3 **Effects:** Calls `basic_ios<charT, traits>::swap(rhs)`.

### 27.7.3.4 Class basic_ostream::sentry

#### namespace std {

1 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 (27.5.5.4)`).

The `sentry()` constructor and destructor can also perform additional implementation-dependent operations.

If `((os.flags() & ios_base::unitbuf) && !uncaught_exception() && os.good())` is `true`, calls `os.rdbuf()->pubsync()`. If that function returns `-1`, sets `badbit` in `os.rdstate()` without propagating an exception.

**explicit operator bool() const;**

*Effects: Returns `ok_`.*

### 27.7.3.5 basic_ostream seek members

Each seek member function begins execution by constructing an object of class `sentry`. It returns by destroying the `sentry` object.

`pos_type tellp();`

*Returns: If `fail()` != `false`, returns `pos_type(-1)` to indicate failure. Otherwise, returns `rdbuf()->pubseekoff(0, cur, out)`.*

`basic_ostream<charT,traits>& seekp(pos_type pos);`

*Effects: If `fail()` != `true`, executes `rdbuf()->pubseekpos(pos, ios_base::out)`. In case of failure, the function calls `setstate(failbit)` (which may throw `ios_base::failure`).*

*Returns: *this.*

`basic_ostream<charT,traits>& seekp(off_type off, ios_base::seekdir dir);`

*Effects: If `fail()` != `true`, executes `rdbuf()->pubseekoff(off, dir, ios_base::out)`. In case of failure, the function calls `setstate(failbit)` (which may throw `ios_base::failure`).*

*Returns: *this.*

### 27.7.3.6 Formatted output functions

Each formatted output function begins execution by constructing an object of class `sentry`. If this object returns `true` when converted to a value of type `bool`, the function endeavors to generate the requested output. If the generation fails, then the formatted output function does `setstate(ios_base::failbit)`, which might throw an exception. If an exception is thrown during output, then `ios::badbit` is turned on in `*this`’s error state. If `(exceptions()&badbit) != 0` then the exception is rethrown. Whether or not an exception is thrown, the `sentry` object is destroyed before leaving the formatted output function. If no exception is thrown, the result of the formatted output function is `*this`.

The descriptions of the individual formatted output functions describe how they perform output and do not mention the `sentry` object.

If a formatted output function of a stream `os` determines padding, it does so as follows. Given a `charT` character sequence `seq` where `charT` is the character type of the stream, if the length of `seq` is less than

---

325) The call `os.tie()->flush()` does not necessarily occur if the function can determine that no synchronization is necessary.

326) The `sentry` constructor and destructor can also perform additional implementation-dependent operations.

327) without causing an `ios::failure` to be thrown.
os.width(), then enough copies of os.fill() are added to this sequence as necessary to pad to a width of os.width() characters. If (os.flags() & ios_base::adjustfield) == ios_base::left is true, the fill characters are placed after the character sequence; otherwise, they are placed before the character sequence.

27.7.3.6.2 Arithmetic inserters

operator<<(bool val);
operator<<(short val);
operator<<(unsigned short val);
operator<<(int val);
operator<<(unsigned int val);
operator<<(long val);
operator<<(unsigned long val);
operator<<(long long val);
operator<<(unsigned long long val);
operator<<(float val);
operator<<(double val);
operator<<(long double val);
operator<<(const void* val);

Effects: The classes num_get<> and num_put<> handle locale-dependent numeric formatting and parsing. These inserter functions use the imbued locale value to perform numeric formatting. When val is of type bool, long, unsigned long, long long, unsigned long long, double, long double, or const void*, the formatting conversion occurs as if it performed the following code fragment:

```cpp
bool failed = use_facet<
    num_put<charT,ostreambuf_iterator<charT,traits> >
    >(getloc()).put(*this, *this, fill(), val).failed();
```

When val is of type short the formatting conversion occurs as if it performed the following code fragment:

```cpp
ios_base::fmtflags baseflags = ios_base::flags() & ios_base::basefield;
bool failed = use_facet<
    num_put<charT,ostreambuf_iterator<charT,traits> >
    >(getloc()).put(*this, *this, fill(),
    baseflags == ios_base::oct || baseflags == ios_base::hex
    ? static_cast<long>(static_cast<unsigned short>(val))
    : static_cast<long>(val)).failed();
```

When val is of type int the formatting conversion occurs as if it performed the following code fragment:

```cpp
ios_base::fmtflags baseflags = ios_base::flags() & ios_base::basefield;
bool failed = use_facet<
    num_put<charT,ostreambuf_iterator<charT,traits> >
    >(getloc()).put(*this, *this, fill(),
    baseflags == ios_base::oct || baseflags == ios_base::hex
    ? static_cast<long>(static_cast<unsigned int>(val))
    : static_cast<long>(val)).failed();
```

When val is of type unsigned short or unsigned int the formatting conversion occurs as if it performed the following code fragment:

```cpp
bool failed = use_facet<
    num_put<charT,ostreambuf_iterator<charT,traits> >
    >(getloc()).put(*this, *this, fill(),
    static_cast<unsigned long>(val)).failed();
```

§ 27.7.3.6.2
When `val` is of type `float` the formatting conversion occurs as if it performed the following code fragment:

```c
bool failed = use_facet<
    num_put<charT,ostreambuf_iterator<charT,traits>>
>(getloc()).put(*this, *this, fill(),
    static_cast<double>(val)).failed();
```

The first argument provides an object of the `ostreambuf_iterator<>` class which is an iterator for class `basic_ostream<>`. It bypasses `ostreams` and uses `streambufs` directly. Class `locale` relies on these types as its interface to `iostreams`, since for flexibility it has been abstracted away from direct dependence on `ostream`. The second parameter is a reference to the base 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`.

### 27.7.3.6.3 basic_ostream::operator<<

`basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& (*pf)(basic_ostream<charT,traits>&));`

**Effects:** None. Does not behave as a formatted output function (as described in 27.7.3.6.1).

**Returns:** `pf(*this)`.

**basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& (*pf)(basic_ostream<charT,traits>&));**

`basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& (*pf)(basic_ostream<charT,traits>&));`

**Effects:** Calls `pf(*this)`. This inserter does not behave as a formatted output function (as described in 27.7.3.6.1).

**Returns:** `*this`.

**basic_ostream<charT,traits>& operator<<(basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&));**

`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 27.7.3.6.1).

**Returns:** `*this`.

**basic_ostream<charT,traits>& operator<<(basic_streambuf<charT,traits>* sb);**

`basic_ostream<charT,traits>& operator<<(basic_streambuf<charT,traits>* sb);`

**Effects:** Behaves as an unformatted output function (as described in 27.7.3.7, paragraph 1). 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`.

---

328) See, for example, the function signature `endl(basic_ostream&)` (27.7.3.8).

329) See, for example, the function signature `dec(ios_base&)` (27.5.6.3).
If the function inserts no characters, it calls `setstate(failbit)` (which may throw `ios_base::failure` (27.5.5.4)). 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.

27.7.3.6.4 Character inserter function templates

```cpp
template<class charT, class traits>
basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& out, charT c);

// specialization
template<class traits>
basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out, char c);

// signed and unsigned
template<class traits>
basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out, signed char c);

template<class traits>
basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out, unsigned char c);
```

Effects: Behaves as a formatted output function (27.7.3.6.1) 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 27.7.3.6.1. Inserts `seq` into `out`. Calls `os.width(0)`.

Returns: `out`.

```cpp
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 27.7.3.6.1) of `out`. Creates a character sequence `seq` of `n` characters starting at `s`, each widened using `out.widen()` (27.5.5.3), where `n` is the number that would be computed as if by:

(4.1) \[ \text{traits::length(s)} \]

§ 27.7.3.6.4

1062
std::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*,

traits::length(reinterpret_cast<const char*>(s)) for the other two overloads.

Determines padding for seq as described in 27.7.3.6.1. Inserts seq into out. Calls width(0).

Returns: out.

27.7.3.7 Unformatted output functions

Each unformatted output function begins execution by constructing an object of class sentry. If this object returns true, while converting to a value of type bool, the function endeavors to generate the requested output. If an exception is thrown during output, then ios::badbit is turned on\(^{330}\) 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 in 27.7.3.7, paragraph 1). After constructing a sentry object, inserts the character c, if possible.\(^{331}\) Otherwise, calls setstate(badbit) (which may throw ios_base::failure (27.5.5.4)).

Returns: *this.

basic_ostream& write(const char_type* s, streamsize n);

Effects: Behaves as an unformatted output function (as described in 27.7.3.7, paragraph 1). After constructing a sentry object, obtains characters to insert from successive locations of an array whose first element is designated by s.\(^{332}\) Characters are inserted until either of the following occurs:

(5.1) n characters are inserted;
(5.2) inserting in the output sequence fails (in which case the function calls setstate(badbit), which may throw ios_base::failure (27.5.5.4)).

Returns: *this.

basic_ostream& flush();

Effects: Behaves as an unformatted output function (as described in 27.7.3.6.1, paragraph 1). If rdbuf() is not a null pointer, constructs a sentry object. If this object returns true when converted to a value of type bool the function calls rdbuf()->pubsync(). If that function returns -1 calls setstate(badbit) (which may throw ios_base::failure (27.5.5.4)). Otherwise, if the sentry object returns false, does nothing.

Returns: *this.

27.7.3.8 Standard basic_ostream manipulators

namespace std {
    template <class charT, class traits>
    basic_ostream<charT, traits>& endl(basic_ostream<charT, traits>& os);
}

\(^{330}\) without causing an ios::failure to be thrown.

\(^{331}\) Note that this function is not overloaded on types signed char and unsigned char.

\(^{332}\) Note that this function is not overloaded on types signed char and unsigned char.
Effects: Calls `os.put(os.widen(\'\n\'))`, then `os.flush()`.

Returns: `os`.

```cpp
namespace std {
    template <class charT, class traits>
    basic_ostream<charT, traits>& ends(basic_ostream<charT, traits>&& os);
    }  // namespace
```

Effects: Inserts a null character into the output sequence: calls `os.put(charT())`.

Returns: `os`.

```cpp
namespace std {
    template <class charT, class traits>
    basic_ostream<charT, traits>& flush(basic_ostream<charT, traits>& os);
    }  // namespace
```

Effects: Calls `os.flush()`.

Returns: `os`.

### 27.7.3.9 Rvalue stream insertion

[ostream.rvalue]

```cpp
template <class charT, class traits, class T>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>&& os, const T& x);
```

Effects: `os << x`

Returns: `os`

### 27.7.4 Standard manipulators

[std.manip]

The header `<iomanip>` defines several functions that support extractors and inserters that alter information maintained by class `ios_base` and its derived classes.

```cpp
unsatisfactory  resetiosflags(ios_base::fmtflags mask);
```

Returns: An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out << resetiosflags(mask)` behaves as if it called `f(out, mask)`, or if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >> resetiosflags(mask)` behaves as if it called `f(in, mask)`, where the function `f` is defined as:

```cpp
void f(ios_base& str, ios_base::fmtflags mask) {
    // reset specified flags
    str.setf(ios_base::fmtflags(0), mask);
}
```

The expression `out << resetiosflags(mask)` shall have type `basic_ostream<charT, traits>&` and value `out`. The expression `in >> resetiosflags(mask)` shall have type `basic_istream<charT, traits>&` and value `in`.

```cpp
unsatisfactory  setiosflags(ios_base::fmtflags mask);
```

333) The expression `cin >> resetiosflags(ios_base::skipws)` clears `ios_base::skipws` in the format flags stored in the `basic_istream<charT, traits>` object `cin` (the same as `cin >> noskipws`), and the expression `cout << resetiosflags(ios_base::showbase)` clears `ios_base::showbase` in the format flags stored in the `basic_ostream<charT, traits>` object `cout` (the same as `cout << no_showbase`).
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 base field
    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`.

```c
uns_specified setw(int n);
```

*Returns:* An object of unspecified type such that if `out` is an instance of `basic_ostream<charT, traits>` then the expression `out << setw(n)` behaves as if it called `f(out, n)`, or if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >> setw(n)` behaves as if it called `f(in, n)`, where the function `f` is defined as:

```c
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`.

### 27.7.5 Extended manipulators

The header `<iomanip>` defines several functions that support extractors and inserters that allow for the parsing and formatting of sequences and values for money and time.

```c
template <class moneyT> unspecified get_money(moneyT& mon, bool intl = false);
```

*Requires:* The type `moneyT` shall be either `long double` or a specialization of the `basic_string` template (Clause 21).

*Effects:* The expression `in >> get_money(mon, intl)` described below behaves as a formatted input function (27.7.2.2.1).

*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:

```c
template <class charT, class traits, class moneyT>
void f(basic_ios<charT, traits>& str, moneyT& mon, bool intl) {
  typedef istreambuf_iterator<charT, traits> Iter;
  typedef money_get<charT, Iter> MoneyGet;
  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`.

```c
template <class moneyT> unspecified put_money(const moneyT& mon, bool intl = false);
```

*Requires:* The type `moneyT` shall be either `long double` or a specialization of the `basic_string` template (Clause 21).

§ 27.7.5
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 input function 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) {
    typedef ostreambuf_iterator<charT, traits> Iter;
    typedef money_put<charT, Iter> MoneyPut;

    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`.

```cpp
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`, 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 `in` is an object of type `basic_istream<charT, traits>` then the expression `in >> get_time(tmb, fmt)` behaves as if it called `f(in, tmb, fmt)`, where the function `f` is defined as:

```cpp
template <class charT, class traits>
void f(basic_ios<charT, traits>& str, struct tm* tmb, const charT* fmt) {
    typedef istreambuf_iterator<charT, traits> Iter;
    typedef time_get<charT, Iter> TimeGet;

    ios_base::iostate err = ios_base::goodbit;
    const TimeGet& tg = use_facet<TimeGet>(str.getloc());

    tg.get(Iter(str.rdbuf())), Iter(), str, err, tmb,
        fmt, fmt + traits::length(fmt));

    if (err != ios_base::goodbit)
        str.setstate(err);
}
```

The expression `in >> get_time(tmb, fmt)` shall have type `basic_istream<charT, traits>&` and value `in`.

```cpp
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:
template <class charT, class traits>
void f(basic_ios<charT, traits>& str, const struct tm* tmb, const charT* fmt) {
    typedef ostreambuf_iterator<charT, traits> Iter;
    typedef time_put<charT, Iter> TimePut;
    const TimePut& tp = use_facet<TimePut>(str.getloc());
    const Iter end = tp.put(Iter(str.rdbuf()), str, str.fill(), tmb,
        fmt, fmt + traits::length(fmt));
    if (end.failed())
        str.setstate(ios_base::badbit);
}

The expression out << put_time(tmb, fmt) shall have type basic_ostream<charT, traits>& and value out.

27.7.6 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]

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('\\'));

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 form is the same as traits, then the expression out << quoted(s, delim, escape) behaves as a formatted output function (27.7.3.6.1) of out. This forms a character sequence seq, initially consisting of the following elements:

1. delim.
2. Each character in s. If the character to be output is equal to escape or delim, as determined by traits_type::eq, first output escape.
3. delim.

Let x be the number of elements initially in seq. Then padding is determined for seq as described in 27.7.3.6.1, 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:

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 basic_istream::operator>> (27.7.2.2.3) which may throw ios_base::failure (27.5.3.1.1):
2. 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 '!', 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`.

27.8 String-based streams

27.8.1 Overview

The header `<sstream>` defines four class templates and eight types that associate stream buffers with objects of class `basic_string`, as described in 21.3.

Header `<sstream>` synopsis

```cpp
namespace std {
    template <class charT, class traits = char_traits<charT>,
              class Allocator = allocator<charT> >
    class basic_stringbuf;

typedef basic_stringbuf<char> stringbuf;
typedef basic_stringbuf<wchar_t> wstringbuf;

template <class charT, class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
    class basic_istringstream;

typedef basic_istringstream<char> istringstream;
typedef basic_istringstream<wchar_t> wistringstream;

template <class charT, class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
    class basic_ostringstream;

typedef basic_ostringstream<char> ostringstream;
typedef basic_ostringstream<wchar_t> wostringstream;

template <class charT, class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
    class basic_stringstream;

typedef basic_stringstream<char> stringstream;
typedef basic_stringstream<wchar_t> wstringstream;
}
```
27.8.2 Class template basic_stringbuf

namespace std {
    template <class charT, class traits = char_traits<charT>,
              class Allocator = allocator<charT> >
    class basic_stringbuf : public basic_streambuf<charT,traits> {
public:
    typedef charT char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits traits_type;
    typedef Allocator allocator_type;

    // 27.8.2.1 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);
    // 27.8.2.2 Assign and swap:
    basic_stringbuf& operator=(const basic_stringbuf& rhs) = delete;
    basic_stringbuf& operator=(basic_stringbuf&& rhs);
    void swap(basic_stringbuf& rhs);
    // 27.8.2.3 Get and set:
    basic_string<charT,traits,Allocator> str() const;
    void str(const basic_string<charT,traits,Allocator>& s);

    protected:
    // 27.8.2.4 Overridden virtual functions:
    virtual int_type underflow();
    virtual int_type pbackfail(int_type c = traits::eof());
    virtual int_type overflow (int_type c = traits::eof());
    virtual basic_streambuf<charT,traits>* setbuf(charT*, streamsize);

    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);

    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); 
};

§ 27.8.2 1070
The class \texttt{basic\_stringbuf} is derived from \texttt{basic\_streambuf} to associate possibly the input sequence and possibly the output sequence with a sequence of arbitrary \texttt{characters}. The sequence can be initialized from, or made available as, an object of class \texttt{basic\_string}.

For the sake of exposition, the maintained data is presented here as:

\begin{enumerate}
\item \texttt{ios\_base::openmode mode}, has \texttt{in} set if the input sequence can be read, and \texttt{out} set if the output sequence can be written.
\end{enumerate}

\subsection*{27.8.2.1 basic\_stringbuf constructors}

\begin{verbatim}
explicit basic_stringbuf(ios_base::openmode which =
    ios_base::in | ios_base::out);
\end{verbatim}

\textit{Effects:} Constructs an object of class \texttt{basic\_stringbuf}, initializing the base class with \texttt{basic\_streambuf()} (27.6.3.1), and initializing \texttt{mode} with \texttt{which}.

\textit{Postcondition:} \texttt{str()} == "".

\begin{verbatim}
explicit basic_stringbuf(const basic_string<charT,traits,Allocator>& s,
    ios_base::openmode which = ios_base::in | ios_base::out);
\end{verbatim}

\textit{Effects:} Constructs an object of class \texttt{basic\_stringbuf}, initializing the base class with \texttt{basic\_streambuf()} (27.6.3.1), and initializing \texttt{mode} with \texttt{which}. Then calls \texttt{str(s)}.

\begin{verbatim}
basic_stringbuf(basic_stringbuf&& rhs);
\end{verbatim}

\textit{Effects:} Move constructs from the rvalue \texttt{rhs}. It is implementation-defined whether the sequence pointers in \texttt{*this (eback(), gpstr(), egptr(), pbase(), pptr(), epptr())} obtain the values which \texttt{rhs} had. Whether they do or not, \texttt{*this} and \texttt{rhs} reference separate buffers (if any at all) after the construction. The openmode, locale and any other state of \texttt{rhs} is also copied.

\textit{Postconditions:} Let \texttt{rhs\_p} refer to the state of \texttt{rhs} just prior to this construction and let \texttt{rhs\_a} refer to the state of \texttt{rhs} just after this construction.

\begin{enumerate}
\item \texttt{str()} == \texttt{rhs\_p.str()}
\item \texttt{gpstr()} - \texttt{eback()} == \texttt{rhs\_p.gpstr()} - \texttt{rhs\_p.eback()}
\item \texttt{egptr()} - \texttt{eback()} == \texttt{rhs\_p.egptr()} - \texttt{rhs\_p.eback()}
\item \texttt{pptr()} - \texttt{pbase()} == \texttt{rhs\_p.pptr()} - \texttt{rhs\_p.pbase()}
\item \texttt{epptr()} - \texttt{pbase()} == \texttt{rhs\_p.epptr()} - \texttt{rhs\_p.pbase()}
\item if (\texttt{eback()} \texttt{eback()} != \texttt{rhs\_a.eback()})
\item if (\texttt{gpstr()} \texttt{gpstr()} != \texttt{rhs\_a.gpstr()})
\item if (\texttt{egptr()} \texttt{egptr()} != \texttt{rhs\_a.egptr()})
\item if (\texttt{pbase()} \texttt{pbase()} != \texttt{rhs\_a.pbase()})
\item if (\texttt{pptr()} \texttt{pptr()} != \texttt{rhs\_a.pptr()})
\item if (\texttt{epptr()} \texttt{epptr()} != \texttt{rhs\_a.epptr()})
\end{enumerate}

\subsection*{27.8.2.2 Assign and swap}

\begin{verbatim}
basic_stringbuf\& operator=(basic_stringbuf\&\& rhs);
\end{verbatim}

\textit{Effects:} After the move assignment \texttt{*this} has the observable state it would have had if it had been move constructed from \texttt{rhs} (see 27.8.2.1).

\textit{Returns:} \texttt{*this}.

\begin{verbatim}
void swap(basic_stringbuf\& rhs);
\end{verbatim}

\section*{§ 27.8.2.2}
Effects: Exchanges the state of *this and rhs.

```cpp
template <class charT, class traits, class Allocator>
void swap(basic_stringbuf<charT, traits, Allocator>& x,
    basic_stringbuf<charT, traits, Allocator>& y);
```

Effects: x.swap(y).

### 27.8.2.3 Member functions

```cpp
basic_string<charT, traits, Allocator> str() const;
```

Returns: A basic_string object whose content is equal to the basic_stringbuf underlying character sequence. If the basic_stringbuf was created only in input mode, the resultant basic_string contains the character sequence in the range [eback(), egptr()). If the basic_stringbuf was created with which & ios_base::out being true then the resultant basic_string contains the character sequence in the range [pbase(), high_mark), where high_mark represents the position one past the highest initialized character in the buffer. Characters can be initialized by writing to the stream, by constructing the basic_stringbuf with a basic_string, or by calling the str(basic_string) member function. In the case of calling the str(basic_string) member function, all characters initialized prior to the call are now considered uninitialized (except for those characters re-initialized by the new basic_string). Otherwise the basic_stringbuf has been created in neither input nor output mode and a zero length basic_string is returned.

```cpp
void str(const basic_string<charT, traits, Allocator>& s);
```

Effects: Copies the content of s into the basic_stringbuf underlying character sequence and initializes the input and output sequences according to mode.

Postconditions: If mode & ios_base::out is true, pbase() points to the first underlying character and eptr() >= pbase() + s.size() holds; in addition, if mode & ios_base::ate is true, pptr() == pbase() + s.size() holds, otherwise pptr() == pbase() is true. If mode & ios_base::in is true, eback() points to the first underlying character, and both gptr() == eback() and egptr() == eback() + s.size() hold.

### 27.8.2.4 Overridden virtual functions

```cpp
int_type underflow();
```

Returns: If the input sequence has a read position available, returns traits::to_int_type(*gptr()). Otherwise, returns traits::eof(). Any character in the underlying buffer which has been initialized is considered to be part of the input sequence.

```cpp
int_type pbackfail(int_type c = traits::eof());
```

Effects: Puts back the character designated by c to the input sequence, if possible, in one of three ways:

1. If traits::eq_int_type(c, traits::eof()) returns false and if the input sequence has a putback position available, and if traits::eq(to_char_type(c), gptr()[−1]) returns true, assigns gptr() − 1 to gptr().
   Returns: c.

2. If traits::eq_int_type(c, traits::eof()) returns false and if the input sequence has a putback position available, and if mode & ios_base::out is nonzero, assigns c to *−−gptr().
   Returns: c.

§ 27.8.2.4
— If `traits::eq_int_type(c, traits::eof())` returns `true` and if the input sequence has a put-
back position available, assigns `gptr() - 1` to `gptr()`.

Returns: `traits::not_eof(c)`.

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());
```

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: `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);
```

Effects: Alters the stream position within one of the controlled sequences, if possible, as indicated in
Table 130.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Result</th>
</tr>
</thead>
</table>
| `(which & ios_base::in) == ios_-
  base::in` | positions the input sequence |
| `(which & ios_base::out) == ios_-
  base::out` | positions the output sequence |
| `(which & (ios_base::in |
  ios_base::out)) ==
  (ios_base::in |
  ios_base::out))` and way == either
  `ios_base::beg` or
  `ios_base::end` | positions both the input and the output sequences |
| Otherwise | the positioning operation fails. |

For a sequence to be positioned, if its next pointer (either `gptr()` or `pptr()`) is a null pointer and
the new offset `newoff` is nonzero, the positioning operation fails. Otherwise, the function determines
`newoff` as indicated in Table 131.
Table 131 — 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 beginning pointer (xnext - xbeg).</td>
</tr>
<tr>
<td>way == ios_base::end</td>
<td>the high mark pointer minus the beginning pointer (high_mark - xbeg).</td>
</tr>
</tbody>
</table>

If \((\text{newoff} + \text{off}) < 0\), or if \(\text{newoff} + \text{off}\) refers to an uninitialized character (as defined in 27.8.2.3 paragraph 1), the positioning operation fails. Otherwise, the function assigns \(\text{xbeg} + \text{newoff} + \text{off}\) to the next pointer \(\text{xnext}\).

\(\text{Returns: pos_type(\text{newoff})}\), constructed from the resultant offset \(\text{newoff}\) (of type \(\text{off_type}\)), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is \(\text{pos_type(\text{off_type}(-1))}\).

\(\text{pos_type seekpos(pos_type \text{sp}, \text{ios_base::openmode which} = \text{ios_base::in} \mid \text{ios_base::out});}\)

\(\text{Effects: Equivalent to seekoff(\text{off_type(\text{sp})}, \text{ios_base::beg}, \text{which});}\)

\(\text{Returns: \text{sp} to indicate success, or pos_type(\text{off_type}(-1)) to indicate failure.}\)

\(\text{basic_streambuf<charT,traits>* setbuf(charT* \text{s}, streamsize \text{n});}\)

\(\text{Effects: implementation-defined, except that setbuf(0,0) has no effect.}\)

\(\text{Returns: this.}\)

27.8.3 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:
            typedef charT char_type;
            typedef typename traits::int_type int_type;
            typedef typename traits::pos_type pos_type;
            typedef typename traits::off_type off_type;
            typedef traits traits_type;
            typedef Allocator allocator_type;

            // 27.8.3.1 Constructors:
            explicit basic_istringstream(ios_base::openmode which = ios_base::in);
            explicit basic_istringstream(
                const basic_string<charT,traits,Allocator>& \text{str},
                ios_base::openmode which = ios_base::in);
            basic_istringstream(const basic_istringstream& \text{rhs}) = delete;
            basic_istringstream(basic_istringstream&& \text{rhs});

            // 27.8.3.2 Assign and swap:
            basic_istringstream& operator=(const basic_istringstream& \text{rhs}) = delete;
            basic_istringstream& operator=(basic_istringstream&& \text{rhs});
            void swap(basic_istringstream& \text{rhs});

    }
The class `basic_istringstream<charT, traits, Allocator>` supports reading objects of class `basic_string<charT, traits, Allocator>`. It uses a `basic_stringbuf<charT, traits, Allocator>` object to control the associated storage. For the sake of exposition, the maintained data is presented here as:

\[ (1.1) \quad \text{sb, the stringbuf object.} \]

### 27.8.3.1 `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(&sb)` and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(which | ios_base::in)` (27.8.2.1).

**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(&sb)` and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::in)` (27.8.2.1).

**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`.

### 27.8.3.2 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: \texttt{x.swap(y)}.

27.8.3.3 Member functions

\begin{verbatim}
basic_stringbuf<charT,traits,Allocator>* rdbuf() const;

Returns: \texttt{const_cast<basic_stringbuf<charT,traits,Allocator>*>(&sb)}.

basic_string<charT,traits,Allocator> str() const;

Returns: \texttt{rdbuf()->str().}

void str(const basic_string<charT,traits,Allocator>& s);

Effects: Calls \texttt{rdbuf()->str(s)}.
\end{verbatim}

27.8.4 Class template basic\_ostreamstring

\begin{verbatim}
namespace std {
  template <class charT, class traits = char_traits<charT>,
            class Allocator = allocator<charT> >
  class basic\_ostreamstring : public basic\_ostream<charT,traits> {

    public:

    // types:
    typedef charT char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits traits_type;
    typedef Allocator allocator_type;

    // 27.8.4.1 Constructors/destructor:
    explicit basic\_ostreamstring(ios_base::openmode which = ios_base::out);
    explicit basic\_ostreamstring(const basic\_string<charT,traits,Allocator>& str,
                                ios_base::openmode which = ios_base::out);
    basic\_ostreamstring(const basic\_ostreamstring& rhs) = delete;
    basic\_ostreamstring(basic\_ostreamstring&& rhs);

    // 27.8.4.2 Assign/swap:
    basic\_ostreamstring& operator=(const basic\_ostreamstring& rhs) = delete;
    basic\_ostreamstring& operator=(basic\_ostreamstring&& rhs);
    void swap(basic\_ostreamstring& rhs);

    // 27.8.4.3 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\_ostreamstring<charT, traits, Allocator>& x,
            basic\_ostreamstring<charT, traits, Allocator>& y);
}
\end{verbatim}
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.

### 27.8.4.1 `basic_ostringstream` constructors

#### explicit `basic_ostringstream(ios_base::openmode which = ios_base::out);`

**Effects:** Constructs an object of class `basic_ostringstream`, initializing the base class with `basic_ostream(&sb)` and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(which | ios_base::out)` (27.8.2.1).

#### 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(&sb)` and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::out)` (27.8.2.1).

#### `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`.

### 27.8.4.2 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`.

**Returns:** `*this`.

#### `void swap(basic_ostringstream& rhs);`

**Effects:** Exchanges the state of `*this` and `rhs` by calling `basic_ostream<charT,traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

#### `template <class charT, class traits, class Allocator> void swap(basic_ostringstream<charT, traits, Allocator>& x, basic_ostringstream<charT, traits, Allocator>& y);`

**Effects:** `x.swap(y)`.

### 27.8.4.3 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)`.
27.8.5 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:

        // types:
        typedef charT char_type;
        typedef typename traits::int_type int_type;
        typedef typename traits::pos_type pos_type;
        typedef typename traits::off_type off_type;
        typedef traits traits_type;
        typedef Allocator allocator_type;

        // constructors/destructor
        explicit basic_stringstream(
            ios_base::openmode which = ios_base::out|ios_base::in);
        explicit basic_stringstream(
            const basic_string<charT,traits,Allocator>& str,
            ios_base::openmode which = ios_base::out|ios_base::in);
        basic_stringstream(const basic_stringstream& rhs) = delete;
        basic_stringstream(basic_stringstream&& rhs);

        // 27.8.5.2 Assign/swap:
        basic_stringstream& operator=(const basic_stringstream& rhs) = delete;
        basic_stringstream& operator=(basic_stringstream&& rhs);
        void swap(basic_stringstream& rhs);

        // Members:
        basic_stringbuf<charT, traits, Allocator>* rdbuf() const;
        basic_string<charT, traits, Allocator> str() const;
        void str(const basic_string<charT, traits, Allocator>& str);

    private:
        basic_stringbuf<charT, traits> sb; // exposition only
    }

    template <class charT, class traits, class Allocator>
    void swap(basic_stringstream<charT, traits, Allocator>& x,
              basic_stringstream<charT, traits, Allocator>& y);
}

The class template basic_stringstream<charT, traits> supports reading and writing from objects of class basic_string<charT, traits, Allocator>. It uses a basic_stringbuf<charT, traits, Allocator> object to control the associated sequence. For the sake of exposition, the maintained data is presented here as

(1.1) — sb, the stringbuf object.

27.8.5.1 basic_stringstream constructors

explicit basic_stringstream(
    ios_base::openmode which = ios_base::out|ios_base::in);
1. **Effects:** Constructs an object of class `basic_stringstream<charT,traits>`, initializing the base class with `basic_iostream(&sb)` and initializing `sb` with `basic_stringbuf<charT,traits,Allocator>(which)`. 

   `explicit basic_stringstream(
    const basic_string<charT,traits,Allocator>& str,
    ios_base::openmode which = ios_base::out|ios_base::in);`

2. **Effects:** Constructs an object of class `basic_stringstream<charT, traits>`, initializing the base class with `basic_iostream(&sb)` and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which)`. 

   `basic_stringstream(basic_stringstream&& rhs);`

3. **Effects:** Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_stringbuf`. Next `basic_istream<charT,traits>::set_rdbuf(&sb)` is called to install the contained `basic_stringbuf.`

27.8.5.2 Assign and swap [stringstream.assign]

   `basic_stringstream& operator=(basic_stringstream&& rhs);`

1. **Effects:** Move assigns the base and members of `*this` from the base and corresponding members of `rhs`. 

2. **Returns:** `*this`.

   `void swap(basic_stringstream& rhs);`

3. **Effects:** Exchanges the state of `*this` and `rhs` by calling `basic_iostream<charT,traits>::swap(rhs)` and `sb.swap(rhs.sb).`

   `template <class charT, class traits, class Allocator>
   void swap(basic_stringstream<charT, traits, Allocator>& x,
             basic_stringstream<charT, traits, Allocator>& y);`

4. **Effects:** `x.swap(y)`.

27.8.5.3 Member functions [stringstream.members]

   `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>& str);`

3. **Effects:** Calls `rdbuf()->str(str).`

27.9 File-based streams [file.streams]

27.9.1 File streams [fstreams]

1. The header `<fstream>` defines four class templates and eight types that associate stream buffers with files and assist reading and writing files.

   **Header `<fstream>` synopsis**
namespace std {

template <class charT, class traits = char_traits<charT> >
  class basic_filebuf;
typedef basic_filebuf<char>    filebuf;
typedef basic_filebuf<wchar_t> wfilebuf;

template <class charT, class traits = char_traits<charT> >
  class basic_ifstream;
typedef basic_ifstream<char>  ifstream;
typedef basic_ifstream<wchar_t> wifstream;

template <class charT, class traits = char_traits<charT> >
  class basic_ofstream;
typedef basic_ofstream<char>  ofstream;
typedef basic_ofstream<wchar_t> wofstream;

template <class charT, class traits = char_traits<charT> >
  class basic_fstream;
typedef basic_fstream<char>  fstream;
typedef basic_fstream<wchar_t> wfstream;
}

2 In this subclause, the type name FILE refers to the type FILE declared in <cstdio> (27.9.2).

3 [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]

27.9.1.1 Class template basic_filebuf

namespace std {

template <class charT, class traits = char_traits<charT> >
  class basic_filebuf : public basic_streambuf<charT,traits> {
public:
  typedef charT      char_type;
  typedef typename traits::int_type int_type;
  typedef typename traits::pos_type pos_type;
  typedef typename traits::off_type off_type;
  typedef traits    traits_type;

  // 27.9.1.1.2 Constructors/destructor:
  basic_filebuf();
  basic_filebuf(const basic_filebuf& rhs) = delete;
  basic_filebuf(basic_filebuf& rhs);
  virtual ~basic_filebuf();

  // 27.9.1.3 Assign/swap:
  basic_filebuf& operator=(const basic_filebuf& rhs) = delete;
  basic_filebuf& operator=(basic_filebuf& rhs);
  void swap(basic_filebuf& rhs);

  // 27.9.1.4 Members:
  bool is_open() const;
  basic_filebuf<charT,traits>  * open(const char* s,
                                      ios_base::openmode mode);
  basic_filebuf<charT,traits>  * open(const string& s,  
                                      ios_base::openmode mode);

}
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 Standard C library `FILE`s.

In particular:

1. If the file is not open for reading the input sequence cannot be read.
2. If the file is not open for writing the output sequence cannot be written.
3. A joint file position is maintained for both the input sequence and the output sequence.

An instance of `basic_filebuf` behaves as described in §27.9.1.1 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 sections, obtained as if by

```cpp
const codecvt<charT,char,typename traits::state_type>& a_codecvt = use_facet<codecvt<charT,char,typename traits::state_type>>(getloc());
```

### 27.9.1.2 `basic_filebuf` Constructors

#### [filebuf.cons] `basic_filebuf`;

#### §27.9.1.2

1. **Effects:** Constructs an object of class `basic_filebuf<charT,traits>`, initializing the base class with `basic_streambuf<charT,traits>()`. (27.6.3.1)

2. **Postcondition:** `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()
(4.5)  pptr() - pbase() == rhs_p.pptr() - rhs_p.pbase()
(4.6)  epptr() - pbase() == rhs_p.epptr() - rhs_p.pbase()
(4.7)  if (eback()) eback() != rhs_a.eback()
(4.8)  if (gptr()) gptr() != rhs_a.gptr()
(4.9)  if (egptr()) egptr() != rhs_a.egptr()
(4.10) if (pbase()) pbase() != rhs_a.pbase()
(4.11) if (pptr()) pptr() != rhs_a.pptr()
(4.12) if (epptr()) epptr() != rhs_a.epptr()

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 17.6.5.12).

27.9.1.3 Assign and swap

basic_filebuf& operator=(basic_filebuf&& rhs);

Effects: Calls this->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 27.9.1.2).

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: x.swap(y).
27.9.1.4 Member functions

bool is_open() const;

Returns: true if a previous call to open succeeded (returned a non-null value) and there has been no intervening call to close.

basic_filebuf<charT,traits*>* open(const char* s,
ios_base::openmode mode);

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 std::fopen(s,modstr)). The NTBS modstr is determined from mode & ~ios_base::ate as indicated in Table 132. If mode is not some combination of flags shown in the table then the open fails.

Table 132 — File open modes

<table>
<thead>
<tr>
<th>binary</th>
<th>flag combination</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>&quot;w&quot;</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>&quot;a&quot;</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>&quot;u&quot;</td>
</tr>
<tr>
<td>+</td>
<td>&quot;r&quot;</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>&quot;r+&quot;</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>&quot;w+&quot;</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>&quot;a+&quot;</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>&quot;a+b&quot;</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>&quot;wb&quot;</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>&quot;ab&quot;</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>&quot;ab&quot;</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>&quot;wb&quot;</td>
</tr>
<tr>
<td>+</td>
<td>&quot;rb&quot;</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>&quot;r+b&quot;</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>&quot;w+b&quot;</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>&quot;a+b&quot;</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>&quot;a+b&quot;</td>
</tr>
</tbody>
</table>

If the open operation succeeds and (mode & ios_base::ate) != 0, positions the file to the end (as if by calling std::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<charT,traits*>* open(const string& s,
ios_base::openmode mode);

Returns: open(s.c_str(), mode);

basic_filebuf<charT,traits*>* close();

334) The macro SEEK_END is defined, and the function signatures fopen(const char*, const char*) and fseek(FILE*, long, int) are declared, in <cstdio> (27.9.2).
Effects: If \texttt{is\_open()} == \texttt{false}, returns a null pointer. If a put area exists, calls \texttt{overflow(traits::eof())} to flush characters. If the last virtual member function called on \texttt{*this} (between \texttt{underflow}, \texttt{overflow}, \texttt{seekoff}, and \texttt{seekpos}) was \texttt{overflow} then calls \texttt{a\_codecvt\_unshift} (possibly several times) to determine a termination sequence, inserts those characters and calls \texttt{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 \texttt{std::fclose(file)}).\footnote{335} If any of the calls made by the function, including \texttt{std::fclose}, fails, \texttt{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: \texttt{this} on success, a null pointer otherwise.

Postcondition: \texttt{is\_open()} == \texttt{false}.

27.9.1.5 Overridden virtual functions

\texttt{streamsize showmanyc();}

\texttt{Effects: Behaves the same as basic\_streambuf::showmanyc() (27.6.3.4).}

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.

\texttt{int\_type underflow();}

\texttt{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 (\texttt{extern\_buf}) and then as if by doing}

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

This shall be done in such a way that the class can recover the position (\texttt{fpos\_t}) corresponding to each character between \texttt{intern\_buf} and \texttt{intern\_end}. If the value of \texttt{r} indicates that \texttt{a\_codecvt\_in()} ran out of space in \texttt{intern\_buf}, retry with a larger \texttt{intern\_buf}.

\texttt{int\_type uflow();}

\texttt{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 \texttt{underflow}.}

\texttt{int\_type pbackfail(int\_type c = traits::eof());}

\texttt{Effects: Puts back the character designated by \texttt{c} to the input sequence, if possible, in one of three ways:}

\begin{itemize}
    \item If \texttt{traits::eq\_int\_type(c,traits::eof())} returns \texttt{false} and if the function makes a putback position available and if \texttt{traits::eq(to\_char\_type(c),gptr()[-1])} returns \texttt{true}, decrements the next pointer for the input sequence, \texttt{gptr()}.
\end{itemize}

Returns: \texttt{c}.

\footnote{335} The function signature \texttt{fclose(FILE*)} is declared in \texttt{<cstdio>} (27.9.2).
— If \texttt{traits::eq\_int\_type(c,traits::eof())} returns \texttt{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 \texttt{c} there.

Returns: \texttt{c}.

— If \texttt{traits::eq\_int\_type(c,traits::eof())} returns \texttt{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, \texttt{gptr()}.

Returns: \texttt{traits::not\_eof(c)}.

6 Returns: \texttt{traits::eof()} to indicate failure.

Remarks: If \texttt{is\_open()} == \texttt{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.

\begin{verbatim}
int_type overflow(int_type c = traits::eof());
\end{verbatim}

Effects: Behaves according to the description of \texttt{basic\_streambuf\langle charT, traits\rangle::overflow(c)}, except that the behavior of “consuming characters” is performed by first converting as if by:

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

and then

\begin{verbatim}
— If \texttt{r == codecvt\_base::error} then fail.
— If \texttt{r == codecvt\_base::noconv} then output characters from \texttt{b} up to (and not including) \texttt{p}.
— If \texttt{r == codecvt\_base::partial} then output to the file characters from \texttt{xbuf} up to \texttt{xbuf\_end}, and repeat using characters from \texttt{end} to \texttt{p}. If output fails, fail (without repeating).
— Otherwise output from \texttt{xbuf} to \texttt{xbuf\_end}, and fail if output fails. At this point if \texttt{b} != \texttt{p} and \texttt{b} == \texttt{end} (\texttt{xbuf} isn’t large enough) then increase \texttt{XSIZE} and repeat from the beginning.
\end{verbatim}

Returns: \texttt{traits::not\_eof(c)} to indicate success, and \texttt{traits::eof()} to indicate failure. If \texttt{is\_open()} == \texttt{false}, the function always fails.

\begin{verbatim}
basic\_streambuf* setbuf(char_type* s, streamsize n);
\end{verbatim}

Effects: If \texttt{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 \texttt{pbase()} and \texttt{pptr()} always return null and output to the file should appear as soon as possible.

\begin{verbatim}
pos_type seekoff(off_type off, ios_base::seekdir way,
    ios_base::openmode which = ios_base::in | ios_base::out);
\end{verbatim}

Effects: Let \texttt{width} denote \texttt{a\_codecvt\_encoding()}. If \texttt{is\_open()} == \texttt{false}, or \texttt{off} != 0 && \texttt{width} <= 0, then the positioning operation fails. Otherwise, if \texttt{way} != \texttt{basic\_ios::cur} or \texttt{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 std::fseek(file, width * off, whence), otherwise call std::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 133.

<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);

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;
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();

Effects: If a put area exists, calls filebuf::overflow to write the characters to the file. If a get area exists, the effect is implementation-defined.

void imbue(const locale& loc);

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 (22.4.1.4.2) 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.

Remark: 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.
27.9.1.6  Class template basic_ifstream

namespace std {
    template <class charT, class traits = char_traits<charT> >
    class basic_ifstream : public basic_istream<charT,traits> {
        public:
            typedef charT char_type;
            typedef typename traits::int_type int_type;
            typedef typename traits::pos_type pos_type;
            typedef typename traits::off_type off_type;
            typedef traits traits_type;

            // 27.9.1.7 Constructors:
            basic_ifstream();
            explicit basic_ifstream(const char* s, 
                ios_base::openmode mode = ios_base::in);
            explicit basic_ifstream(const string& s, 
                ios_base::openmode mode = ios_base::in);
            basic_ifstream(const basic_ifstream& rhs) = delete;
            basic_ifstream(basic_ifstream&& rhs);

            // 27.9.1.8 Assign/swap:
            basic_ifstream& operator=(const basic_ifstream& rhs) = delete;
            basic_ifstream& operator=(basic_ifstream&& rhs);
            void swap(basic_ifstream& rhs);

            // 27.9.1.9 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 string& 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);
}

1 The class basic_ifstream<charT, traits> supports reading from named files. It uses a basic_filebuf<charT, traits> object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

— sb, the filebuf object.

27.9.1.7  basic_ifstream constructors

basic_ifstream();

1  Effects: Constructs an object of class basic_ifstream<charT,traits>, initializing the base class with basic_istream(&sb) and initializing sb with basic_filebuf<charT,traits>() (27.7.2.1.1, 27.9.1.2).
ios_base::openmode mode = ios_base::in);

Effects: Constructs an object of class basic_ifstream, initializing the base class with basic_istream<sb> and initializing sb with basic_filebuf<charT, traits>() (27.7.2.1.1, 27.9.1.2), then calls rdbuf()->open(s, mode | ios_base::in). If that function returns a null pointer, calls setstate(failbit).

explicit basic_ifstream(const string& s,
   ios_base::openmode mode = ios_base::in);

Effects: the same as basic_ifstream(s.c_str(), mode).

basic_ifstream(basic_ifstream&& rhs);

Effects: Move constructs from the rvalue rhs. This is accomplished by move constructing the base class, and the contained basic_filebuf. Next basic_istream<charT, traits>::set_rdbuf(&sb) is called to install the contained basic_filebuf.

27.9.1.8 Assign and swap

basic_ifstream& operator=(basic_ifstream&& rhs);

Effects: Move assigns the base and members of *this from the base and corresponding members of rhs.

Returns: *this.

void swap(basic_ifstream& rhs);

Effects: Exchanges the state of *this and rhs by calling basic_istream<charT, traits>::swap(rhs) and sb.swap(rhs.sb).

template <class charT, class traits>
void swap(basic_ifstream<charT, traits>& x,
   basic_ifstream<charT, traits>& y);

Effects: x.swap(y).

27.9.1.9 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::in);

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 (27.5.5.4)).

void open(const string& 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 (27.5.5.4)).

§ 27.9.1.9
27.9.1.10 Class template basic_ofstream

```cpp
namespace std {
    template <class charT, class traits = char_traits<charT> >
    class basic_ofstream : public basic_ostream<charT,traits> {
    public:
        typedef charT char_type;
        typedef typename traits::int_type int_type;
        typedef typename traits::pos_type pos_type;
        typedef typename traits::off_type off_type;
        typedef traits traits_type;

        // 27.9.1.11 Constructors:
        basic_ofstream();
        explicit basic_ofstream(const char* s,
                                 ios_base::openmode mode = ios_base::out);
        explicit basic_ofstream(const string& s,
                                 ios_base::openmode mode = ios_base::out);
        basic_ofstream(const basic_ofstream& rhs) = delete;
        basic_ofstream(basic_ofstream&& rhs);

        // 27.9.1.12 Assign/swap:
        basic_ofstream& operator=(const basic_ofstream& rhs) = delete;
        basic_ofstream& operator=(basic_ofstream&& rhs);
        void swap(basic_ofstream& rhs);

        // 27.9.1.13 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 string& s, ios_base::openmode mode = ios_base::out);
        void close();
    private:
        basic_filebuf<charT,traits> sb; // exposition only
        basic_ofstream<charT, traits>& x,
        basic_ofstream<charT, traits>& y);
    }

    template <class charT, class traits>
    void swap(basic_ofstream<charT, traits>& x,
              basic_ofstream<charT, traits>& y);
}
```

1 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.

27.9.1.11 basic_ofstream constructors

```cpp
basic_ofstream();
```

1 *Effects*: Constructs an object of class `basic_ofstream<charT,traits>`, initializing the base class with `basic_ostream(&sb)` and initializing `sb` with `basic_filebuf<charT,traits>()` (27.7.3.2, 27.9.1.2).
Effects: Constructs an object of class `basic_ofstream<charT, traits>`, initializing the base class with `basic_ostream<basic_filebuf<charT, traits>>(sb)` and initializing `sb` with `basic_filebuf<charT, traits>()` (27.7.3.2, 27.9.1.2), then calls `rdbuf()->open(s, ios_base::out)` and `setstate(failbit)`.

```
explicit basic_ofstream(const string& 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`.

### 27.9.1.12 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: `x.swap(y)`.

### 27.9.1.13 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);
```

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` (27.5.5.4)).

```
void close();
```

Effects: Calls `rdbuf()->close()` and, if that function fails (returns a null pointer), calls `setstate(failbit)` (which may throw `ios_base::failure` (27.5.5.4)).

```
void open(const string& s, ios_base::openmode mode = ios_base::out);
```

Effects: calls `open(s.c_str(), mode);`
27.9.1.14 Class template basic_fstream

namespace std {
    template <class charT, class traits=char_traits<charT> >
    class basic_fstream
        : public basic_iosstream<charT,traits> {

    public:
        typedef charT char_type;
        typedef typename traits::int_type int_type;
        typedef typename traits::pos_type pos_type;
        typedef typename traits::off_type off_type;
        typedef traits traits_type;

        // constructors/destructor
        basic_fstream();
        explicit basic_fstream(const char* s,
                                ios_base::openmode mode = ios_base::in|ios_base::out);
        explicit basic_fstream(const string& s,
                                ios_base::openmode mode = ios_base::in|ios_base::out);
        basic_fstream(const basic_fstream& rhs) = delete;
        basic_fstream(basic_fstream&& rhs);

        // 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 string& 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:

— sb, the basic_filebuf object.
Effects: Constructs an object of class basic_fstream<charT,traits>, initializing the base class with basic_iosstream(&sb) and initializing sb with basic_filebuf<charT,traits>().

explicit basic_fstream(const char* s, 
ios_base::openmode mode = ios_base::in|ios_base::out);

Effects: Constructs an object of class basic_fstream<charT,traits>, initializing the base class with basic_iosstream(&sb) 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);

Effects: the same as basic_fstream(s.c_str(), mode);

basic_fstream(basic_fstream&& rhs);

Effects: Move constructs from the rvalue rhs. This is accomplished by move constructing the base class, and the contained basic_filebuf. Next basic_istream<charT,traits>::set_rdbuf(&sb) is called to install the contained basic_filebuf.

27.9.1.16 Assign and swap
[fstream.assign]

basic_fstream& operator=(basic_fstream&& rhs);

Effects: Move assigns the base and members of *this from the base and corresponding members of rhs.

Returns: *this.

void swap(basic_fstream& rhs);

Effects: Exchanges the state of *this and rhs by calling basic_iosstream<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);

Effects: x.swap(y).

27.9.1.17 Member functions
[fstream.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|ios_base::out);

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) (27.5.5.4).

void open(const string& 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 returns a null pointer, calls `setstate(failbit)` (27.5.5.4) (which may throw `ios_base::failure`).

27.9.2 C library files

Table 134 describes header `<cstdio>`. [Note: C++ does not define the function `gets`. — end note]

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td></td>
</tr>
<tr>
<td>BUFSIZ</td>
<td>FOPEN_MAX</td>
</tr>
<tr>
<td>EOF</td>
<td>L_tmpnam</td>
</tr>
<tr>
<td>FILENAME_MAX</td>
<td>NULL</td>
</tr>
<tr>
<td>Functions:</td>
<td></td>
</tr>
<tr>
<td>clearerr</td>
<td>fopen</td>
</tr>
<tr>
<td>fclose</td>
<td>fprintf</td>
</tr>
<tr>
<td>feof</td>
<td>fprintf</td>
</tr>
<tr>
<td>ferror</td>
<td>fputs</td>
</tr>
<tr>
<td>fflush</td>
<td>fread</td>
</tr>
<tr>
<td>fgetc</td>
<td>freopen</td>
</tr>
<tr>
<td>fgetpos</td>
<td>fscanf</td>
</tr>
<tr>
<td>fget</td>
<td>fseek</td>
</tr>
</tbody>
</table>

2 Calls to the function `tmpnam` with an argument of `NULL` may introduce a data race (17.6.5.9) with other calls to `tmpnam` with an argument of `NULL`.

See also: ISO C 7.9, Amendment 1 4.6.2.

Table 135 describes header `<cinttypes>`. [Note: The macros defined by `<cinttypes>` are provided unconditionally. In particular, the symbol `__STDC_FORMAT_MACROS`, mentioned in footnote 182 of the C standard, plays no role in C++. — end note]

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td></td>
</tr>
<tr>
<td>PRI{d i o u x X}{FAST LEAST}{8 16 32 64}</td>
<td>imaxdiv_t</td>
</tr>
<tr>
<td>PRI{d i o u x X}{MAX PTR}</td>
<td></td>
</tr>
<tr>
<td>SCN{d i o u x X}{FAST LEAST}{8 16 32 64}</td>
<td></td>
</tr>
<tr>
<td>SCN{d i o u x X}{MAX PTR}</td>
<td></td>
</tr>
<tr>
<td>Types:</td>
<td></td>
</tr>
<tr>
<td>abs</td>
<td>imaxabs</td>
</tr>
<tr>
<td>div</td>
<td>imaxdiv</td>
</tr>
</tbody>
</table>

4 The contents of header `<cinttypes>` are the same as the Standard C Library header `<inttypes.h>`, with the following changes:

---

(4.1) — the header `<cinttypes>` includes the header `<cstddef>` instead of `<cstdint.h>`, and

§ 27.9.2

1093
if and only if the type `intmax_t` designates an extended integer type (3.9.1), 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.
28  Regular expressions library  [re]

28.1 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 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 136.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.2 Definitions</td>
<td></td>
</tr>
<tr>
<td>28.3 Requirements</td>
<td></td>
</tr>
<tr>
<td>28.5 Constants</td>
<td></td>
</tr>
<tr>
<td>28.6 Exception type</td>
<td></td>
</tr>
<tr>
<td>28.7 Traits</td>
<td></td>
</tr>
<tr>
<td>28.8 Regular expression template</td>
<td>&lt;regex&gt;</td>
</tr>
<tr>
<td>28.9 Submatches</td>
<td></td>
</tr>
<tr>
<td>28.10 Match results</td>
<td></td>
</tr>
<tr>
<td>28.11 Algorithms</td>
<td></td>
</tr>
<tr>
<td>28.12 Iterators</td>
<td></td>
</tr>
<tr>
<td>28.13 Grammar</td>
<td></td>
</tr>
</tbody>
</table>

28.2 Definitions  [re.def]
1  The following definitions shall apply to this Clause:

28.2.1  [defns.regex.collating.element]
collating element
a sequence of one or more characters within the current locale that collate as if they were a single character.

28.2.2  [defns.regex.finite.state.machine]
finite state machine
an unspecified data structure that is used to represent a regular expression, and which permits efficient matches against the regular expression to be obtained.

28.2.3  [defns.regex.format.specifier]
format specifier
a sequence of one or more characters that is to be replaced with some part of a regular expression match.

28.2.4  [defns.regex.matched]
matched
§ 28.2
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.

28.2.5 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.

28.2.6 regular expression

A pattern that selects specific strings from a set of character strings.

28.2.7 sub-expression

A subset of a regular expression that has been marked by parenthesis.

28.3 Requirements

1. This subclause defines requirements on classes representing regular expression traits. [Note: The class template \texttt{regex\_traits}, defined in Clause 28.7, satisfies these requirements. —end note]

2. The class template \texttt{basic\_regex}, defined in Clause 28.8, 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 typedefs 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>}.\n
4. In Table 137 \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 (24.2.3); \texttt{F1} and \texttt{F2} are forward iterators (24.2.5); \texttt{c} is a value of type \texttt{const charT}; \texttt{s} is an object of type \texttt{X::string\_type}; \texttt{cs} is an object of type \texttt{const X::string\_type}; \texttt{b} is a value of type \texttt{bool}; \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 137 — Regular expression traits class requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>\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{std::basic_-}</td>
<td>A copy constructible type</td>
</tr>
<tr>
<td></td>
<td>\texttt{string&lt;CharT&gt;}</td>
<td>A type that represents the locale used by the traits class.</td>
</tr>
<tr>
<td>\texttt{X::locale_type}</td>
<td>A bitmask type (17.5.2.1.3).</td>
<td>A bitmask type representing a particular character classification.</td>
</tr>
<tr>
<td>\texttt{X::char_class_type}</td>
<td>\texttt{std::size_t}</td>
<td>Yields the smallest ( i ) such that ( p[i] == 0 ). Complexity is linear in ( i ).</td>
</tr>
</tbody>
</table>

§ 28.3
Table 137 — Regular expression traits class requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>v.translate(c)</code></td>
<td><code>X::char_type</code></td>
<td>Returns a character such that for any character <code>d</code> that is to be considered equivalent to <code>c</code> then <code>v.translate(c) == v.translate(d)</code>.</td>
</tr>
<tr>
<td><code>v.translate_nocase(c)</code></td>
<td><code>X::char_type</code></td>
<td>For all characters <code>C</code> that are to be considered equivalent to <code>c</code> when comparisons are to be performed without regard to case, then <code>v.translate_nocase(c) == v.translate_nocase(C)</code>.</td>
</tr>
<tr>
<td><code>v.transform(F1, F2)</code></td>
<td><code>X::string_type</code></td>
<td>Returns a sort key for the character sequence designated by the iterator range <code>[F1,F2)</code> such that if the character sequence <code>[G1,G2)</code> sorts before the character sequence <code>[H1,H2)</code> then <code>v.transform(G1, G2) &lt; v.transform(H1, H2)</code>.</td>
</tr>
<tr>
<td><code>v.transform_primary(F1, F2)</code></td>
<td><code>X::string_type</code></td>
<td>Returns a sort key for the character sequence designated by the iterator range <code>[F1,F2)</code> such that if the character sequence <code>[G1,G2)</code> sorts before the character sequence <code>[H1,H2)</code> when character case is not considered then <code>v.transform_primary(G1, G2) &lt; v.transform_primary(H1, H2)</code>.</td>
</tr>
<tr>
<td><code>v.lookup_collatename(F1, F2)</code></td>
<td><code>X::string_type</code></td>
<td>Returns a sequence of characters that represents the collating element consisting of the character sequence designated by the iterator range <code>[F1,F2)</code>. Returns an empty string if the character sequence is not a valid collating element.</td>
</tr>
<tr>
<td><code>v.lookup_classname(F1, F2, b)</code></td>
<td><code>X::char_class_type</code></td>
<td>Converts the character sequence designated by the iterator range <code>[F1,F2)</code> into a value of a bitmask type that can subsequently be passed to <code>isctype</code>. Values returned from <code>lookup_classname</code> can be bitwise or’ed together; the resulting value represents membership in either of the corresponding character classes. If <code>b</code> is true, the returned bitmask is suitable for matching characters without regard to their case. Returns 0 if the character sequence is not the name of a character class recognized by <code>X</code>. The value returned shall be independent of the case of the characters in the sequence.</td>
</tr>
<tr>
<td><code>v.isctype(c, cl)</code></td>
<td><code>bool</code></td>
<td>Returns <code>true</code> if character <code>c</code> is a member of one of the character classes designated by <code>cl</code>, <code>false</code> otherwise.</td>
</tr>
</tbody>
</table>
Table 137 — Regular expression traits class requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>v.value(c, I)</td>
<td>int</td>
<td>Returns the value represented by the digit c in base I if the character c is a valid digit in base I; otherwise returns -1. [Note: The value of I will only be 8, 10, or 16. —end note]</td>
</tr>
<tr>
<td>u.imbue(loc)</td>
<td>X::locale_type</td>
<td>Imbues u with the locale loc and returns the previous locale used by u if any.</td>
</tr>
<tr>
<td>v.getloc()</td>
<td>X::locale_type</td>
<td>Returns the current locale used by v, if any.</td>
</tr>
</tbody>
</table>

5 [Note: Class template regex_traits satisfies the requirements for a regular expression traits class when it is specialized for char or wchar_t. This class template is described in the header <regex>, and is described in Clause 28.7. —end note]

### 28.4 Header <regex> synopsis

```cpp
#include <initializer_list>

namespace std {

// 28.5, regex constants:
namespace regex_constants {
    enum error_type;
} // namespace regex_constants

// 28.6, class regex_error:
class regex_error;

// 28.7, class template regex_traits:
template <class charT> struct regex_traits;

// 28.8, class template basic_regex:
template <class charT, class traits = regex_traits<charT> > class basic_regex;
typedef basic_regex<char> regex;
typedef basic_regex<wchar_t> wregex;

// 28.8.6, basic_regex swap:
template <class charT, class traits>
    void swap(basic_regex<charT, traits>& e1, basic_regex<charT, traits>& e2);

// 28.9, class template sub_match:
template <class BidirectionalIterator>
    class sub_match;
typedef sub_match<const char*> csub_match;
typedef sub_match<const wchar_t*> wcsub_match;
typedef sub_match<string::const_iterator> sssub_match;
typedef sub_match<wstring::const_iterator> wssub_match;

// 28.9.2, sub_match non-member operators:
```

§ 28.4
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);

template <class BiIter, class ST, class SA>
bool operator==
    (const sub_match<BiIter>& lhs,
    const basic_string<
        typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
template <class BiIter, class ST, class SA>
bool operator!=
    (const sub_match<BiIter>& lhs,
    const basic_string<
        typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
template <class BiIter, class ST, class SA>
bool operator<(
    const sub_match<BiIter>& lhs,
    const basic_string<
        typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template <class BiIter, class ST, class SA>
bool operator>(
    const sub_match<BiIter>& lhs,
    const basic_string<
        typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template <class BiIter, class ST, class SA>
bool operator>=(
    const sub_match<BiIter>& lhs,
    const basic_string<
        typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template <class BiIter, class ST, class SA>
bool operator<=(
    const sub_match<BiIter>& lhs,
    const basic_string<
        typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template <class BiIter, class ST, class SA>
bool operator==(typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter, class ST, class SA>
bool operator!=(typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter, class ST, class SA>
bool operator<(typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter, class ST, class SA>
bool operator>(typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter, class ST, class SA>
bool operator>=(typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter, class ST, class SA>
bool operator<=(typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

§ 28.4 1100
template <class BiIter>
bool operator<=(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const* rhs);

template <class BiIter>
bool operator==(typename iterator_traits<BiIter>::value_type const& lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter>
bool operator!=(typename iterator_traits<BiIter>::value_type const& lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter>
bool operator<(typename iterator_traits<BiIter>::value_type const& lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter>
bool operator>(typename iterator_traits<BiIter>::value_type const& lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter>
bool operator>=(typename iterator_traits<BiIter>::value_type const& lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter>
bool operator==(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

template <class BiIter>
bool operator!=(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

template <class BiIter>
bool operator<(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

template <class BiIter>
bool operator>(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

template <class BiIter>
bool operator>=(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

template <class BiIter>
bool operator<=(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

template <class BiIter>
bool operator!=(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

template <class BiIter>
bool operator<(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

template <class BiIter>
bool operator>(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

template <class BiIter>
bool operator>=(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

template <class charT, class ST, class BiIter>
basic_ostream<charT, ST>&
    operator<<((basic_ostream<charT, ST>& os, const sub_match<BiIter>& m);

// 28.10, class template match_results:
template <class BidirectionalIterator, 
    class Allocator = allocator<sub_match<BidirectionalIterator> > >
    class match_results;

typedef match_results<const char*> cmatch;
typedef match_results<const wchar_t*> wcmatch;
typedef match_results<string::const_iterator> smatch;
typedef match_results<wstring::const_iterator> wsmatch;
// match_results comparisons
template <class BidirectionalIterator, class Allocator>
bool operator== (const match_results<BidirectionalIterator, Allocator>& m1,
                 const match_results<BidirectionalIterator, Allocator>& m2);

template <class BidirectionalIterator, class Allocator>
bool operator!= (const match_results<BidirectionalIterator, Allocator>& m1,
                 const match_results<BidirectionalIterator, Allocator>& m2);

// 28.10.7, match_results swap:
template <class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1,
          match_results<BidirectionalIterator, Allocator>& m2);

// 28.11.2, function template regex_match:
template <class BidirectionalIterator, class Allocator,
          class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 match_results<BidirectionalIterator, Allocator>& m,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags =
                 regex_constants::match_default);

template <class BidirectionalIterator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags =
                 regex_constants::match_default);

template <class charT, class Allocator, class traits>
bool regex_match(const charT* str, match_results<const charT*, Allocator>& m,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags =
                 regex_constants::match_default);

template <class ST, class SA, class Allocator, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
                 match_results<typename basic_string<charT, ST, SA>::const_iterator,
                 Allocator>& m,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags =
                 regex_constants::match_default);

template <class ST, class SA, class Allocator, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>&&,
                 match_results<typename basic_string<charT, ST, SA>::const_iterator,
                 Allocator>&,
                 const basic_regex<charT, traits>&,
                 regex_constants::match_flag_type =
                 regex_constants::match_default) = delete;

template <class charT, class traits>
bool regex_match(const charT* str,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags =
                 regex_constants::match_default);

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);

// 28.11.3, function template regex_search:
template <class BidirectionalIterator, class Allocator,
    class charT, class traits>
    bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                       match_results<BidirectionalIterator, Allocator>& m,
                       const basic_regex<charT, traits>& e,
                       regex_constants::match_flag_type flags =
                       regex_constants::match_default);

template <class BidirectionalIterator, class charT, class traits>
    bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                       const basic_regex<charT, traits>& e,
                       regex_constants::match_flag_type flags =
                       regex_constants::match_default);

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);

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);

template <class ST, class SA, class charT, class traits>
    bool regex_search(const basic_string<charT, ST, SA>& s,
                       const basic_regex<charT, traits>& e,
                       regex_constants::match_flag_type flags =
                       regex_constants::match_default);

template <class ST, class SA, class Allocator, class charT, class traits>
    bool regex_search(const basic_string<charT, ST, SA>&& s,
                       match_results<
                       typename basic_string<charT, ST, SA>::const_iterator,
                       Allocator>& m,
                       const basic_regex<charT, traits>& e,
                       regex_constants::match_flag_type flags =
                       regex_constants::match_default);

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 =
                       regex_constants::match_default) = delete;

// 28.11.4, function template regex_replace:
template <class OutputIterator, class BidirectionalIterator,
    class traits, class charT, class ST, class SA>
    OutputIterator
    regex_replace(OutputIterator out,
BidirectionalIterator first, BidirectionalIterator last,
const basic_regex<charT, traits>& e,
const basic_string<charT, ST, SA>& fmt,
regex_constants::match_flag_type flags =
regex_constants::match_default);

template <class OutputIterator, class BidirectionalIterator,
class traits, class charT>
OutputIterator
regex_replace(OutputIterator out,
BidirectionalIterator first, BidirectionalIterator last,
const basic_regex<charT, traits>& e,
const charT* fmt,
regex_constants::match_flag_type flags =
regex_constants::match_default);

template <class traits, class charT, class ST, class SA,
class FST, class FSA>
basic_string<charT, ST, SA>
regex_replace(const basic_string<charT, ST, SA>& s,
const basic_regex<charT, traits>& e,
const basic_string<charT, FST, FSA>& fmt,
regex_constants::match_flag_type flags =
regex_constants::match_default);

template <class traits, class charT, class ST, class SA>
basic_string<charT, ST, SA>
regex_replace(const basic_string<charT, ST, SA>& s,
const basic_regex<charT, traits>& e,
const charT* fmt,
regex_constants::match_flag_type flags =
regex_constants::match_default);

template <class traits, class charT, 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);

// 28.12.1, class template regex_iterator:
template <class BidirectionalIterator,
class charT = typename iterator_traits<BidirectionalIterator>::value_type,
class traits = regex_traits<charT> >
class regex_iterator;

typedef regex_iterator<const char*> cregex_iterator;
typedef regex_iterator<const wchar_t*> wcregex_iterator;
typedef regex_iterator<string::const_iterator> sregex_iterator;
typedef regex_iterator<wstring::const_iterator> wsregex_iterator;
28.12.2, class template regex_token_iterator:

```cpp
// 28.12.2, class template regex_token_iterator:
template <class BidirectionalIterator,
    class charT = typename iterator_traits<BidirectionalIterator>::value_type,
    class traits = regex_traits<charT> >
class regex_token_iterator;
```

typedef regex_token_iterator<const char*> cregex_token_iterator;
typedef regex_token_iterator<const wchar_t*> wcregex_token_iterator;
typedef regex_token_iterator<string::const_iterator> sregex_token_iterator;
typedef regex_token_iterator<wstring::const_iterator> wsregex_token_iterator;

28.5 Namespace std::regex_constants

The namespace `std::regex_constants` holds symbolic constants used by the regular expression library. This namespace provides three types, `syntax_option_type`, `match_flag_type`, and `error_type`, along with several constants of these types.

28.5.1 Bitmask type syntax_option_type

```cpp
namespace std::regex_constants {
    typedef T1 syntax_option_type;
    constexpr syntax_option_type icase = unspecified;
    constexpr syntax_option_type nosubs = unspecified;
    constexpr syntax_option_type optimize = unspecified;
    constexpr syntax_option_type collate = unspecified;
    constexpr syntax_option_type ECMAScript = unspecified;
    constexpr syntax_option_type basic = unspecified;
    constexpr syntax_option_type extended = unspecified;
    constexpr syntax_option_type awk = unspecified;
    constexpr syntax_option_type grep = unspecified;
    constexpr syntax_option_type egrep = unspecified;
}
```

1 The type `syntax_option_type` is an implementation-defined bitmask type (17.5.2.1.3). Setting its elements has the effects listed in table 138. 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.

28.5.2 Bitmask type regex_constants::match_flag_type

```cpp
namespace std::regex_constants {
    typedef T2 match_flag_type;
    constexpr match_flag_type match_default = {};
    constexpr match_flag_type match_not_bol = unspecified;
    constexpr match_flag_type match_not_eol = unspecified;
    constexpr match_flag_type match_not_bow = unspecified;
    constexpr match_flag_type match_not_eow = unspecified;
    constexpr match_flag_type match_any = unspecified;
    constexpr match_flag_type match_not_null = unspecified;
    constexpr match_flag_type match_continuous = unspecified;
    constexpr match_flag_type match_prev_avail = unspecified;
    constexpr match_flag_type match_format_default = {};
    constexpr match_flag_type match_format_sed = unspecified;
    constexpr match_flag_type match_format_no_copy = unspecified;
}
```

§ 28.5.2
Table 138 — 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 28.13.</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>
</tbody>
</table>

\[
\text{constexpr match_flag_type format_first_only = unspecified ;}
\]

1 The type regex_constants::match_flag_type is an implementation-defined bitmask type (17.5.2.1.3). 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 \([\text{first}, \text{last})\) proceeds according to the rules of the grammar specified for the regular expression object, modified according to the effects listed in Table 139 for any bitmask elements set.

Table 139 — regex_constants::match_flag_type effects when obtaining a match against a character container sequence \([\text{first}, \text{last})\).

<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 ([\text{first}, \text{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 ([\text{first}, \text{first})).</td>
</tr>
<tr>
<td>match_not_eol</td>
<td>The last character in the sequence ([\text{first}, \text{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 ([\text{last}, \text{last})).</td>
</tr>
</tbody>
</table>
Table 139 — `regex_constants::match_flag_type` effects when obtaining a match against a character container sequence \([\text{first, last})\). (continued)

<table>
<thead>
<tr>
<th>Element</th>
<th>Effect(s) if set</th>
</tr>
</thead>
<tbody>
<tr>
<td>match_not_bow</td>
<td>The expression &quot;\b&quot; shall not match the sub-sequence ([\text{first,first})).</td>
</tr>
<tr>
<td>match_not_eow</td>
<td>The expression &quot;\b&quot; shall not match the sub-sequence ([\text{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 (\text{first}).</td>
</tr>
<tr>
<td>match_prev_avail</td>
<td>(\text{--first}) is a valid iterator position. When this flag is set the flags match_\text{-not_bol} and match_\text{-not_bow} shall be ignored by the regular expression algorithms 28.11 and iterators 28.12.</td>
</tr>
<tr>
<td>format_default</td>
<td>When a regular expression match is to be replaced by a new string, the new string shall be constructed using the rules used by the ECMAScript replace function in ECMA-262, part 15.5.4.11 String.prototype.replace. In addition, during search and replace operations all non-overlapping occurrences of the regular expression shall be located and replaced, and sections of the input that did not match the expression shall be copied unchanged to the output string.</td>
</tr>
<tr>
<td>format_sed</td>
<td>When a regular expression match is to be replaced by a new string, the new string shall be constructed using the rules used by the sed utility in POSIX.</td>
</tr>
<tr>
<td>format_no_copy</td>
<td>During a search and replace operation, sections of the character container sequence being searched that do not match the regular expression shall not be copied to the output string.</td>
</tr>
<tr>
<td>format_first_only</td>
<td>When specified during a search and replace operation, only the first occurrence of the regular expression shall be replaced.</td>
</tr>
</tbody>
</table>

28.5.3 Implementation-defined `error_type`

```cpp
namespace std::regex_constants {
    typedef T3 error_type;
    constexpr error_type error_collate = unspecified;
    constexpr error_type error_ctype = unspecified;
    constexpr error_type error_escape = unspecified;
    constexpr error_type error_backref = unspecified;
    constexpr error_type error_brack = unspecified;
    constexpr error_type error_paren = unspecified;
    constexpr error_type error_brace = unspecified;
    constexpr error_type error_badbrace = unspecified;
    constexpr error_type error_range = unspecified;
    constexpr error_type error_space = unspecified;
    constexpr error_type error_badrepeat = unspecified;
    constexpr error_type error_complexity = unspecified;
    constexpr error_type error_stack = unspecified;
}
```

1 The type `error_type` is an implementation-defined enumerated type (17.5.2.1.2). Values of type `error_type` represent the error conditions described in Table 140:

§ 28.5.3
Table 140 — error_type values in the C locale

<table>
<thead>
<tr>
<th>Value</th>
<th>Error condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>error_collate</td>
<td>The expression contained an invalid collating element name.</td>
</tr>
<tr>
<td>error_ctype</td>
<td>The expression contained an invalid character class name.</td>
</tr>
<tr>
<td>error_escape</td>
<td>The expression contained an invalid escaped character, or a trailing escape.</td>
</tr>
<tr>
<td>error_backref</td>
<td>The expression contained an invalid back reference.</td>
</tr>
<tr>
<td>error_brack</td>
<td>The expression contained mismatched [ and ].</td>
</tr>
<tr>
<td>error_paren</td>
<td>The expression contained mismatched ( and ).</td>
</tr>
<tr>
<td>error_brace</td>
<td>The expression contained mismatched { and }</td>
</tr>
<tr>
<td>error_badbrace</td>
<td>The expression contained an invalid range in a {} expression.</td>
</tr>
<tr>
<td>error_range</td>
<td>The expression contained an invalid character range, such as [b-a] in most encodings.</td>
</tr>
<tr>
<td>error_space</td>
<td>There was insufficient memory to convert the expression into a finite state machine.</td>
</tr>
<tr>
<td>error_badrepeat</td>
<td>One of *+{ was not preceded by a valid regular expression.</td>
</tr>
<tr>
<td>error_complexity</td>
<td>The complexity of an attempted match against a regular expression exceeded a pre-set level.</td>
</tr>
<tr>
<td>error_stack</td>
<td>There was insufficient memory to determine whether the regular expression could match the specified character sequence.</td>
</tr>
</tbody>
</table>

28.6 Class regex_error

```cpp
class regex_error : public std::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 Postcondition: ecode == code()

```cpp
regex_constants::error_type code() const;
```

4 Returns: The error code that was passed to the constructor.

28.7 Class template regex_traits

```cpp
namespace std {
  template <class charT>
  struct regex_traits {
    public:
      typedef charT char_type;
      typedef std::basic_string<char_type> string_type;
      typedef std::locale locale_type;
      typedef bitmask_type char_class_type;

      regex_traits();
      static std::size_t length(const char_type* p);
  };
}
```

§ 28.7
The specializations regex_traits<char> and regex_traits<wchar_t> shall be valid and shall satisfy the requirements for a regular expression traits class (28.3).

typedef bitmask_type char_class_type;

The type char_class_type is used to represent a character classification and is capable of holding an implementation specific set returned by lookup_classname.

static std::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:

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 \texttt{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. The value returned shall be independent of the case of the characters in the character sequence. If the name is not recognized then returns \texttt{char_class_type(\_).}

Remarks: For \texttt{regex_traits<char>}, at least the narrow character names in Table 141 shall be recognized. For \texttt{regex_traits<wchar_t>}, at least the wide character names in Table 141 shall be recognized.

```
bool isctype(charT c, char_class_type f) const;
```

Effects: Determines if the character \(c\) is a member of the character classification represented by \(f\).

Returns: Given the following function declaration:

```
// for exposition only
template<class C>
ctype_base::mask convert(typename regex_traits<C>::char_class_type f);
```

that returns a value in which each \texttt{ctype_base::mask} value corresponding to a value in \(f\) named in Table 141 is set, then the result is determined as if by:

```
cctype_base::mask m = convert<charT>(f);
const ctype<charT>& ct = use_facet<ctype<charT>>(getloc());
if (ct.is(m, c)) {
    return true;
} else if (c == ct.widen('_')) {
    charT w[1] = { ct.widen('w') };  
    char_class_type x = lookup_classname(w, w+1);
    return (f\&x) == x;
} else {
    return false;
}
```

\[\text{Example:}\]

```
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());
cctype_base::mask m = convert<charT>(f); // m == cctype_base::digit | cctype_base::upper
```

\[\text{Example:}\]

\[336\) For example, if the parameter \texttt{icase} is true then \texttt{[:lower:]} is the same as \texttt{[:alpha:]}.\]
regex_traits<char> t;
string w("w");
regex_traits<char>::char_class_type f;
f = t.lookup_classname(w.begin(), w.end());
t.isctype('A', f); // returns true
f = t.lookup_classname(w.begin(), w.end());
t.isctype('_', f); // returns true
f = t.lookup_classname(w.begin(), w.end());
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.

Postcondition: 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.

Table 141 — Character class names and corresponding ctype masks

<table>
<thead>
<tr>
<th>Narrow character name</th>
<th>Wide character name</th>
<th>Corresponding ctype_base::mask value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;alnum&quot;</td>
<td>L&quot;alnum&quot;</td>
<td>ctype_base::alnum</td>
</tr>
<tr>
<td>&quot;alpha&quot;</td>
<td>L&quot;alpha&quot;</td>
<td>ctype_base::alpha</td>
</tr>
<tr>
<td>&quot;blank&quot;</td>
<td>L&quot;blank&quot;</td>
<td>ctype_base::blank</td>
</tr>
<tr>
<td>&quot;cntrl&quot;</td>
<td>L&quot;cntrl&quot;</td>
<td>ctype_base::cntrl</td>
</tr>
<tr>
<td>&quot;digit&quot;</td>
<td>L&quot;digit&quot;</td>
<td>ctype_base::digit</td>
</tr>
<tr>
<td>&quot;d&quot;</td>
<td>L&quot;d&quot;</td>
<td>ctype_base::digit</td>
</tr>
<tr>
<td>&quot;graph&quot;</td>
<td>L&quot;graph&quot;</td>
<td>ctype_base::graph</td>
</tr>
<tr>
<td>&quot;lower&quot;</td>
<td>L&quot;lower&quot;</td>
<td>ctype_base::lower</td>
</tr>
<tr>
<td>&quot;print&quot;</td>
<td>L&quot;print&quot;</td>
<td>ctype_base::print</td>
</tr>
<tr>
<td>&quot;punct&quot;</td>
<td>L&quot;punct&quot;</td>
<td>ctype_base::punct</td>
</tr>
<tr>
<td>&quot;space&quot;</td>
<td>L&quot;space&quot;</td>
<td>ctype_base::space</td>
</tr>
<tr>
<td>&quot;s&quot;</td>
<td>L&quot;s&quot;</td>
<td>ctype_base::space</td>
</tr>
<tr>
<td>&quot;upper&quot;</td>
<td>L&quot;upper&quot;</td>
<td>ctype_base::upper</td>
</tr>
<tr>
<td>&quot;u&quot;</td>
<td>L&quot;u&quot;</td>
<td>ctype_base::alnum</td>
</tr>
<tr>
<td>&quot;xdigit&quot;</td>
<td>L&quot;xdigit&quot;</td>
<td>ctype_base::xdigit</td>
</tr>
</tbody>
</table>

§ 28.7 1111
28.8 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 28.8, 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.

namespace std {
    template <class charT,
             class traits = regex_traits<charT> >
    class basic_regex {
    public:
       // types:
       typedef charT value_type;
       typedef traits traits_type;
       typedef typename traits::string_type string_type;
       typedef regex_constants::syntax_option_type flag_type;
       typedef typename traits::locale_type locale_type;

       // 28.8.1, 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;

       // 28.8.2, 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);
    }
}
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 = regex_constants::ECMAScript);

~basic_regex();

basic_regex& operator=(const basic_regex&);

basic_regex& operator=(basic_regex&&) noexcept;

basic_regex& operator=(const charT* ptr);

basic_regex& operator=(initializer_list<charT> il);

template <class ST, class SA>
    basic_regex& operator=(const basic_string<charT, ST, SA>& p);

    // 28.8.3 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);
    basic_regex& assign(const charT* p, size_t len, flag_type f);

template <class string_traits, class A>
    basic_regex& assign(const basic_string<charT, string_traits, A>& s,
        flag_type f = regex_constants::ECMAScript);

template <class InputIterator>
    basic_regex& assign(InputIterator first, InputIterator last,
        flag_type f = regex_constants::ECMAScript);

    // 28.8.4 const operations:
    unsigned mark_count() const;
    flag_type flags() const;

    // 28.8.5 locale:
    locale_type imbue(locale_type loc);
    locale_type getloc() const;

    // 28.8.6 swap:
    void swap(basic_regex&);
};

28.8.1 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;
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;

1 The static constant members are provided as synonyms for the constants declared in namespace regex_constants.

28.8.2 basic_regex constructors

\[re.regex.construct\]

basic_regex();
  Effects: Constructs an object of class basic_regex that does not match any character sequence.

explicit basic_regex(const charT* p, flag_type f = regex_constants::ECMAScript);
  Requires: p shall not be a null pointer.
  Throws: regex_error if p is not a valid regular expression.
  Effects: Constructs an object of class basic_regex; the object’s internal finite state machine is constructed from the regular expression contained in the array of charT of length char_traits<charT>::length(p) whose first element is designated by p, and interpreted according to the flags f.
  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);
14 \textit{Throws:} \texttt{regex\_error} if \texttt{s} is not a valid regular expression.

15 \textit{Effects:} Constructs an object of class \texttt{basic\_regex}; the object’s internal finite state machine is constructed from the regular expression contained in the string \texttt{s}, and interpreted according to the flags specified in \texttt{f}.

16 \textit{Postconditions:} \texttt{flags()} returns \texttt{f}. \texttt{mark\_count()} returns the number of marked sub-expressions within the expression.

\begin{verbatim}
template <class ForwardIterator>
basic_regex(ForwardIterator first, ForwardIterator last,
    flag_type f = regex_constants::ECMAScript);
\end{verbatim}

17 \textit{Throws:} \texttt{regex\_error} if the sequence \texttt{[first,last)} is not a valid regular expression.

18 \textit{Effects:} Constructs an object of class \texttt{basic\_regex}; the object’s internal finite state machine is constructed from the regular expression contained in the sequence of characters \texttt{[first,last)}, and interpreted according to the flags specified in \texttt{f}.

19 \textit{Postconditions:} \texttt{flags()} returns \texttt{f}. \texttt{mark\_count()} returns the number of marked sub-expressions within the expression.

\begin{verbatim}
basic_regex(initializer\_list<charT> il,
    flag_type f = regex_constants::ECMAScript);
\end{verbatim}

20 \textit{Effects:} Same as \texttt{basic\_regex(il.begin(), il.end(), f)}.

\section*{28.8.3 \texttt{basic\_regex assign}}

\begin{verbatim}
basic_regex\& operator=(const basic_regex\& e); \hfill [re.regex.assign]
\end{verbatim}

1 \textit{Effects:} returns \texttt{assign(e)}.

\begin{verbatim}
basic_regex\& operator=(basic_regex\& e) noexcept;
\end{verbatim}

2 \textit{Effects:} returns \texttt{assign(std::move(e))}.

\begin{verbatim}
basic_regex\& operator=(const charT* ptr);
\end{verbatim}

3 \textit{Requires:} \texttt{ptr} shall not be a null pointer.

4 \textit{Effects:} returns \texttt{assign(ptr)}.

\begin{verbatim}
basic_regex\& operator=(initializer\_list<charT> il);
\end{verbatim}

5 \textit{Effects:} returns \texttt{assign(il.begin(), il.end())}.

\begin{verbatim}
template <class ST, class SA>
basic_regex\& operator=(const basic_string<charT, ST, SA>& p);
\end{verbatim}

6 \textit{Effects:} returns \texttt{assign(p)}.

\begin{verbatim}
basic_regex\& assign(const basic_regex\& that);
\end{verbatim}

7 \textit{Effects:} copies \texttt{that} into \texttt{*this} and returns \texttt{*this}.

8 \textit{Postconditions:} \texttt{flags()} and \texttt{mark\_count()} return \texttt{that.flags()} and \texttt{that.mark\_count()}, respectively.

\begin{verbatim}
basic_regex\& assign(basic_regex\& that) noexcept;
\end{verbatim}
Effects: move assigns from that into *this and returns *this.

Postconditions: flags() and mark_count() return the values that that.flags() and that.mark_count(), respectively, had before assignment. that is in a valid state with unspecified value.

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 (24.2.3).

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.

28.8.4 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.

28.8.5 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.
28.8.6 basic_regex swap

void swap(basic_regex& e);

Effects: Swaps the contents of the two regular expressions.

Postcondition: *this contains the regular expression that was in e, e contains the regular expression that was in *this.

Complexity: Constant time.

28.8.7 basic_regex non-member functions

28.8.7.1 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).

28.9 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 std::pair<BidirectionalIterator, BidirectionalIterator> {
        public:
            typedef typename iterator_traits<BidirectionalIterator>::
                value_type value_type;
            typedef typename iterator_traits<BidirectionalIterator>::
                difference_type difference_type;
            typedef BidirectionalIterator iterator;
            typedef basic_string<value_type> string_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;
    };
}

28.9.1 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;
3 \textit{Returns:} matched ? string\_type(first, second) : string\_type().

string\_type str() const;

4 \textit{Returns:} matched ? string\_type(first, second) : string\_type().

int compare(const sub\_match\& s) const;

\textit{Returns:} str().compare(s.str()).

5 int compare(const string\_type\& s) const;

\textit{Returns:} str().compare(s).

int compare(const value\_type* s) const;

\textit{Returns:} str().compare(s).

28.9.2 \textbf{sub\_match} non-member operators

\textbf{[re.submatch.op]}

\textbf{template <class BiIter>}
bool operator==(const sub\_match\& l, const sub\_match\& r);

\textit{Returns:} l.compare(r) == 0.

\textbf{template <class BiIter>}
bool operator!=(const sub\_match\& l, const sub\_match\& r);

\textit{Returns:} l.compare(r) != 0.

\textbf{template <class BiIter>}
bool operator<(const sub\_match\& l, const sub\_match\& r);

\textit{Returns:} l.compare(r) < 0.

\textbf{template <class BiIter>}
bool operator<=(const sub\_match\& l, const sub\_match\& r);

\textit{Returns:} l.compare(r) <= 0.

\textbf{template <class BiIter>}
bool operator>(const sub\_match\& l, const sub\_match\& r);

\textit{Returns:} l.compare(r) > 0.

\textbf{template <class BiIter>}
bool operator>=(const sub\_match\& l, const sub\_match\& r);

\textit{Returns:} l.compare(r) >= 0.

\textbf{template <class BiIter, class ST, class SA>}
bool operator==(const basic\_string<typename iterator\_traits<BiIter>::value\_type, ST, SA>& l, const sub\_match<BiIter>& r);

\textit{Returns:} rhs.compare(typename sub\_match<BiIter>::string\_type(l.data(), l.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);

>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);

>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);

>Returns: rhs < lhs.

template <class BiIter, class ST, class SA>
bool operator==(const sub_match<BiIter>& lhs,
    const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

>Returns: lhs.compare(typename sub_match<BiIter>::string_type(rhs.data(), rhs.size()))
    == 0.

template <class BiIter, class ST, class SA>
bool operator!=(const sub_match<BiIter>& lhs,
    const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

>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);

Returns: rhs < lhs.

template <class BiIter, class ST, class SA>
bool operator>=(const sub_match<BiIter>& lhs, const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

Returns: !(lhs < 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);

Returns: !(rhs < lhs).

template <class BiIter>
bool operator==(typename iterator_traits<BiIter>::value_type const* lhs, const sub_match<BiIter>& rhs);

Returns: rhs.compare(lhs) == 0.

template <class BiIter>
bool operator!=(typename iterator_traits<BiIter>::value_type const* lhs, const sub_match<BiIter>& rhs);

Returns: !(lhs == rhs).

template <class BiIter>
bool operator<(typename iterator_traits<BiIter>::value_type const* lhs, const sub_match<BiIter>& rhs);

Returns: rhs < lhs.

template <class BiIter>
bool operator>=(typename iterator_traits<BiIter>::value_type const* lhs, const sub_match<BiIter>& rhs);

Returns: !(lhs < rhs).

template <class BiIter>
bool operator<=(typename iterator_traits<BiIter>::value_type const* lhs, const sub_match<BiIter>& rhs);

Returns: !(rhs < lhs).
template <class BiIter>
bool operator==(const sub_match<BiIter>& lhs,
        typename iterator_traits<BiIter>::value_type const* rhs);

Returns: lhs.compare(rhs) == 0.

template <class BiIter>
bool operator!=(const sub_match<BiIter>& lhs,
        typename iterator_traits<BiIter>::value_type const* rhs);

Returns: !(lhs == rhs).

template <class BiIter>
bool operator<(const sub_match<BiIter>& lhs,
        typename iterator_traits<BiIter>::value_type const* rhs);

Returns: lhs.compare(rhs) < 0.

template <class BiIter>
bool operator>(const sub_match<BiIter>& lhs,
        typename iterator_traits<BiIter>::value_type const* rhs);

Returns: rhs < lhs.

template <class BiIter>
bool operator<=(const sub_match<BiIter>& lhs,
        typename iterator_traits<BiIter>::value_type const* rhs);

Returns: !(lhs < rhs).

template <class BiIter>
bool operator>=(const sub_match<BiIter>& lhs,
        typename iterator_traits<BiIter>::value_type const* rhs);

Returns: !(rhs < lhs).

template <class BiIter>
bool operator==(typename iterator_traits<BiIter>::value_type const& lhs,
        const sub_match<BiIter>& rhs);

Returns: rhs.compare(typename sub_match<BiIter>::string_type(1, lhs)) == 0.

template <class BiIter>
bool operator!=(typename iterator_traits<BiIter>::value_type const& lhs,
        const sub_match<BiIter>& rhs);

Returns: !(lhs == rhs).

template <class BiIter>
bool operator<(typename iterator_traits<BiIter>::value_type const& lhs,
        const sub_match<BiIter>& rhs);

Returns: rhs.compare(typename sub_match<BiIter>::string_type(1, lhs)) > 0.

template <class BiIter>
bool operator>(typename iterator_traits<BiIter>::value_type const& lhs,
        const sub_match<BiIter>& rhs);

Returns: rhs < lhs.
template <class BiIter>
    bool operator>=(typename iterator_traits<BiIter>::value_type const& lhs,
                   const sub_match<BiIter>& rhs);

    Returns: !(lhs < rhs).

template <class BiIter>
    bool operator<=(typename iterator_traits<BiIter>::value_type const& lhs,
                   const sub_match<BiIter>& rhs);

    Returns: !(rhs < lhs).

template <class BiIter>
    bool operator==(const sub_match<BiIter>& lhs,
                    typename iterator_traits<BiIter>::value_type const& rhs);

    Returns: lhs.compare(typename sub_match<BiIter>::string_type(1, rhs)) == 0.

template <class BiIter>
    bool operator!=(const sub_match<BiIter>& lhs,
                    typename iterator_traits<BiIter>::value_type const& rhs);

    Returns: !(lhs == rhs).

template <class BiIter>
    bool operator<(const sub_match<BiIter>& lhs,
                   typename iterator_traits<BiIter>::value_type const& rhs);

    Returns: rhs < lhs.

template <class BiIter>
    bool operator>=(const sub_match<BiIter>& lhs,
                    typename iterator_traits<BiIter>::value_type const& rhs);

    Returns: !(lhs < rhs).

template <class BiIter>
    bool operator<=(const sub_match<BiIter>& lhs,
                    typename iterator_traits<BiIter>::value_type const& 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()).
28.10  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` shall satisfy the requirements of an allocator-aware container and of a sequence container, as specified in 23.2.3, except that only operations defined for const-qualified sequence containers are supported.

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.

```cpp
namespace std {
    template <class BidirectionalIterator,
              class Allocator = allocator<sub_match<BidirectionalIterator>>>
    class match_results {
        public:
            typedef sub_match<BidirectionalIterator> value_type;
            typedef const value_type& const_reference;
            typedef value_type& reference;
            typedef implementation-defined const_iterator;
            typedef const_iterator iterator;
            typedef typename
                iterator_traits<BidirectionalIterator>::difference_type difference_type;
            typedef typename allocator_traits<Allocator>::size_type size_type;
            typedef Allocator allocator_type;
            typedef typename
                iterator_traits<BidirectionalIterator>::value_type char_type;
            typedef basic_string<char_type> string_type;

            // 28.10.1, 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();

            // 28.10.2, state:
            bool ready() const;

            // 28.10.3, size:
            size_type size() const;
            size_type max_size() const;
            bool empty() const;
    }  // class match_results
```

§ 28.10
// 28.10.4, element access:
difference_type length(size_type sub = 0) const;
difference_type position(size_type sub = 0) const;
string_type str(size_type sub = 0) const;
const_reference operator[](size_type n) const;

const_reference prefix() const;
const_reference suffix() const;
const_iterator begin() const;
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;

// 28.10.5, format:
template <class OutputIter>
OutputIter
format(OutputIter out,
    const char_type* fmt_first, const char_type* fmt_last,
    regex_constants::match_flag_type flags =
    regex_constants::format_default) const;
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;
basic_string<char_type, ST, SA>
format(const basic_string<char_type, ST, SA>& fmt,
    regex_constants::match_flag_type flags =
    regex_constants::format_default) const;

string_type
format(const char_type* fmt,
    regex_constants::match_flag_type flags =
    regex_constants::format_default) const;

// 28.10.6, allocator:
allocator_type get_allocator() const;

// 28.10.7, swap:
void swap(match_results& that);
};

28.10.1 match_results constructors

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.
4 \textit{Effects}: Constructs an object of class \texttt{match\_results}, as a copy of \texttt{m}.

\texttt{match\_results(match\_results&& m) noexcept;}

5 \textit{Effects}: Move-constructs an object of class \texttt{match\_results} from \texttt{m} satisfying the same postconditions as Table 142. Additionally, the stored \texttt{Allocator} value is move constructed from \texttt{m.get\_allocator()}.

6 \textit{Throws}: Nothing.

\texttt{match\_results& operator=(const match\_results& m);}  

7 \textit{Effects}: Assigns \texttt{m} to \texttt{*this}. The postconditions of this function are indicated in Table 142.

\texttt{match\_results& operator=(match\_results&& m);}  

8 \textit{Effects}: Move-assigns \texttt{m} to \texttt{*this}. The postconditions of this function are indicated in Table 142.

Table 142 — \texttt{match\_results} assignment operator effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ready()}</td>
<td>\texttt{m.ready()}</td>
</tr>
<tr>
<td>\texttt{size()}</td>
<td>\texttt{m.size()}</td>
</tr>
<tr>
<td>\texttt{str(n)}</td>
<td>\texttt{m.str(n)} for all integers (n &lt; m.size)</td>
</tr>
<tr>
<td>\texttt{prefix()}</td>
<td>\texttt{m.prefix()}</td>
</tr>
<tr>
<td>\texttt{suffix()}</td>
<td>\texttt{m.suffix()}</td>
</tr>
<tr>
<td>\texttt{(*this)[n]}</td>
<td>\texttt{m[n]} for all integers (n &lt; m.size)</td>
</tr>
<tr>
<td>\texttt{length(n)}</td>
<td>\texttt{m.length(n)} for all integers (n &lt; m.size)</td>
</tr>
<tr>
<td>\texttt{position(n)}</td>
<td>\texttt{m.position(n)} for all integers (n &lt; m.size)</td>
</tr>
</tbody>
</table>

28.10.2 \texttt{match\_results state}  

\texttt{bool ready() const;}  

1 \textit{Returns}: true if \texttt{*this} has a fully established result state, otherwise \texttt{false}.

28.10.3 \texttt{match\_results size}  

\texttt{size\_type size() const;}  

1 \textit{Returns}: One plus the number of marked sub-expressions in the regular expression that was matched if \texttt{*this} represents the result of a successful match. Otherwise returns 0. [\textit{Note}: The state of a \texttt{match\_results} object can be modified only by passing that object to \texttt{regex\_match} or \texttt{regex\_search}. Sections 28.11.2 and 28.11.3 specify the effects of those algorithms on their \texttt{match\_results} arguments. — end note]

\texttt{size\_type max\_size() const;}  

2 \textit{Returns}: The maximum number of \texttt{sub\_match} elements that can be stored in \texttt{*this}.

\texttt{bool empty() const;}  

3 \textit{Returns}: \texttt{size()} == 0.
28.10.4 match_results element access

\[
\text{difference_type length(size_type sub = 0) const;}
\]

\[
\begin{align*}
&1 & \text{Requires: } \text{ready()} == \text{true}. \\
&2 & \text{Returns: } (\text{*this})[\text{sub}].\text{length()}. \\
&3 \text{difference_type position(size_type sub = 0) const;}
\end{align*}
\]

\[
\begin{align*}
&3 & \text{Requires: } \text{ready()} == \text{true}. \\
&4 & \text{Returns: } \text{The distance from the start of the target sequence to } (\text{*this})[\text{sub}].\text{first}. \\
&5 \text{string_type str(size_type sub = 0) const;}
\end{align*}
\]

\[
\begin{align*}
&5 & \text{Requires: } \text{ready()} == \text{true}. \\
&6 & \text{Returns: } \text{string_type((*this)[sub])}. \\
&7 \text{const_reference operator[](size_type n) const;}
\end{align*}
\]

\[
\begin{align*}
&7 & \text{Requires: } \text{ready()} == \text{true}. \\
&8 & \text{Returns: } \text{A reference to the } \text{sub_match} \text{ object representing the character sequence that matched marked sub-expression n. If } n == 0 \text{ then returns a reference to a } \text{sub_match} \text{ object representing the character sequence that matched the whole regular expression. If } n >= \text{size()} \text{ then returns a } \text{sub_match} \text{ object representing an unmatched sub-expression.} \\
&9 \text{const_reference prefix() const;}
\end{align*}
\]

\[
\begin{align*}
&9 & \text{Requires: } \text{ready()} == \text{true}. \\
&10 & \text{Returns: } \text{A reference to the } \text{sub_match} \text{ object representing the character sequence from the start of the string being matched/searched to the start of the match found.} \\
&11 \text{const_reference suffix() const;}
\end{align*}
\]

\[
\begin{align*}
&11 & \text{Requires: } \text{ready()} == \text{true}. \\
&12 & \text{Returns: } \text{A reference to the } \text{sub_match} \text{ object representing the character sequence from the end of the match found to the end of the string being matched/searched.} \\
&13 \text{const_iterator begin() const;}
\end{align*}
\]

\[
\begin{align*}
&13 & \text{Returns: } \text{A starting iterator that enumerates over all the sub-expressions stored in } \text{*this}. \\
&14 \text{const_iterator end() const;}
\end{align*}
\]

\[
\begin{align*}
&14 & \text{Returns: } \text{A terminating iterator that enumerates over all the sub-expressions stored in } \text{*this}. \\
&15 \text{28.10.5 match_results formatting}
\end{align*}
\]

\[
\text{\begin{verbatim}
\text{template <class OutputIter>}
\text{OutputIter format(OutputIter out,}
\text{ \text{const char_type* fmt_first, const char_type* fmt_last,}
\text{ \text{regex_constants::match_flag_type flags =}
\text{ \text{regex_constants::format_default}) const;}
\end{verbatim}}
\]

§ 28.10.5
Requires: \( \text{ready}() == \text{true} \) and \text{OutputIter} \ shall satisfy the requirements for an Output Iterator (24.2.4).

Effects: Copies the character sequence \([\text{fmt\_first},\text{fmt\_last})\) to \text{OutputIter} \text{out}. Replaces each format specifier or escape sequence in the copied range with either the character(s) it represents or the sequence of characters within \(*\text{this}\) to which it refers. The bitmasks specified in \text{flags} determine which format specifiers and escape sequences are recognized.

Returns: \text{out}.

template <class OutputIter, class ST, class SA>
OutputIter format(OutputIter out,
    const basic_string<char_type, ST, SA>& fmt,
    regex_constants::match_flag_type flags =
    regex_constants::format_default) const;

Effects: Equivalent to \( \text{return format(out, fmt.data(), fmt.data() + fmt.size(), flags)} \).

template <class ST, class SA>
basic_string<char_type, ST, SA>
format(const basic_string<char_type, ST, SA>& fmt,
    regex_constants::match_flag_type flags =
    regex_constants::format_default) const;

Requires: \( \text{ready}() == \text{true} \).

Effects: Constructs an empty string \text{result} of type \text{basic_string<char_type, ST, SA>} and calls \text{format(back_inserter(result), fmt, flags)}.

Returns: \text{result}.

string_type
format(const char_type* fmt,
    regex_constants::match_flag_type flags =
    regex_constants::format_default) const;

Requires: \( \text{ready}() == \text{true} \).

Effects: Constructs an empty string \text{result} of type \text{string_type} and calls \text{format(back_inserter(result), fmt, fmt + char_traits<char_type>::length(fmt), flags)}.

Returns: \text{result}.

### 28.10.6 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.

### 28.10.7 match_results swap

 void swap(match_results& that);

Effects: Swaps the contents of the two sequences.

Postcondition: \(*\text{this}\) contains the sequence of matched sub-expressions that were in \text{that}, \text{that} contains the sequence of matched sub-expressions that were in \(*\text{this}\).

Complexity: Constant time.

template <class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1,
    match_results<BidirectionalIterator, Allocator>& m2);

Effects: \( \text{m1} \text{.swap(m2)} \).

§ 28.10.7

1127
28.10.8 match_results non-member functions

<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>
</tbody>
</table>

28.11 Regular expression algorithms

28.11.1 exceptions

1. 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.

28.11.2 regex_match

| Requires: | The type BidirectionalIterator shall satisfy the requirements of a Bidirectional Iterator (24.2.6). |
| 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. Returns true if such a match 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 143. |
### Table 143 — Effects of `regex_match` algorithm (continued)

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>m.prefix().second</td>
<td>first</td>
</tr>
<tr>
<td>m.prefix().matched</td>
<td>false</td>
</tr>
<tr>
<td>m.suffix().first</td>
<td>last</td>
</tr>
<tr>
<td>m.suffix().second</td>
<td>last</td>
</tr>
<tr>
<td>m.suffix().matched</td>
<td>false</td>
</tr>
<tr>
<td>m[0].first</td>
<td>first</td>
</tr>
<tr>
<td>m[0].second</td>
<td>last</td>
</tr>
<tr>
<td>m[0].matched</td>
<td>true</td>
</tr>
<tr>
<td>m[n].first</td>
<td>For all integers $0 &lt; n &lt; m.size()$, the start of the sequence that matched sub-expression $n$. Alternatively, if sub-expression $n$ did not participate in the match, then last.</td>
</tr>
<tr>
<td>m[n].second</td>
<td>For all integers $0 &lt; n &lt; m.size()$, the end of the sequence that matched sub-expression $n$. Alternatively, if sub-expression $n$ did not participate in the match, then last.</td>
</tr>
<tr>
<td>m[n].matched</td>
<td>For all integers $0 &lt; n &lt; m.size()$, true if sub-expression $n$ participated in the match, false otherwise.</td>
</tr>
</tbody>
</table>

```cpp
template <class BidirectionalIterator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags =
                 regex_constants::match_default);
```

4 **Effects:** Behaves “as if” by constructing an instance of `match_results<BidirectionalIterator>` `what`, and then returning the result of `regex_match(first, last, what, e, flags)`.

```cpp
template <class charT, class Allocator, class traits>
bool regex_match(const charT* str,
                 match_results<const charT*, Allocator>& m,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags =
                 regex_constants::match_default);
```

5 **Returns:** `regex_match(str, str + char_traits<charT>::length(str), m, e, flags)`.

```cpp
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);
```

6 **Returns:** `regex_match(s.begin(), s.end(), m, e, flags)`.

```cpp
template <class charT, class traits>
bool regex_match(const charT* str,
                 ...
```
const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags =
    regex_constants::match_default);

Returns: regex_match(str, str + char_traits<charT>::length(str), e, flags)

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).

28.11.3 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 (24.2.6).

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 144.

Table 144 — 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>
<tr>
<td>m[n].first</td>
<td>For all integers 0 &lt; n &lt; m.size(), the start of the sequence that matched sub-expression n. Alternatively, if sub-expression n did not participate in the match, then last.</td>
</tr>
</tbody>
</table>
Table 144 — Effects of regex_search algorithm (continued)

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>m[n].second</td>
<td>For all integers 0 &lt; n &lt; m.size(), the end of the sequence that matched sub-expression n. Alternatively, if sub-expression n did not participate in the match, then last.</td>
</tr>
<tr>
<td>m[n].matched</td>
<td>For all integers 0 &lt; n &lt; m.size(), true if sub-expression n participated in the match, false otherwise.</td>
</tr>
</tbody>
</table>

```cpp
template <class charT, class Allocator, class traits>
bool regex_search(const charT* str, match_results<const charT*, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags =
                  regex_constants::match_default);
```

4 Returns: The result of regex_search(str, str + char_traits<charT>::length(str), m, e, flags).

```cpp
template <class ST, class SA, class Allocator, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
                  match_results<typename basic_string<charT, ST, SA>::const_iterator,
                  Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags =
                  regex_constants::match_default);
```

5 Returns: The result of regex_search(s.begin(), s.end(), m, e, flags).

```cpp
template <class BidirectionalIterator, class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags =
                  regex_constants::match_default);
```

6 Effects: Behaves “as if” by constructing an object what of type match_results<BidirectionalIterator> and then returning the result of regex_search(first, last, what, e, flags).

```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);
```

7 Returns: regex_search(str, str + char_traits<charT>::length(str), e, flags)

```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);
```

8 Returns: regex_search(s.begin(), s.end(), e, flags).
28.11.4 \texttt{regex\_replace} \hfill [re.alg.replace]

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

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

\textit{Effects:} Constructs a \texttt{regex\_iterator} object \texttt{i} as if by
\begin{verbatim}
    regex\_iterator<BidirectionalIterator, charT, traits> i(first, last, e, flags)
\end{verbatim}
and uses \texttt{i} to enumerate through all of the matches \texttt{m} of type \texttt{match\_results<BidirectionalIterator>} that occur within the sequence \texttt{[first, last)}. If no such matches are found and \texttt{!(flags \& regex\_constants::format\_no\_copy)} then calls
\begin{verbatim}
    out = std::copy(first, last, out)
\end{verbatim}
If any matches are found then, for each such match:

\begin{enumerate}
\item[(1.1)] If \texttt{!(flags \& regex\_constants::format\_no\_copy)}, calls
\begin{verbatim}
    out = std::copy(m.prefix().first, m.prefix().second, out)
\end{verbatim}
\item[(1.2)] Then calls
\begin{verbatim}
    out = m.format(out, fmt, flags)
\end{verbatim}
for the first form of the function and
\begin{verbatim}
    out = m.format(out, fmt, fmt + char\_traits<charT>::length(fmt), flags)
\end{verbatim}
for the second.

Finally, if such a match is found and \texttt{!(flags \& regex\_constants::format\_first\_only)} is non-zero then only the first match found is replaced.

\textit{Returns:} \texttt{out}.

\begin{verbatim}
\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, 
\end{verbatim}

\section*{§ 28.11.4}
28.12 Regular expression iterators

28.12.1 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 {
public:
    typedef basic_regex<charT, traits>   regex_type;
    typedef match_results<BidirectionalIterator> value_type;
    typedef std::ptrdiff_t         difference_type;
    typedef const value_type*      pointer;
    typedef const value_type&     reference;
    typedef std::forward_iterator_tag iterator_category;

    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 a, BidirectionalIterator b,
                   const regex_type&& re,
                   regex_constants::match_flag_type m = 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

};

2 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]

28.12.1.1 regex_iterator constructors [re.regexiter.cnstr]

regex_iterator();

    Effects: Constructs an end-of-sequence iterator.

regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
               const regex_type& re,
               regex_constants::match_flag_type m = regex_constants::match_default);

    Effects: Initializes begin and end to a and b, respectively, sets pregex to &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.

§ 28.12.1.1  1134
28.12.1.2 regex_iterator comparisons

```cpp
bool operator==(const regex_iterator& right) const;
```  
*Returns:* true if *this and right are both end-of-sequence iterators or if the following conditions all hold:

1. \( \text{begin} == \text{right.begin} \)
2. \( \text{end} == \text{right.end} \)
3. \( \text{pregex} == \text{right.pregex} \)
4. \( \text{flags} == \text{right.flags} \)
5. \( \text{match}[0] == \text{right.match}[0] \)

otherwise false.

```cpp
bool operator!=(const regex_iterator& right) const;
```  
*Returns:* !(*this == right).

28.12.1.3 regex_iterator indirection

```cpp
const value_type& operator*() const;
```  
*Returns:* match.

```cpp
const value_type* operator->() const;
```  
*Returns:* &match.

28.12.1.4 regex_iterator increment

```cpp
regex_iterator& operator++();
```  
*Effects:* Constructs a local variable start of type BidirectionalIterator and initializes it with the value of match[0].second.

1. If the iterator holds a zero-length match and start == end the operator sets *this to the end-of-sequence iterator and returns *this.
2. Otherwise, if the iterator holds a zero-length match the operator calls regex_search(start, end, match, *pregex, flags | regex_constants::match_not_null | regex_constants::match_continuous). If the call returns true the operator returns *this. Otherwise the operator increments start and continues as if the most recent match was not a zero-length match.
3. If the most recent match was not a zero-length match, the operator sets flags to flags | regex_constants::match_prev_avail and calls regex_search(start, end, match, *pregex, flags). If the call returns false the iterator sets *this to the end-of-sequence iterator. The iterator then returns *this.
4. In all cases in which the call to regex_search returns true, match.prefix().first shall be equal to the previous value of match[0].second, and for each index i in the half-open range [0, match.size()) for which match[i].matched is true, match[i].position() shall return distance(begin, match[i].first).
5. [Note: This means that match[i].position() gives the offset from the beginning of the target sequence, which is often not the same as the offset from the sequence passed in the call to regex_search. — end note]
6. It is unspecified how the implementation makes these adjustments.
7. [Note: This means that a compiler may call an implementation-specific search function, in which case a user-defined specialization of regex_search will not be called. — end note]
regex_iterator operator++(int);

Effects:

    regex_iterator tmp = *this;
    ++(*this);
    return tmp;

28.12.2 Class template regex_token_iterator [re tokiter]

The class template regex_token_iterator is an iterator adaptor; that is to say it represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence, and presenting one or more sub-expressions for each match found. Each position enumerated by the iterator is a sub_match class template instance that represents what matched a particular sub-expression within the regular expression.

When class regex_token_iterator is used to enumerate a single sub-expression with index -1 the iterator performs field splitting: that is to say it enumerates one sub-expression for each section of the character container sequence that does not match the regular expression specified.

After it is constructed, the iterator finds and stores a value regex_iterator&lt;BidirectionalIterator&gt; position and sets the internal count N to zero. It also maintains a sequence subs which contains a list of the sub-expressions which will be enumerated. Every time operator++ is used the count N is incremented; if N exceeds or equals subs.size(), then the iterator increments member position and sets count N to zero.

If the end of sequence is reached (position is equal to the end of sequence iterator), the iterator becomes equal to the end-of-sequence iterator value, unless the sub-expression being enumerated has index -1, in which case the iterator enumerates one last sub-expression that contains all the characters from the end of the last regular expression match to the end of the input sequence being enumerated, provided that this would not be an empty sub-expression.

The default constructor constructs an end-of-sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of operator* on an end-of-sequence iterator is not defined. For any other iterator value a const sub_match&lt;BidirectionalIterator&gt;&amp; is returned. The result of operator-&gt; on an end-of-sequence iterator is not defined. For any other iterator value a const sub_match&lt;BidirectionalIterator&gt;* is returned.

It is impossible to store things into regex_token_iterators. Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.

namespace std {
    template &lt;class BidirectionalIterator,
     class charT = typename iterator_traits&lt;
        BidirectionalIterator&gt;::value_type,
     class traits = regex_traits&lt;charT&gt; &gt;
    class regex_token_iterator {
        public:
            typedef basic_regex&lt;charT, traits&gt; regex_type;
            typedef sub_match&lt;BidirectionalIterator&gt; value_type;
            typedef std::ptrdiff_t difference_type;
            typedef const value_type* pointer;
            typedef const value_type& reference;
            typedef std::forward_iterator_tag iterator_category;

            regex_token_iterator();
            regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                const regex_type&amp; re,
                &lt;other parameters&gt;);
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 std::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 <std::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 std::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 <std::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:

typedef
regex_iterator<BidirectionalIterator, charT, traits> position_iterator; // exposition only
position_iterator position; // exposition only
const value_type* result; // exposition only
value_type suffix; // exposition only
std::size_t N; // exposition only

§ 28.12.2
A suffix iterator is a regex_token_iterator object that points to a final sequence of characters at the end of the target sequence. In a suffix iterator the member result holds a pointer to the data member suffix, the value of the member suffix.match is true, suffix.first points to the beginning of the final sequence, and suffix.second points to the end of the final sequence.

[Note: For a suffix iterator, data member suffix.first is the same as the end of the last match found, and suffix.second is the same as the end of the target sequence — end note]

The current match is (*position).prefix() if subs[N] == -1, or (*position)[subs[N]] for any other value of subs[N].

### 28.12.2.1 regex_token_iterator constructors

```cpp
regex_token_iterator();
```

**Effects:** Constructs the end-of-sequence iterator.

```cpp
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b, const regex_type& re, int submatch = 0, regex_constants::match_flag_type m = regex_constants::match_default);
```

```cpp
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b, const std::vector<int>& submatches, regex_constants::match_flag_type m = regex_constants::match_default);
```

```cpp
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b, const initializer_list<int> submatches, regex_constants::match_flag_type m = regex_constants::match_default);
```

```cpp
template <std::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

§ 28.12.2.1 1138
iterator that points to the range \([a, b)\), otherwise the constructor sets \(*this\) to an end-of-sequence iterator.

28.12.22 regex_token_iterator comparisons

bool operator==(const regex_token_iterator& right) const;

1 Returns: true if \(*this\) and \(right\) are both end-of-sequence iterators, or if \(*this\) and \(right\) are both suffix iterators and \(suffix == right.suffix\); otherwise returns \(false\) if \(*this\) or \(right\) is an end-of-sequence iterator or a suffix iterator. Otherwise returns \(true\) if \(position == right.position\), \(N == right.N\), and \(subs == right.subs\). Otherwise returns \(false\).

bool operator!=(const regex_token_iterator& right) const;

2 Returns: \(!(*this == right)\).

28.12.23 regex_token_iterator indirection

const value_type& operator*() const;

1 Returns: \(*result\).

const value_type* operator->() const;

2 Returns: \(result\).

28.12.24 regex_token_iterator increment

regex_token_iterator& operator++();

1 Effects: Constructs a local variable \(prev\) of type position_iterator, initialized with the value of \(position\).

If \(*this\) is a suffix iterator, sets \(*this\) to an end-of-sequence iterator.

Otherwise, if \(N + 1 < subs.size()\), increments \(N\) and sets \(result\) to the address of the current match.

Otherwise, sets \(N\) to 0 and increments \(position\). If \(position\) is not an end-of-sequence iterator the operator sets \(result\) to the address of the current match.

Otherwise, if any of the values stored in \(subs\) is equal to -1 and \(prev->suffix().length()\) is not 0 the operator sets \(*this\) to a suffix iterator that points to the range \([prev->suffix().first, prev->suffix().second)\).

Otherwise, sets \(*this\) to an end-of-sequence iterator.

Returns: \(*this\)

regex_token_iterator& operator++(int);

7 Effects: Constructs a copy \(tmp\) of \(*this\), then calls ++(*this).

8 Returns: \(tmp\).

28.13 Modified ECMAScript regular expression grammar

The regular expression grammar recognized by basic_regex objects constructed with the ECMAScript flag is that specified by ECMA-262, except as specified below.

2 Objects of type specialization of basic_regex store within themselves a default-constructed instance of their traits template parameter, henceforth referred to as traits_inst. This traits_inst object is used to support localization of the regular expression: basic_regex member functions shall not call any locale dependent C or C++ API, including the formatted string input functions. Instead they shall call the appropriate traits member function to achieve the required effect.
The following productions within the ECMAScript grammar are modified as follows:

```plaintext
ClassAtom ::
   -
   ClassAtomNoDash
   ClassAtomExClass
   ClassAtomCollatingElement
   ClassAtomEquivalence
```

The following new productions are then added:

```plaintext
ClassAtomExClass ::
   [: ClassName :]

ClassAtomCollatingElement ::
   [. ClassName .]

ClassAtomEquivalence ::
   [= ClassName =]

ClassName ::
   ClassNameCharacter
   ClassNameCharacter ClassName

ClassNameCharacter ::
   SourceCharacter but not one of "." "=" ":" 
```

The productions `ClassAtomExClass`, `ClassAtomCollatingElement` and `ClassAtomEquivalence` provide functionality equivalent to that of the same features in regular expressions in POSIX.

The regular expression grammar may be modified by any `regex_constants::syntax_option_type` flags specified when constructing an object of type specialization of `basic_regex` according to the rules in Table 138.

A `ClassName` production, when used in `ClassAtomExClass`, is not valid if `traits_inst.lookup_classname` returns zero for that name. The names recognized as valid `ClassNames` are determined by the type of the traits class, but at least the following names shall be recognized: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`, `d`, `s`, `w`. In addition the following expressions shall be equivalent:

- \d and `[[:digit:]]`
- \D and `[^[:digit:]]`
- \s and `[[:space:]]`
- \S and `[^[:space:]]`
- \w and `[_[:alnum:]]`
- \W and `[^_[:alnum:]]`

A `ClassName` production when used in a `ClassAtomCollatingElement` production is not valid if the value returned by `traits_inst.lookup_collatename` for that name is an empty string.

The results from multiple calls to `traits_inst.lookup_classname` can be bitwise OR’ed together and subsequently passed to `traits_inst.isectype`.

§ 28.13
A ClassName production when used in a ClassAtomEquivalence production is not valid if the value returned by traits_inst.lookup_collatename for that name is an empty string or if the value returned by traits_inst.transform_primary for the result of the call to traits_inst.lookup_collatename is an empty string.

When the sequence of characters being transformed to a finite state machine contains an invalid class name the translator shall throw an exception object of type regex_error.

If the CV of a UnicodeEscapeSequence is greater than the largest value that can be held in an object of type charT the translator shall throw an exception object of type regex_error. [Note: This means that values of the form "uxxxx" that do not fit in a character are invalid. — end note]

Where the regular expression grammar requires the conversion of a sequence of characters to an integral value, this is accomplished by calling traits_inst.value.

The behavior of the internal finite state machine representation when used to match a sequence of characters is as described in ECMA-262. The behavior is modified according to any match_flag_type flags 28.5.2 specified when using the regular expression object in one of the regular expression algorithms 28.11. The behavior is also localized by interaction with the traits class template parameter as follows:

(14.1) — During matching of a regular expression finite state machine against a sequence of characters, two characters c and d are compared using the following rules:

1. if (flags() & regex_constants::icase) the two characters are equal if traits_inst.translate_nocase(c) == traits_inst.translate_nocase(d);
2. otherwise, if flags() & regex_constants::collate the two characters are equal if traits_inst.translate(c) == traits_inst.translate(d);
3. otherwise, the two characters are equal if c == d.

(14.2) — During matching of a regular expression finite state machine against a sequence of characters, comparison of a collating element range c1-c2 against a character c is conducted as follows: if flags() & regex_constants::collate is false then the character c is matched if c1 <= c && c <= c2, otherwise c is matched in accordance with the following algorithm:

```cpp
string_type str1 = string_type(1, flags() & icase ? traits_inst.translate_nocase(c1) : traits_inst.translate(c1);
string_type str2 = string_type(1, flags() & icase ? traits_inst.translate_nocase(c2) : traits_inst.translate(c2);
string_type str = string_type(1, flags() & icase ? traits_inst.translate_nocase(c) : traits_inst.translate(c);
return traits_inst.transform(str1.begin(), str1.end()) <= traits_inst.transform(str.begin(), str.end())
&& traits_inst.transform(str.begin(), str.end())
<= traits_inst.transform(str2.begin(), str2.end());
```

(14.3) — During matching of a regular expression finite state machine against a sequence of characters, testing whether a collating element is a member of a primary equivalence class is conducted by first converting the collating element and the equivalence class to sort keys using traits::transform_primary, and then comparing the sort keys for equality.

(14.4) — During matching of a regular expression finite state machine against a sequence of characters, a character c is a member of a character class designated by an iterator range [first, last) if traits_inst.isctype(c, traits_inst.lookup_classname(first, last, flags() & icase)) is true.
29 Atomic operations library

29.1 General

This Clause describes components for fine-grained atomic access. This access is provided via operations on atomic objects.

The following subclauses describe atomics requirements and components for types and operations, as summarized below.

Table 145 — Atomics library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.3 Order and Consistency</td>
<td></td>
</tr>
<tr>
<td>29.4 Lock-free Property</td>
<td></td>
</tr>
<tr>
<td>29.5 Atomic Types</td>
<td>&lt;atomic&gt;</td>
</tr>
<tr>
<td>29.6 Operations on Atomic Types</td>
<td></td>
</tr>
<tr>
<td>29.7 Flag Type and Operations</td>
<td></td>
</tr>
<tr>
<td>29.8 Fences</td>
<td></td>
</tr>
</tbody>
</table>

29.2 Header <atomic> synopsis

namespace std {
    // 29.3, order and consistency
    enum memory_order;
    template <class T>
        T kill_dependency(T y) noexcept;

    // 29.4, 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

    // 29.5, generic types
    template<class T> struct atomic;
    template<> struct atomic<integral>;
    template<class T> struct atomic<T*>;

    // 29.6.1, general operations on atomic types
    // In the following declarations, atomic-type is either
    // atomic<T> or a named base class for T from
    // Table 146 or inferred from Table 147 or from bool.
    // If it is atomic<T>, then the declaration is a template
    // declaration prefixed with template <class T>.

§ 29.2
bool atomic_is_lock_free(const volatile atomic-type*) noexcept;
bool atomic_is_lock_free(const atomic-type*) noexcept;
void atomic_init(volatile atomic-type*, T) noexcept;
void atomic_init(atomic-type*, T) noexcept;
void atomic_store(volatile atomic-type*, T) noexcept;
void atomic_store(atomic-type*, T) noexcept;
void atomic_store_explicit(volatile atomic-type*, T, memory_order) noexcept;
void atomic_store_explicit(atomic-type*, T, memory_order) noexcept;
T atomic_load(const volatile atomic-type*) noexcept;
T atomic_load(const atomic-type*) noexcept;
T atomic_load_explicit(const volatile atomic-type*, memory_order) noexcept;
T atomic_load_explicit(const atomic-type*, memory_order) noexcept;
T atomic_exchange(volatile atomic-type*, T) noexcept;
T atomic_exchange(atomic-type*, T) noexcept;
T atomic_exchangeExplicit(volatile atomic-type*, T, memory_order) noexcept;
T atomic_exchangeExplicit(atomic-type*, T, memory_order) noexcept;
bool atomic_compare_exchange_weak(volatile atomic-type*, T*, T) noexcept;
bool atomic_compare_exchange_weak(atomic-type*, T*, T) noexcept;
bool atomic_compare_exchange_strong(volatile atomic-type*, T*, T) noexcept;
bool atomic_compare_exchange_strong(atomic-type*, T*, T) noexcept;
bool atomic_compare_exchange_weakExplicit(volatile atomic-type*, T*, T,
memory_order, memory_order) noexcept;
bool atomic_compare_exchange_weakExplicit(atomic-type*, T*, T,
memory_order, memory_order) noexcept;
bool atomic_compare_exchange_strongExplicit(volatile atomic-type*, T*, T,
memory_order, memory_order) noexcept;
bool atomic_compare_exchange_strongExplicit(atomic-type*, T*, T,
memory_order, memory_order) noexcept;

// 29.6.2, templated operations on atomic types
template <class T>
  T atomic_fetch_add(volatile atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_add(atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_addExplicit(volatile atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_addExplicit(atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_sub(volatile atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_sub(atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_subExplicit(volatile atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_subExplicit(atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_and(volatile atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_and(atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_andExplicit(volatile atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_andExplicit(atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_andExplicit(atomic<T>*, T, memory_order) noexcept;

§ 29.2
T atomic_fetch_or(volatile atomic<T>*, T) noexcept;

template <class T>
T atomic_fetch_or(atomic<T>*, T) noexcept;

template <class T>
T atomic_fetch_or_explicit(volatile atomic<T>*, T, memory_order) noexcept;

template <class T>
T atomic_fetch_or_explicit(atomic<T>*, T, memory_order) noexcept;

T atomic_fetch_xor(volatile atomic<T>*, T) noexcept;

template <class T>
T atomic_fetch_xor(atomic<T>*, T) noexcept;

template <class T>
T atomic_fetch_xor_explicit(volatile atomic<T>*, T, memory_order) noexcept;

template <class T>
T atomic_fetch_xor_explicit(atomic<T>*, T, memory_order) noexcept;

// 29.6.3, arithmetic operations on atomic types
// In the following declarations, atomic-integral is either
// atomic<T> or a named base class for T from
// Table 146 or inferred from Table 147.
// If it is atomic<T>, then the declaration is a template
// specialization declaration prefixed with template < >.

int integral atomic_fetch_add(volatile atomic<integral>*), integral) noexcept;

int integral atomic_fetch_add(atomic<integral>*, integral) noexcept;

int integral atomic_fetch_add_explicit(volatile atomic<integral>*, integral, memory_order) noexcept;

int integral atomic_fetch_add_explicit(atomic<integral>*, integral, memory_order) noexcept;

int integral atomic_fetch_sub(volatile atomic<integral>*), integral) noexcept;

int integral atomic_fetch_sub(atomic<integral>*, integral) noexcept;

int integral atomic_fetch_sub_explicit(volatile atomic<integral>*, integral, memory_order) noexcept;

int integral atomic_fetch_sub_explicit(atomic<integral>*, integral, memory_order) noexcept;

int integral atomic_fetch_and(volatile atomic<integral>*), integral) noexcept;

int integral atomic_fetch_and(atomic<integral>*, integral) noexcept;

int integral atomic_fetch_and_explicit(volatile atomic<integral>*, integral, memory_order) noexcept;

int integral atomic_fetch_and_explicit(atomic<integral>*, integral, memory_order) noexcept;

int integral atomic_fetch_or(volatile atomic<integral>*), integral) noexcept;

int integral atomic_fetch_or(atomic<integral>*, integral) noexcept;

int integral atomic_fetch_or_explicit(volatile atomic<integral>*, integral, memory_order) noexcept;

int integral atomic_fetch_or_explicit(atomic<integral>*, integral, memory_order) noexcept;

int integral atomic_fetch_xor(volatile atomic<integral>*), integral) noexcept;

int integral atomic_fetch_xor(atomic<integral>*, integral) noexcept;

int integral atomic_fetch_xor_explicit(volatile atomic<integral>*, integral, memory_order) noexcept;

int integral atomic_fetch_xor_explicit(atomic<integral>*, integral, memory_order) noexcept;

// 29.6.4, partial specializations for pointers

template <class T>
T* atomic_fetch_add(volatile atomic<T>**), ptrdiff_t) noexcept;

template <class T>
T* atomic_fetch_add(atomic<T>**), ptrdiff_t) noexcept;

template <class T>
T* atomic_fetch_add_explicit(volatile atomic<T>**), ptrdiff_t, memory_order) noexcept;

template <class T>
T* atomic_fetch_add_explicit(atomic<T>**), ptrdiff_t, memory_order) noexcept;
template <class T>
T* atomic_fetch_sub_volatile(T**, ptrdiff_t) noexcept;

template <class T>
T* atomic_fetch_sub(T**, ptrdiff_t) noexcept;

// 29.6.5, initialization
#define ATOMIC_VAR_INIT(value) see below

// 29.7, flag type and operations
struct atomic_flag;
bool atomic_flag_test_and_set_volatile(atomic_flag*) noexcept;
bool atomic_flag_test_and_set(atomic_flag*) noexcept;
bool atomic_flag_test_and_set_explicit_volatile(atomic_flag*, memory_order) noexcept;
bool atomic_flag_test_and_set_explicit(atomic_flag*, memory_order) noexcept;
void atomic_flag_clear_volatile(atomic_flag*) noexcept;
void atomic_flag_clear(atomic_flag*) noexcept;
void atomic_flag_clear_explicit_volatile(atomic_flag*, memory_order) noexcept;
void atomic_flag_clear_explicit(atomic_flag*, memory_order) noexcept;
#define ATOMIC_FLAG_INIT see below

// 29.8, fences
extern "C" void atomic_thread_fence(memory_order) noexcept;
extern "C" void atomic_signal_fence(memory_order) noexcept;

29.3 Order and consistency
[atomics.order]

namespace std {

typedef enum memory_order {
    memory_order_relaxed, memory_order_consume, memory_order_acquire,
    memory_order_release, memory_order_acq_rel, memory_order_seq_cst
} memory_order;
}

1 The enumeration memory_order specifies the detailed regular (non-atomic) memory synchronization order as defined in 1.10 and may provide for operation ordering. Its enumerated values and their meanings are as follows:

(1.1) — memory_order_relaxed: no operation orders memory.

(1.2) — memory_order_release, memory_order_acq_rel, and memory_order_seq_cst: a store operation performs a release operation on the affected memory location.

(1.3) — memory_order_consume: a load operation performs a consume operation on the affected memory location.

(1.4) — 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 \texttt{memory\_order\_seq\_cst} operations, consistent with the “happens before” order and modification orders for all affected locations, such that each \texttt{memory\_order\_seq\_cst} operation \( B \) that loads a value from an atomic object \( M \) observes one of the following values:

\begin{enumerate}
  \item[(3.1)] the result of the last modification \( A \) of \( M \) that precedes \( B \) in \( S \), if it exists, or
  \item[(3.2)] if \( A \) exists, the result of some modification of \( M \) that is not \texttt{memory\_order\_seq\_cst} and that does not happen before \( A \), or
  \item[(3.3)] if \( A \) does not exist, the result of some modification of \( M \) that is not \texttt{memory\_order\_seq\_cst}.
\end{enumerate}

\[ \text{Note: Although it is not explicitly required that } S \text{ include locks, it can always be extended to an order that does include lock and unlock operations, since the ordering between those is already included in the “happens before” ordering. —end note} \]

4 For an atomic operation \( B \) that reads the value of an atomic object \( M \), if there is a \texttt{memory\_order\_seq\_cst} fence \( X \) sequenced before \( B \), then \( B \) observes either the last \texttt{memory\_order\_seq\_cst} modification of \( M \) preceding \( X \) in the total order \( S \) or a later modification of \( M \) in its modification order.

5 For atomic operations \( A \) and \( B \) on an atomic object \( M \), where \( A \) modifies \( M \) and \( B \) takes its value, if there is a \texttt{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.

6 For atomic operations \( A \) and \( B \) on an atomic object \( M \), where \( A \) modifies \( M \) and \( B \) takes its value, if 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 \( S \), then \( B \) observes either the effects of \( A \) or a later modification of \( M \) in its modification order.

7 For atomic modifications \( A \) and \( B \) of an atomic object \( M \), \( B \) occurs later than \( A \) in the modification order of \( M \) if:

\begin{enumerate}
  \item[(7.1)] there is a \texttt{memory\_order\_seq\_cst} fence \( X \) such that \( A \) is sequenced before \( X \), and \( X \) precedes \( B \) in \( S \), or
  \item[(7.2)] there is a \texttt{memory\_order\_seq\_cst} fence \( Y \) such that \( Y \) is sequenced before \( B \), and \( A \) precedes \( Y \) in \( S \), or
  \item[(7.3)] 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 \( S \).
\end{enumerate}

\[ \text{Note: \texttt{memory\_order\_seq\_cst} ensures sequential consistency only for a program that is free of data races and uses exclusively \texttt{memory\_order\_seq\_cst} operations. Any use of weaker ordering will invalidate this guarantee unless extreme care is used. In particular, \texttt{memory\_order\_seq\_cst} fences ensure a total order only for the fences themselves. Fences cannot, in general, be used to restore sequential consistency for atomic operations with weaker ordering specifications. —end note} \]

8 Implementations should ensure that no “out-of-thin-air” values are computed that circularly depend on their own computation.

\[ \text{Note: For example, with } x \text{ and } y \text{ initially zero,} \]

\[
\begin{verbatim}
// Thread 1:
    r1 = y.load(memory_order_relaxed);
    x.store(r1, memory_order_relaxed);
\end{verbatim}
\]
// 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.

10 Implementations should make atomic stores visible to atomic loads within a reasonable amount of time.

template <class T>
T kill_dependency(T y) noexcept;

13 Effects: The argument does not carry a dependency to the return value (1.10).

Returns: y.

29.4 Lock-free property [atomics.lockfree]

#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

1 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.

2 The function atomic_is_lock_free (29.6) indicates whether the object is lock-free. In any given program execution, the result of the lock-free query shall be consistent for all pointers of the same type.

3 [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]

29.5 Atomic types [atomics.types.generic]
namespace std {
    template <class T> struct atomic {
        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;
        integral() 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;
    };
    template <> struct atomic<integral> {
        bool is_lock_free() const volatile noexcept;
        bool is_lock_free() const noexcept;
        void store(integral, memory_order = memory_order_seq_cst) volatile noexcept;
        void store(integral, memory_order = memory_order_seq_cst) noexcept;
        integral load(memory_order = memory_order_seq_cst) const volatile noexcept;
        integral load(memory_order = memory_order_seq_cst) const noexcept;
        operator integral() const volatile noexcept;
        operator integral() const noexcept;
        integral exchange(integral, memory_order = memory_order_seq_cst) volatile noexcept;
        integral exchange(integral, memory_order = memory_order_seq_cst) noexcept;
        bool compare_exchange_weak(integral&, integral, memory_order, memory_order) volatile noexcept;
        bool compare_exchange_weak(integral&, integral, memory_order, memory_order) noexcept;
        bool compare_exchange_strong(integral&, integral, memory_order, memory_order) volatile noexcept;
        bool compare_exchange_strong(integral&, integral, memory_order, memory_order) noexcept;
        bool compare_exchange_weak(integral&, integral, memory_order = memory_order_seq_cst) volatile noexcept;
        bool compare_exchange_weak(integral&, integral, memory_order = memory_order_seq_cst) noexcept;
        bool compare_exchange_strong(integral&, integral, memory_order = memory_order_seq_cst) volatile noexcept;
        bool compare_exchange_strong(integral&, integral, memory_order = memory_order_seq_cst) noexcept;
        integral fetch_add(integral, memory_order = memory_order_seq_cst) volatile noexcept;
        integral fetch_add(integral, memory_order = memory_order_seq_cst) noexcept;
        integral fetch_sub(integral, memory_order = memory_order_seq_cst) volatile noexcept;
        integral fetch_sub(integral, memory_order = memory_order_seq_cst) noexcept;
        integral fetch_and(integral, memory_order = memory_order_seq_cst) volatile noexcept;
        integral fetch_and(integral, memory_order = memory_order_seq_cst) noexcept;
    };
};
integral fetch_or(integral, memory_order = memory_order_seq_cst) volatile noexcept;
integral fetch_or(integral, memory_order = memory_order_seq_cst) noexcept;
integral fetch_xor(integral, memory_order = memory_order_seq_cst) volatile noexcept;
integral fetch_xor(integral, memory_order = memory_order_seq_cst) noexcept;

atomic() noexcept = default;
constexpr atomic(integral) noexcept;
atomic(const atomic&) = delete;
atomic operator=(const atomic&) volatile = delete;
integral operator=(integral) volatile noexcept;
integral operator=(integral) noexcept;

integral operator++(int) volatile noexcept;
integral operator++(int) noexcept;
integral operator--(int) volatile noexcept;
integral operator--(int) noexcept;
integral operator[](integral) volatile noexcept;
integral operator[](integral) noexcept;

integral operator+=(integral) volatile noexcept;
integral operator+=(integral) noexcept;
integral operator-=(integral) volatile noexcept;
integral operator-=(integral) noexcept;
integral operator&=(integral) volatile noexcept;
integral operator&=(integral) noexcept;
integral operator|=(integral) volatile noexcept;
integral operator|=(integral) noexcept;
integral operator^=(integral) volatile noexcept;
integral operator^=(integral) noexcept;

};

template<class T> struct atomic<T*> {  
bool is_lock_free() const volatile noexcept;
bool is_lock_free() const noexcept;
void store(T*, memory_order = memory_order_seq_cst) volatile noexcept;
void store(T*, memory_order = memory_order_seq_cst) noexcept;
T* load(memory_order = memory_order_seq_cst) const volatile noexcept;
T* load(memory_order = memory_order_seq_cst) const noexcept;
operator T*() const volatile noexcept;
operator T*() const noexcept;
T* exchange(T*, memory_order = memory_order_seq_cst) volatile noexcept;
T* exchange(T*, memory_order = memory_order_seq_cst) noexcept;
bool compare_exchange_weak(T*&, T*, memory_order, memory_order) volatile noexcept;
bool compare_exchange_weak(T*&, T*, memory_order, memory_order) noexcept;
bool compare_exchange_strong(T*&, T*, memory_order, memory_order) volatile noexcept;
bool compare_exchange_strong(T*&, T*, memory_order, memory_order) noexcept;
bool compare_exchange_weak(T*&, T*, memory_order = memory_order_seq_cst) volatile noexcept;
bool compare_exchange_weak(T*&, T*, memory_order = memory_order_seq_cst) noexcept;
bool compare_exchange_strong(T*&, T*, memory_order = memory_order_seq_cst) volatile noexcept;
bool compare_exchange_strong(T*&, T*, memory_order = memory_order_seq_cst) noexcept;
T* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst) volatile noexcept;
T* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst) noexcept;
T* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst) volatile noexcept;
T* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst) noexcept;
};

§ 29.5 1149
T* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst) noexcept;

atomic() noexcept = default;
constexpr atomic(T*) noexcept;
atomic(const atomic&) = delete;
atomic(T) operator=(const atomic&) = delete;
atomic(T) 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;

There is a generic class template `atomic<T>`. The type of the template argument `T` shall be trivially copyable (3.9).  
[Note: Type arguments that are not also statically initializable may be difficult to use. —end note]

The semantics of the operations on specializations of `atomic` are defined in 29.6.

Specializations and instantiations of the `atomic` template shall have a deleted copy constructor, a deleted copy assignment operator, and a constexpr value constructor.

There shall be explicit 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 integral type `integral`, the specialization `atomic<integral>` provides additional atomic operations appropriate to integral types. There shall be a specialization `atomic<bool>` which provides the general atomic operations as specified in 29.6.1.

The atomic integral specializations and the specialization `atomic<bool>` shall have standard layout. They shall each have a trivial default constructor and a trivial destructor. They shall each support aggregate initialization syntax.

There shall be pointer partial specializations of the `atomic` class template. These specializations shall have standard layout, trivial default constructors, and trivial destructors. They shall each support aggregate initialization syntax.

There shall be named types corresponding to the integral specializations of `atomic`, as specified in Table 146, and a named type `atomic_bool` corresponding to the specified `atomic<bool>`. Each named type is either a typedef to the corresponding specialization or a base class of the corresponding specialization. If it is a base class, it shall support the same member functions as the corresponding specialization.

There shall be atomic typedefs corresponding to the typedefs in the header `<inttypes.h>` as specified in Table 147.

[Note: The representation of an atomic specialization need not have the same size as its corresponding
Table 146 — atomic integral typedefs

<table>
<thead>
<tr>
<th>Named type</th>
<th>Integral argument type</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_char</td>
<td>char</td>
</tr>
<tr>
<td>atomic_schar</td>
<td>signed char</td>
</tr>
<tr>
<td>atomic_uchar</td>
<td>unsigned char</td>
</tr>
<tr>
<td>atomic_short</td>
<td>short</td>
</tr>
<tr>
<td>atomic_ushort</td>
<td>unsigned short</td>
</tr>
<tr>
<td>atomic_int</td>
<td>int</td>
</tr>
<tr>
<td>atomic_uint</td>
<td>unsigned int</td>
</tr>
<tr>
<td>atomic_long</td>
<td>long</td>
</tr>
<tr>
<td>atomic_ulong</td>
<td>unsigned long</td>
</tr>
<tr>
<td>atomic_llong</td>
<td>long long</td>
</tr>
<tr>
<td>atomic_ullong</td>
<td>unsigned long long</td>
</tr>
<tr>
<td>atomic_char16_t</td>
<td>char16_t</td>
</tr>
<tr>
<td>atomic_char32_t</td>
<td>char32_t</td>
</tr>
<tr>
<td>atomic_wchar_t</td>
<td>wchar_t</td>
</tr>
</tbody>
</table>

argument type. Specializations should have the same size whenever possible, as this reduces the effort required to port existing code. —end note]

29.6 Operations on atomic types [atomics.types.operations]

29.6.1 General operations on atomic types [atomics.types.operations.general]
1 The implementation shall provide the functions and function templates identified as “general operations on atomic types” in 29.2.

2 In the declarations of these functions and function templates, the name atomic-type refers to either atomic<T> or to a named base class for T from Table 146 or inferred from Table 147.

29.6.2 Templated operations on atomic types [atomics.types.operations.templ]
1 The implementation shall declare but not define the function templates identified as “templated operations on atomic types” in 29.2.

29.6.3 Arithmetic operations on atomic types [atomics.types.operations.arith]
1 The implementation shall provide the functions and function template specializations identified as “arithmetic operations on atomic types” in 29.2.

2 In the declarations of these functions and function template specializations, the name integral refers to an integral type and the name atomic-integral refers to either atomic<integral> or to a named base class for integral from Table 146 or inferred from Table 147.

29.6.4 Operations on atomic pointer types [atomics.types.operations.pointer]
1 The implementation shall provide the function template specializations identified as “partial specializations for pointers” in 29.2.

29.6.5 Requirements for operations on atomic types [atomics.types.operations.req]
1 There are only a few kinds of operations on atomic types, though there are many instances on those kinds. This section specifies each general kind. The specific instances are defined in 29.5, 29.6.1, 29.6.3, and 29.6.4.

2 In the following operation definitions:

(2.1) — an A refers to one of the atomic types.
Table 147 — atomic <inttypes.h> typedefs

<table>
<thead>
<tr>
<th>Atomic typedef</th>
<th>&lt;inttypes.h&gt; type</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_int_least8_t</td>
<td>int_least8_t</td>
</tr>
<tr>
<td>atomic_uint_least8_t</td>
<td>uint_least8_t</td>
</tr>
<tr>
<td>atomic_int_least16_t</td>
<td>int_least16_t</td>
</tr>
<tr>
<td>atomic_uint_least16_t</td>
<td>uint_least16_t</td>
</tr>
<tr>
<td>atomic_int_least32_t</td>
<td>int_least32_t</td>
</tr>
<tr>
<td>atomic_uint_least32_t</td>
<td>uint_least32_t</td>
</tr>
<tr>
<td>atomic_int_least64_t</td>
<td>int_least64_t</td>
</tr>
<tr>
<td>atomic_uint_least64_t</td>
<td>uint_least64_t</td>
</tr>
<tr>
<td>atomic_int_fast8_t</td>
<td>int_fast8_t</td>
</tr>
<tr>
<td>atomic_uint_fast8_t</td>
<td>uint_fast8_t</td>
</tr>
<tr>
<td>atomic_int_fast16_t</td>
<td>int_fast16_t</td>
</tr>
<tr>
<td>atomic_uint_fast16_t</td>
<td>uint_fast16_t</td>
</tr>
<tr>
<td>atomic_int_fast32_t</td>
<td>int_fast32_t</td>
</tr>
<tr>
<td>atomic_uint_fast32_t</td>
<td>uint_fast32_t</td>
</tr>
<tr>
<td>atomic_int_fast64_t</td>
<td>int_fast64_t</td>
</tr>
<tr>
<td>atomic_uint_fast64_t</td>
<td>uint_fast64_t</td>
</tr>
<tr>
<td>atomicintptr_t</td>
<td>intptr_t</td>
</tr>
<tr>
<td>atomicuintptr_t</td>
<td>uintptr_t</td>
</tr>
<tr>
<td>atomic_size_t</td>
<td>size_t</td>
</tr>
<tr>
<td>atomicptrdiff_t</td>
<td>ptrdiff_t</td>
</tr>
<tr>
<td>atomicintmax_t</td>
<td>intmax_t</td>
</tr>
<tr>
<td>atomicuintmax_t</td>
<td>uintmax_t</td>
</tr>
</tbody>
</table>

(2.2) — a C refers to its corresponding non-atomic type.

(2.3) — an M refers to type of the other argument for arithmetic operations. For integral atomic types, M is C. For atomic address types, M is std::ptrdiff_t.

(2.4) — the non-member functions not ending in _explicit have the semantics of their corresponding _explicit functions with memory_order arguments of memory_order_seq_cst.

3 [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. Thus, volatile qualified operations on non-volatile objects may be merged under some conditions. —end note]

A::A() noexcept = default;

Effects: leaves the atomic object in an uninitialized state. [Note: These semantics ensure compatibility with C. —end note]

constexpr A::A(C desired) noexcept;

Effects: Initializes the object with the value desired. Initialization is not an atomic operation (1.10). [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

\[29.6.5\]
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:

```c
atomic<int> v = ATOMIC_VAR_INIT(5);
```
—end example]

```c
bool atomic_is_lock_free(const volatile A* object) noexcept;
bool atomic_is_lock_free(const A* object) noexcept;
bool A::is_lock_free() const volatile noexcept;
bool A::is_lock_free() const noexcept;
```

Returns: True if the object’s operations are lock-free, false otherwise.

```c
void atomic_init(volatile A* object, C desired) noexcept;
void atomic_init(A* object, C 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]

```c
void atomic_store(volatile A* object, C desired) noexcept;
void atomic_store(A* object, C desired) noexcept;
void atomic_store_explicit(volatile A* object, C desired, memory_order order) noexcept;
void atomic_store_explicit(A* object, C desired, memory_order order) noexcept;
void A::store(C desired, memory_order order = memory_order_seq_cst) volatile noexcept;
void A::store(C 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 `object` or by `this` with the value of `desired`. Memory is affected according to the value of `order`.

```c
C A::operator=(C desired) volatile noexcept;
C A::operator=(C desired) noexcept;
```

Effects: `store(desired)`

Returns: `desired`

```c
C atomic_load(const volatile A* object) noexcept;
C atomic_load(const A* object) noexcept;
C atomic_load_explicit(const volatile A* object, memory_order) noexcept;
C atomic_load_explicit(const A* object, memory_order) noexcept;
C A::load(memory_order order = memory_order_seq_cst) const volatile noexcept;
C A::load(memory_order_order = memory_order_seq_cst) const noexcept;
```

Requires: The `order` argument shall not be `memory_order_release` nor `memory_order_acq_rel`.

Effects: Memory is affected according to the value of `order`.

Returns: Atomically returns the value pointed to by `object` or by `this`.

```c
A::operator C() const volatile noexcept;
A::operator C() const noexcept;
```
Effects: load()

Returns: The result of load().

C atomic_exchange(volatile A* object, C desired) noexcept;
C atomic_exchange(A* object, C desired) noexcept;
C atomic_exchange_explicit(volatile A* object, C desired, memory_order) noexcept;
C atomic_exchange_explicit(A* object, C desired, memory_order) noexcept;
C A::exchange(C desired, memory_order order = memory_order_seq_cst) volatile noexcept;
C A::exchange(C desired, memory_order order = memory_order_seq_cst) noexcept;

Effects: Atomically replaces the value pointed to by object or by this with desired. Memory is affected according to the value of order. These operations are atomic read-modify-write operations (1.10).

Returns: Atomically returns the value pointed to by object or by this immediately before the effects.

bool atomic_compare_exchange_weak(volatile A* object, C* expected, C desired) noexcept;
bool atomic_compare_exchange_weak(A* object, C* expected, C desired) noexcept;
bool atomic_compare_exchange_strong(volatile A* object, C* expected, C desired) noexcept;
bool atomic_compare_exchange_strong(A* object, C* expected, C desired) noexcept;
bool atomic_compare_exchange_weak_explicit(volatile A* object, C* expected, C desired, memory_order success, memory_order failure) noexcept;
bool atomic_compare_exchange_weak_explicit(A* object, C* expected, C desired, memory_order success, memory_order failure) noexcept;
bool atomic_compare_exchange_strong_explicit(volatile A* object, C* expected, C desired, memory_order success, memory_order failure) noexcept;
bool atomic_compare_exchange_strong_explicit(A* object, C* expected, C desired, memory_order success, memory_order failure) noexcept;
bool A::compare_exchange_weak(Ck expected, C desired, memory_order order = memory_order_seq_cst) volatile noexcept;
bool A::compare_exchange_weak(Ck expected, C desired, memory_order order = memory_order_seq_cst) noexcept;
bool A::compare_exchange_strong(Ck expected, C desired, memory_order order = memory_order_seq_cst) volatile noexcept;
bool A::compare_exchange_strong(Ck expected, C desired, memory_order order = memory_order_seq_cst) noexcept;

Requires: The failure argument shall not be memory_order_release nor memory_order_acq_rel. The failure argument shall be no stronger than the success argument.

Effects: Atomically, compares the contents of the memory pointed to by object or by this for equality with that in expected, and if true, replaces the contents of the memory pointed to by object or by this with that in desired, and if false, updates the contents of the memory in expected with the contents of the memory pointed to by object or by this. Further, 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 the operation returns `true`, these operations are atomic read-modify-write operations (1.10). Otherwise, these operations are atomic load operations.

Returns: The result of the comparison.

[Note: For example, the effect of `atomic_compare_exchange_strong` is]

```c
if (memcmp(object, expected, sizeof(*object)) == 0)
    memcpy(object, &desired, sizeof(*object));
else
    memcpy(expected, object, sizeof(*object));
```

—end note] [Example: the expected use of the compare-and-exchange operations is as follows. The compare-and-exchange operations will update `expected` when another iteration of the loop is needed.

```c
expected = current.load();
do {
    desired = function(expected);
} while (!current.compare_exchange_weak(expected, desired));
```

—end example]

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.

Remark: A weak compare-and-exchange operation may fail spuriously. That is, even when the contents of memory referred to by `expected` and `object` 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. Thus, `compare_exchange_strong` should be used with extreme care. On the other hand, `compare_exchange_weak` should converge rapidly. —end note]

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></td>
</tr>
<tr>
<td>and</td>
<td>&amp;</td>
<td>bitwise and</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Op</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>xor</td>
<td>^</td>
<td>bitwise exclusive or</td>
</tr>
</tbody>
</table>

C `atomic_fetch_key(volatile A* object, M operand) noexcept;`
C `atomic_fetch_key(A* object, M operand) noexcept;`
C `atomic_fetch_key_explicit(volatile A* object, M operand, memory_order order) noexcept;`
C `atomic_fetch_key_explicit(A* object, M operand, memory_order order) noexcept;`
C A::fetch_key(M operand, memory_order order = memory_order_seq_cst) volatile noexcept;
C A::fetch_key(M operand, memory_order order = memory_order_seq_cst) noexcept;

Effects: Atomically replaces the value pointed to by object or by this with the result of the computation applied to the value pointed to by object or by this and the given operand. Memory is affected according to the value of order. These operations are atomic read-modify-write operations (1.10).

Returns: Atomically, the value pointed to by object or by this immediately before the effects.

Remark: For signed integer types, arithmetic is defined to use two’s complement representation. There are no undefined results. For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior.

C A::operator op=(M operand) volatile noexcept;
C A::operator op=(M operand) noexcept;

Effects: fetch_key(operand)

Returns: fetch_key(operand) op operand

C A::operator++(int) volatile noexcept;
C A::operator++(int) noexcept;

Returns: fetch_add(1)

C A::operator--(int) volatile noexcept;
C A::operator--(int) noexcept;

Returns: fetch_sub(1)

C A::operator++() volatile noexcept;
C A::operator++() noexcept;

Effects: fetch_add(1)

Returns: fetch_add(1) + 1

C A::operator--() volatile noexcept;
C A::operator--() noexcept;

Effects: fetch_sub(1)

Returns: fetch_sub(1) - 1

29.7 Flag type and operations

namespace std {

typedef 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;
} atomic_flag;

bool atomic_flag_test_and_set(volatile atomic_flag*) noexcept;
bool atomic_flag_test_and_set(atomic_flag*) noexcept;

§ 29.7
The `atomic_flag` type provides the classic test-and-set functionality. It has two states, set and clear.

Operations on an object of type `atomic_flag` shall be lock-free. [Note: Hence the operations should also be address-free. No other type requires lock-free operations, so the `atomic_flag` type is the minimum hardware-implemented type needed to conform to this International standard. The remaining types can be emulated with `atomic_flag`, though with less than ideal properties. —end note] The `atomic_flag` type shall have standard layout. It shall have a trivial default constructor, a deleted copy constructor, a deleted copy assignment operator, and a trivial destructor.

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:

```c
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.

```c
bool atomic_flag_test_and_set(volatile atomic_flag* object) noexcept;
bool atomic_flag_test_and_set(atomic_flag* object) noexcept;
bool atomic_flag_test_and_set_explicit(volatile atomic_flag* object, memory_order order) noexcept;
bool atomic_flag_test_and_set_explicit(atomic_flag* object, memory_order order) noexcept;
bool atomic_flag::test_and_set(memory_order order = memory_order_seq_cst) volatile noexcept;
bool atomic_flag::test_and_set(memory_order order = memory_order_seq_cst) noexcept;
```

**Effects:** Atomically sets the value pointed to by `object` or by `this` to true. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (1.10).

**Returns:** Atomically, the value of the object immediately before the effects.

```c
void atomic_flag_clear(volatile atomic_flag* object) noexcept;
void atomic_flag_clear(atomic_flag* object) noexcept;
void atomic_flag_clear_explicit(volatile atomic_flag* object, memory_order order) noexcept;
void atomic_flag_clear_explicit(atomic_flag* object, memory_order order) noexcept;
void atomic_flag::clear(memory_order order = memory_order_seq_cst) volatile noexcept;
void atomic_flag::clear(memory_order order = memory_order_seq_cst) noexcept;
```

**Requires:** The `order` argument shall not be `memory_order_consume`, `memory_order_acquire`, nor `memory_order_acq_rel`.

**Effects:** Atomically sets the value pointed to by `object` or by `this` to false. Memory is affected according to the value of `order`.

### 29.8 Fences

This section introduces synchronization primitives called fences. Fences can have acquire semantics, release semantics, or both. A fence with acquire semantics is called an acquire fence. A fence with release semantics is called a release fence.
A release fence \( A \) synchronizes with an acquire fence \( B \) if there exist atomic operations \( X \) and \( Y \), both operating on some atomic object \( M \), such that \( A \) is sequenced before \( X \), \( X \) modifies \( M \), \( Y \) is sequenced before \( B \), and \( Y \) reads the value written by \( X \) or a value written by any side effect in the hypothetical release sequence \( X \) would head if it were a release operation.

A release fence \( A \) synchronizes with an atomic operation \( B \) that performs an acquire operation on an atomic object \( M \) if there exists an atomic operation \( X \) such that \( A \) is sequenced before \( X \), \( X \) modifies \( M \), and \( B \) reads the value written by \( X \) or a value written by any side effect in the hypothetical release sequence \( X \) would head if it were a release operation.

An atomic operation \( A \) that is a release operation on an atomic object \( M \) synchronizes with an acquire fence \( B \) if there exists some atomic operation \( X \) on \( M \) such that \( X \) is sequenced before \( B \) and reads the value written by \( A \) or a value written by any side effect in the release sequence headed by \( A \).

```cpp
extern "C" void atomic_thread_fence(memory_order order) noexcept;
```

**Effects:** depending on the value of \( order \), this operation:

1. has no effects, if \( order == memory_order_relaxed \);
2. is an acquire fence, if \( order == memory_order_acquire || order == memory_order_consume \);
3. is a release fence, if \( order == memory_order_release \);
4. is both an acquire fence and a release fence, if \( order == memory_order_acq_rel \);
5. is a sequentially consistent acquire and release fence, if \( order == memory_order_seq_cst \).

```cpp
extern "C" void atomic_signal_fence(memory_order order) noexcept;
```

**Effects:** Equivalent to \( \text{atomic_thread_fence}(order) \), except that the resulting ordering constraints are established only between a thread and a signal handler executed in the same thread.

**Note:** \( \text{atomic_signal_fence} \) can be used to specify the order in which actions performed by the thread become visible to the signal handler.

**Note:** compiler optimizations and reorderings of loads and stores are inhibited in the same way as with \( \text{atomic_thread_fence} \), but the hardware fence instructions that \( \text{atomic_thread_fence} \) would have inserted are not emitted.
30 Thread support library

30.1 General

The following subclauses describe components to create and manage threads (1.10), perform mutual exclusion, and communicate conditions and values between threads, as summarized in Table 149.

Table 149 — Thread support library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.2 Requirements</td>
<td>&lt;thread&gt;</td>
</tr>
<tr>
<td>30.3 Threads</td>
<td>&lt;mutex&gt;</td>
</tr>
<tr>
<td>30.4 Mutual exclusion</td>
<td>&lt;shared_mutex&gt;</td>
</tr>
<tr>
<td>30.5 Condition variables</td>
<td>&lt;condition_variable&gt;</td>
</tr>
<tr>
<td>30.6 Futures</td>
<td>&lt;future&gt;</td>
</tr>
</tbody>
</table>

30.2 Requirements

30.2.1 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.

30.2.2 Exceptions

Some functions described in this Clause are specified to throw exceptions of type system_error (19.5.6). 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 17.6.5.12.

[ 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 (19.5) and an exception of type system_error is thrown. — end example ]

The error_code reported by such an exception’s code() member function shall compare equal to one of the conditions specified in the function’s error condition element.

30.2.3 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 ]
30.2.4 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 20.12.

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 member 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. Given a duration argument $D_t$, the real-time duration of the timeout is $D_t + D_i + D_m$.

The member 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 $C_a <= C_t$, the waiting function should not time out until Clock::now() returns a time $C_a >= C_t$, 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 (20.12.3).

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 20.12.7 do not throw exceptions. —end note]

30.2.5 Requirements for Lockable types

30.2.5.1 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-

337) All implementations for which standard time units are meaningful must necessarily have a steady clock within their hardware implementation.
pool tasks. —end note] The calling agent is determined by context, e.g. the calling thread that contains the call, and so on.

2 [Note: Some lockable objects are “agent oblivious” in that they work for any execution agent model because they do not determine or store the agent’s ID (e.g., an ordinary spin lock). —end note]

The standard library templates unique_lock (30.4.2.2), lock_guard (30.4.2.1), lock, try_lock (30.4.3), and condition_variable_any (30.5.2) 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]

30.2.5.2 BasicLockable requirements [thread.req.lockable.basic]

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).

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.

30.2.5.3 Lockable requirements [thread.req.lockable.req]

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.try_lock()

Effects: attempts to acquire a lock for the current execution agent without blocking. If an exception is thrown then a lock shall not have been acquired for the current execution agent.

Returns: true if the lock was acquired, false otherwise.

30.2.5.4 TimedLockable requirements [thread.req.lockable.timed]

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 (20.12.5), and abs_time denotes a value of an instantiation of time_point (20.12.6)).

m.try_lock_for(rel_time)

Effects: attempts to acquire a lock for the current execution agent within the relative timeout (30.2.4) 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.

Returns: true if the lock was acquired, false otherwise.

§ 30.2.5.4
m.try_lock_until(abs_time)

Effects: attempts to acquire a lock for the current execution agent before the absolute timeout (30.2.4) 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.

30.2.6 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); }
```

30.3 Threads

30.3 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]

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);
    }
}
```

30.3.1 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]
typedef implementation-defined native_handle_type; // See 30.2.3

// construct/copy/destroy:
thread() noexcept;
template <class F, class ...Args> explicit thread(F&& f, Args&&... args);
~thread();
thread(const thread&) = delete;
thread(thread&) noexcept;
thread& operator=(const thread&) = delete;
thread& operator=(thread&) noexcept;

// members:
void swap(thread&) noexcept;
bool joinable() const noexcept;
void join();
void detach();
int get_id() const noexcept;
native_handle_type native_handle(); // See 30.2.3

// static members:
static unsigned hardware_concurrency() noexcept;
};

30.3.1.1 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;
  
    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 (30.3.1). 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 std::thread object that does not represent threads of execution.

2 thread::id shall be a trivially copyable class (Clause 9). The library may reuse the value of a thread::id of a terminated thread that can no longer be joined.

§ 30.3.1.1
id() noexcept;

Effects: Constructs an object of type id.

Postconditions: The constructed object does not represent a thread of execution.

bool operator==(thread::id x, thread::id y) noexcept;

Returns: true only if x and y represent the same thread of execution or neither x nor y represents a thread of execution.

bool operator!=(thread::id x, thread::id y) noexcept;

Returns: !x == y

bool operator<(thread::id x, thread::id y) noexcept;

Returns: A value such that operator< is a total ordering as described in 25.4.

bool operator<=(thread::id x, thread::id y) noexcept;

Returns: !y < x

bool operator>(thread::id x, thread::id y) noexcept;

Returns: y < x

bool operator>=(thread::id x, thread::id y) noexcept;

Returns: !x < y

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>&& out, thread::id id);

Effects: Inserts an unspecified text representation of id into out. For two objects of type thread::id x and y, if x == y the thread::id objects shall have the same text representation and if x != y the thread::id objects shall have distinct text representations.

Returns: out

template <> struct hash<thread::id>;

The template specialization shall meet the requirements of class template hash (20.9.13).

30.3.1.2 thread constructors

thread() noexcept;

Effects: Constructs a thread object that does not represent a thread of execution.

Postcondition: get_id() == id()

template <class F, class ...Args> explicit thread(F&& f, Args&&... args);
3 Requires: F and each Ti in Args shall satisfy the MoveConstructible requirements. \texttt{INVOKE(DECAY\textunderscore COPY(std::forward<F>(f)), DECAY\textunderscore COPY(std::forward<Args>(args))...) (20.9.2)} shall be a valid expression.

4 Remarks: This constructor shall not participate in overload resolution if \texttt{decay_t<F>} is the same type as \texttt{std::thread}.

5 Effects: Constructs an object of type \texttt{thread}. The new thread of execution executes \texttt{INVOKE(DECAY\textunderscore COPY(std::forward<F>(f)), DECAY\textunderscore COPY(std::forward<Args>(args))...) with the calls to DECAY\textunderscore COPY being evaluated in the constructing thread. Any return value from this invocation is ignored. \texttt{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 \texttt{INVOKE(DECAY\textunderscore COPY(std::forward<F>(f)), DECAY\textunderscore COPY(std::forward<Args>(args))...)} terminates with an uncaught exception, \texttt{std::terminate} shall be called.

6 Synchronization: The completion of the invocation of the constructor synchronizes with the beginning of the invocation of the copy of f.

7 Postconditions: \texttt{get\textunderscore id() != id(). *this} represents the newly started thread.

8 Throws: system\textunderscore error if unable to start the new thread.

9 Error conditions:

10 \texttt{thread(thread&& x) noexcept;}

11 Effects: Constructs an object of type \texttt{thread} from \texttt{x}, and sets \texttt{x} to a default constructed state.

12 Postconditions: \texttt{x.get\textunderscore id() == id()} and \texttt{get\textunderscore id()} returns the value of \texttt{x.get\textunderscore id()} prior to the start of construction.

30.3.1.3 thread destructor

\texttt{~thread();}

1 If \texttt{joinable()}, calls \texttt{std::terminate().} Otherwise, has no effects. \texttt{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 raised. Thus the programmer must ensure that the destructor is never executed while the thread is still joinable. — end note]}

30.3.1.4 thread assignment

\texttt{thread\& operator=(thread\&\& x) noexcept;}

1 Effects: If \texttt{joinable()}, calls \texttt{std::terminate().} Otherwise, assigns the state of \texttt{x} to \texttt{*this} and sets \texttt{x} to a default constructed state.

2 Postconditions: \texttt{x.get\textunderscore id() == id()} and \texttt{get\textunderscore id()} returns the value of \texttt{x.get\textunderscore id()} prior to the assignment.

3 Returns: \texttt{*this}

30.3.1.5 thread members

\texttt{void swap(thread\& x) noexcept;}

1 Effects: Swaps the state of \texttt{*this} and \texttt{x.}
bool joinable() const noexcept;

   Returns: get_id() != id()

void join();

   Requires: joinable() is true.
   Effects: Blocks until the thread represented by *this has completed.
   Synchronization: The completion of the thread represented by *this synchronizes with (1.10) 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 (30.2.2).
   Error conditions:
   (8.1) — resource_deadlock_would_occur — if deadlock is detected or this->get_id() == std::this_thread::get_id().
   (8.2) — no_such_process — if the thread is not valid.
   (8.3) — invalid_argument — if the thread is not joinable.

void detach();

   Requires: joinable() is true.
   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.

   Postcondition: get_id() == id().
   Throws: system_error when an exception is required (30.2.2).
   Error conditions:
   (13.1) — no_such_process — if the thread is not valid.
   (13.2) — 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.

30.3.1.6 thread static members

unsigned hardware_concurrency() noexcept;

   Returns: The number of hardware thread contexts. [ Note: This value should only be considered to be a hint. — end note ] If this value is not computable or well defined an implementation should return 0.

30.3.1.7 thread specialized algorithms

void swap(thread& x, thread& y) noexcept;

   Effects: x.swap(y)
30.3.2 Namespace this_thread

```cpp
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;`

1 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;`

2 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);`

4 Effects: Blocks the calling thread for the absolute timeout (30.2.4) specified by `abs_time`.

Synchronization: None.

Throws: Timeout-related exceptions (30.2.4).

`template <class Rep, class Period>
  void sleep_for(const chrono::duration<Rep, Period>& rel_time);`

7 Effects: Blocks the calling thread for the relative timeout (30.2.4) specified by `rel_time`.

Synchronization: None.

Throws: Timeout-related exceptions (30.2.4).

30.4 Mutual exclusion

`namespace std {` synthesis

```cpp
class mutex;`n
  class recursive_mutex;
  class timed_mutex;
  class recursive_timed_mutex;

  struct defer_lock_t { };`n
  struct try_to_lock_t { };`n
  struct adopt_lock_t { };`n

  constexpr defer_lock_t defer_lock { };`n
  constexpr try_to_lock_t try_to_lock { };`n
  constexpr adopt_lock_t adopt_lock { };`n
```n

§ 30.4
```cpp
template <class Mutex> class lock_guard;
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 {
  constexpr once_flag() noexcept;
  once_flag(const once_flag&) = delete;
  once_flag& operator=(const once_flag&) = delete;
};

template<class Callable, class ...Args>
  void call_once(once_flag& flag, Callable& func, Args&... args);
```

Header `<shared_mutex>` synopsis

```cpp
namespace std {
  class shared_timed_mutex;
  template <class Mutex> class shared_lock;
  template <class Mutex>
    void swap(shared_lock<Mutex>& x, shared_lock<Mutex>& y) noexcept;
}
```

30.4.1 Mutex requirements [thread.mutex.requirements]

30.4.1.1 In general [thread.mutex.requirements.general]

A mutex object facilitates protection against data races and allows safe synchronization of data between execution agents (30.2.5). 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.

30.4.1.2 Mutex types [thread.mutex.requirements.mutex]

The mutex types are the standard library types `std::mutex`, `std::recursive_mutex`, `std::timed_mutex`, `std::recursive_timed_mutex`, and `std::shared_timed_mutex`. They shall meet the requirements set out in this section. In this description, `m` denotes an object of a mutex type.

The mutex types shall meet the Lockable requirements (30.2.5.3).

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:

(4.1) — `resource_unavailable_try_again` — if any native handle type manipulated is not available.

(4.2) — `operation_not_permitted` — if the thread does not have the privilege to perform the operation.

(4.3) — `device_or_resource_busy` — if any native handle type manipulated is already locked.
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 (1.10). 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 (1.10) 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:

\textbf{Requires:} If \texttt{m} is of type \texttt{std::mutex}, \texttt{std::timed_mutex}, or \texttt{std::shared_timed_mutex}, the calling thread does not own the mutex.

\textbf{Effects:} Blocks the calling thread until ownership of the mutex can be obtained for the calling thread.

\textbf{Postcondition:} The calling thread owns the mutex.

\textbf{Return type:} \texttt{void}

\textbf{Synchronization:} Prior \texttt{unlock()} operations on the same object shall \textit{synchronize with} (1.10) this operation.

\textbf{Throws:} \texttt{system\_error} when an exception is required (30.2.2).

\textbf{Error conditions:}

\begin{itemize}
\item \texttt{operation\_not\_permitted} — if the thread does not have the privilege to perform the operation.
\item \texttt{resource\_deadlock\_would\_occur} — if the implementation detects that a deadlock would occur.
\item \texttt{device\_or\_resource\_busy} — if the mutex is already locked and blocking is not possible.
\end{itemize}

The expression \texttt{m.try\_lock()} shall be well-formed and have the following semantics:

\textbf{Requires:} If \texttt{m} is of type \texttt{std::mutex}, \texttt{std::timed_mutex}, or \texttt{std::shared_timed_mutex}, the calling thread does not own the mutex.

\textbf{Effects:} Attempts to obtain ownership of the mutex for the calling thread without blocking. If ownership is not obtained, there is no effect and \texttt{try\_lock()} immediately returns. An implementation may fail to obtain the lock even if it is not held by any other thread. [Note: This spurious failure is normally uncommon, but allows interesting implementations based on a simple compare and exchange (Clause 29). — end note] An implementation should ensure that \texttt{try\_lock()} does not consistently return \texttt{false} in the absence of contending mutex acquisitions.

\textbf{Return type:} \texttt{bool}

\textbf{Returns:} \texttt{true} if ownership of the mutex was obtained for the calling thread, otherwise \texttt{false}.

\textbf{Synchronization:} If \texttt{try\_lock()} returns \texttt{true}, prior \texttt{unlock()} operations on the same object \textit{synchronize with} (1.10) 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]

\textbf{Throws:} Nothing.

The expression \texttt{m.unlock()} shall be well-formed and have the following semantics:

\textbf{Requires:} The calling thread shall own the mutex.

\textbf{Effects:} Releases the calling thread’s ownership of the mutex.

\textbf{Return type:} \texttt{void}

\textbf{Synchronization:} This operation synchronizes with (1.10) subsequent lock operations that obtain ownership on the same object.

\textbf{Throws:} Nothing.
30.4.1.2.1 Class mutex

namespace std {
    class mutex {
    public:
        constexpr mutex() noexcept;
        ~mutex();

        mutex(const mutex&) = delete;
        mutex& operator=(const mutex&) = delete;

        void lock();
        bool try_lock();
        void unlock();

        typedef implementation-defined native_handle_type; // See 30.2.3
        native_handle_type native_handle(); // See 30.2.3
    }
}

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 the Mutex requirements (30.4.1). It shall be a standard-layout class (Clause 9).

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.

30.4.1.2.2 Class recursive_mutex

namespace std {
    class recursive_mutex {
    public:
        recursive_mutex();
        ~recursive_mutex();

        recursive_mutex(const recursive_mutex&) = delete;
        recursive_mutex& operator=(const recursive_mutex&) = delete;

        void lock();
        bool try_lock() noexcept;
        void unlock();

        typedef implementation-defined native_handle_type; // See 30.2.3
        native_handle_type native_handle(); // See 30.2.3
    }
}
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 the Mutex requirements (30.4.1). It shall be a standard-layout class (Clause 9).

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:

- it destroys a `recursive_mutex` object owned by any thread or
- a thread terminates while owning a `recursive_mutex` object.

### 30.4.1.3 Timed mutex types

The `timed_mutex` types are the standard library types `std::timed_mutex`, `std::recursive_timed_mutex`, and `std::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` (20.12.5), and `abs_time` denotes an object of an instantiation of `time_point` (20.12.6).

The timed mutex types shall meet the `TimedLockable` requirements (30.2.5.4).

The expression `m.try_lock_for(rel_time)` shall be well-formed and have the following semantics:

**Requires:** If `m` is of type `std::timed_mutex` or `std::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 (30.2.4) 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 (1.10) this operation.

**Throws:** Timeout-related exceptions (30.2.4).

The expression `m.try_lock_until(abs_time)` shall be well-formed and have the following semantics:

**Requires:** If `m` is of type `std::timed_mutex` or `std::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 (30.2.4) 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...]

§ 30.4.1.3
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 (1.10) this operation.

Throws: Timeout-related exceptions (30.2.4).

30.4.1.3.1 Class timed_mutex

namespace std {
    class timed_mutex {
        public:
            timed_mutex();
            ~timed_mutex();

            timed_mutex(const timed_mutex&) = delete;
            timed_mutex& operator=(const timed_mutex&) = delete;

            void lock(); // blocking
            bool try_lock();

            template <class Rep, class Period>
            bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);

            template <class Clock, class Duration>
            bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);

            void unlock();

            typedef implementation-defined native_handle_type; // See 30.2.3

            native_handle_type native_handle(); // See 30.2.3
    };
}

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 TimedMutex requirements (30.4.1.3). It shall be a standard-layout class (Clause 9).

3 The behavior of a program is undefined if:

(3.1) — it destroys a timed_mutex object owned by any thread,

(3.2) — a thread that owns a timed_mutex object calls lock(), try_lock(), try_lock_for(), or try_lock_until() on that object, or

(3.3) — a thread terminates while owning a timed_mutex object.

30.4.1.3.2 Class recursive_timed_mutex

namespace std {
    class recursive_timed_mutex {
        public:
            recursive_timed_mutex();

        § 30.4.1.3.2 1172
    };
}
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 TimedMutex requirements (30.4.1.3). It shall be a standard-layout class (Clause 9).

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:

(4.1) — it destroys a `recursive_timed_mutex` object owned by any thread, or

(4.2) — a thread terminates while owning a `recursive_timed_mutex` object.

### 30.4.1.4 Shared timed mutex types

The standard library type `std::shared_timed_mutex` is a shared timed mutex type. Shared timed mutex types shall meet the requirements of timed mutex types (30.4.1.3), and additionally shall meet the requirements set out below. In this description, `m` denotes an object of a mutex type, `rel_type` denotes an object of an instantiation of `duration` (20.12.5), and `abs_time` denotes an object of an instantiation of `time_point` (20.12.6).

In addition to the exclusive lock ownership mode specified in 30.4.1.2, 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.

3 The expression `m.lock_shared()` shall be well-formed and have the following semantics:

   **Requires:** The calling thread has no ownership of the mutex.

   **Effects:** Blocks the calling thread until shared ownership of the mutex can be obtained for the calling thread. If an exception is thrown then a shared lock shall not have been acquired for the current thread.

   **Postcondition:** The calling thread has a shared lock on the mutex.

   **Return type:** `void`.

   **Synchronization:** Prior `unlock()` operations on the same object shall synchronize with (1.10) this operation.

   **Throws:** `system_error` when an exception is required (30.2.2).

   **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.
   - `device_or_resource_busy` — if the mutex is already locked and blocking is not possible.

9 The expression `m.unlock_shared()` shall be well-formed and have the following semantics:

   **Requires:** The calling thread shall hold a shared lock on the mutex.

   **Effects:** Releases a shared lock on the mutex held by the calling thread.

   **Return type:** `void`.

   **Synchronization:** This operation synchronizes with (1.10) subsequent `lock()` operations that obtain ownership on the same object.

   **Throws:** Nothing.

11 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 (1.10) this operation.

   **Throws:** Nothing.

17 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 time-out (30.2.4) 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 (1.10) this operation.

Throws: Timeout-related exceptions (30.2.4).

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 (30.2.4) 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 (1.10) this operation.

Throws: Timeout-related exceptions (30.2.4).

30.4.1.4.1 Class shared_timed_mutex

namespace std {
    class shared_timed_mutex {
        public:
            shared_timed_mutex();
            ~shared_timed_mutex();

            shared_timed_mutex(const shared_timed_mutex&) = delete;
            shared_timed_mutex& operator=(const shared_timed_mutex&) = delete;

        // Exclusive ownership
        void lock(); // blocking
        bool try_lock();
        template <class Rep, class Period>
            bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
        template <class Clock, class Duration>
            bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
            void unlock();

        // Shared ownership
        void lock_shared(); // blocking
        bool try_lock_shared();
        template <class Rep, class Period>
            bool try_lock_shared_for(const chrono::duration<Rep, Period>& rel_time);

    } // class shared_timed_mutex

} // namespace std
template <class Clock, class Duration>
bool
try_lock_shared_until(const chrono::time_point<Clock, Duration>& abs_time);
void unlock_shared();
};

The class `shared_timed_mutex` provides a non-recursive mutex with shared ownership semantics.

The class `shared_timed_mutex` shall satisfy all of the `SharedTimedMutex` requirements (30.4.1.4). It shall be a standard-layout class (Clause 9).

The behavior of a program is undefined if:

1. it destroys a `shared_timed_mutex` object owned by any thread,
2. a thread attempts to recursively gain any ownership of a `shared_timed_mutex`, or
3. a thread terminates while possessing any ownership of a `shared_timed_mutex`.

30.4.2 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

    constexpr defer_lock_t defer_lock { };      
    constexpr try_to_lock_t try_to_lock { };    
    constexpr adopt_lock_t adopt_lock { };      
}
```

30.4.2.1 Class template `lock_guard`

```cpp
namespace std {
    template <class Mutex>
    class lock_guard {
public:
        typedef Mutex mutex_type;

        explicit lock_guard(mutex_type& m);
        lock_guard(mutex_type& m, adopt_lock_t);
        ~lock_guard();

        lock_guard(lock_guard const&) = delete;
        lock_guard& operator=(lock_guard const&) = delete;
    }
}
```
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 (3.8). 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 (30.2.5.2).

```cpp
explicit lock_guard(mutex_type& m);
```

`Requires:` If `mutex_type` is not a recursive mutex, the calling thread does not own the mutex `m`.

`Effects:` `m.lock()`

`Postcondition:` `&pm == &m`

```cpp
lock_guard(mutex_type& m, adopt_lock_t);
```

`Requires:` The calling thread owns the mutex `m`.

`Postcondition:` `&pm == &m`

`Throws:` Nothing.

```cpp
~lock_guard();
```

`Effects:` `pm.unlock()`

30.4.2.2 Class template `unique_lock` [thread.lock.unique]

```cpp
namespace std {
    template <class Mutex> class unique_lock {
    public:
        typedef Mutex mutex_type;

        // 30.4.2.2.1, construct/copy/destroy:
        unique_lock() noexcept;
        explicit unique_lock(mutex_type& m);
        unique_lock(mutex_type& m, defer_lock_t) noexcept;
        unique_lock(mutex_type& m, try_to_lock_t);
        unique_lock(mutex_type& m, adopt_lock_t);
        template <class Clock, class Duration>
        unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);
        template <class Rep, class Period>
        unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);
        ~unique_lock();

        unique_lock(unique_lock const&) = delete;
        unique_lock& operator=(unique_lock const&) = delete;

        unique_lock(unique_lock&& u) noexcept;
        unique_lock& operator=(unique_lock&& u);

        // 30.4.2.2.2, locking:
        void lock();
        bool try_lock();
    }
}
```
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 (3.8) of the `unique_lock` object. The supplied `Mutex` type shall meet the `BasicLockable` requirements (30.2.5.2).

1

2

[Note: `unique_lock<Mutex>` meets the `BasicLockable` requirements. If `Mutex` meets the `Lockable` requirements (30.2.5.3), `unique_lock<Mutex>` also meets the `Lockable` requirements; if `Mutex` meets the `TimedLockable` requirements (30.2.5.4), `unique_lock<Mutex>` also meets the `TimedLockable` requirements. —end note]

30.4.2.2.1 `unique_lock` constructors, destructor, and assignment

    ```
    template <class Mutex>
    void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;
    ```

1

2

3

4

5

6

7

§ 30.4.2.2.1
unique_lock(mutex_type& m, try_to_lock_t);

Requires: The supplied Mutex type shall meet the Lockable requirements (30.2.5.3). 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 == &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 own the mutex.

Effects: Constructs an object of type unique_lock.

Postconditions: pm == &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 (30.2.5.4).

Effects: Constructs an object of type unique_lock and calls m.try_lock_until(abs_time).

Postconditions: pm == &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 (30.2.5.4).

Effects: Constructs an object of type unique_lock and calls m.try_lock_for(rel_time).

Postconditions: pm == &m and owns == res, where res is the value returned by the call to m.try_lock_for(rel_time).

unique_lock(unique_lock&& u) noexcept;

Postconditions: pm == u_p.pm and owns == u_p.owns (where u_p is the state of u just prior to this construction), u.pm == 0 and u.owns == false.

unique_lock& operator=(unique_lock&& u);

Effects: If owns calls pm->unlock().

Postconditions: pm == u_p.pm and owns == u_p.owns (where u_p is the state of u just prior to this construction), u.pm == 0 and u.owns == false.

[ Note: With a recursive mutex it is possible for both *this and u to own the same mutex before the assignment. In this case, *this will own the mutex after the assignment and u will not. — end note ]

Throws: Nothing.

~unique_lock();

Effects: If owns calls pm->unlock().
30.4.2.2.2 unique_lock locking

```cpp
void lock();
```

1. **Effects:** pm->lock()
2. **Postcondition:** owns == true
3. **Throws:** Any exception thrown by pm->lock(). system_error if an exception is required (30.2.2). system_error with an error condition of operation_not_permitted if pm is 0. system_error with an error condition of resource_deadlock_would_occur if on entry owns is true.

```cpp
bool try_lock();
```

4. **Requires:** The supplied Mutex shall meet the Lockable requirements (30.2.5.3).
5. **Effects:** pm->try_lock()
6. **Returns:** The value returned by the call to try_lock().
7. **Postcondition:** owns == res, where res is the value returned by the call to try_lock().
8. **Throws:** Any exception thrown by pm->try_lock(). system_error if an exception is required (30.2.2). system_error with an error condition of operation_not_permitted if pm is 0. system_error with an error condition of 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);
```

9. **Requires:** The supplied Mutex type shall meet the TimedLockable requirements (30.2.5.4).
10. **Effects:** pm->try_lock_until(abs_time)
11. **Returns:** The value returned by the call to try_lock_until(abs_time).
12. **Postcondition:** owns == res, where res is the value returned by the call to try_lock_until(abs_time).
13. **Throws:** Any exception thrown by pm->try_lock_until(). system_error if an exception is required (30.2.2). system_error with an error condition of operation_not_permitted if pm is 0. system_error with an error condition of resource_deadlock_would_occur if on entry owns is true.

```cpp
template <class Rep, class Period>
bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
```

14. **Requires:** The supplied Mutex type shall meet the TimedLockable requirements (30.2.5.4).
15. **Effects:** pm->try_lock_for(rel_time).
16. **Returns:** The value returned by the call to try_lock_until(rel_time).
17. **Postcondition:** owns == res, where res is the value returned by the call to try_lock_for(rel_time).
18. **Throws:** Any exception thrown by pm->try_lock_for(). system_error if an exception is required (30.2.2). system_error with an error condition of operation_not_permitted if pm is 0. system_error with an error condition of resource_deadlock_would_occur if on entry owns is true.

```cpp
void unlock();
```

19. **Effects:** pm->unlock()
20. **Postcondition:** owns == false
21. **Throws:** system_error when an exception is required (30.2.2).
22. **Error conditions:**

   (22.1) operation_not_permitted — if on entry owns is false.
30.4.2.2.3 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: x.swap(y)

30.4.2.2.4 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

30.4.2.3 Class template shared_lock

namespace std {

template <class Mutex>
class shared_lock {
public:
    typedef Mutex mutex_type;

    // Shared locking
    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(shared_lock const&) = delete;
    shared_lock& operator=(shared_lock const&) = delete;

    shared_lock(shared_lock&& u) noexcept;
    shared_lock& operator=(shared_lock&& u) noexcept;
}
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();

// Setters
void swap(shared_lock& u) noexcept;
mutex_type* release() noexcept;

// Getters
bool owns_lock() const noexcept;
explicit operator bool () const noexcept;
mutex_type* mutex() const noexcept;

private:
  mutex_type* pm; // exposition only
  bool owns; // exposition only
};

// std

An object of type shared_lock controls the shared ownership of a lockable object within a scope. Shared
ownership of the lockable object may be acquired at construction or after construction, and may be trans-
ferred, 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 (3.8) of the shared_lock object.
The supplied Mutex type shall meet the shared mutex requirements (30.4.1.4).

[ Note: shared_lock<Mutex> meets the TimedLockable requirements (30.2.5.4). — end note ]

30.4.2.3.1 shared_lock constructors, destructor, and assignment

shared_lock() noexcept;
1 Effects: Constructs an object of type shared_lock.
2 Postconditions: pm == nullptr and owns == false.

explicit shared_lock(mutex_type& m);
3 Requires: The calling thread does not own the mutex for any ownership mode.
4 Effects: Constructs an object of type shared_lock and calls m.lock_shared().
5 Postconditions: pm == &m and owns == true.

shared_lock(mutex_type& m, defer_lock_t) noexcept;
6 Effects: Constructs an object of type shared_lock.
7 Postconditions: pm == &m and owns == false.

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 == &m and owns == res where res is the value returned by the call to `m.try_lock_shared()`.

```cpp
class 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 == &m and owns == true.

```cpp
template <class Clock, class Duration>
class 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 == &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>
class 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 == &m and owns == res where res is the value returned by the call to `m.try_lock_shared_for(rel_time)`.

```cpp
~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.

30.4.2.3.2 shared_lock locking [thread.lock.shared.locking]

```cpp
void lock();
```

Effects: pm->lock_shared().

Postconditions: owns == true.

Throws: Any exception thrown by pm->lock_shared(). `system_error` if an exception is required (30.2.2). `system_error` with an error condition of `operation_not_permitted` if pm is nullptr. `system_error` with an error condition of `resource_deadlock_would_occur` if on entry owns is true.
bool try_lock();

Effects: 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 if an exception is required (30.2.2). system_error with an error condition of operation_not_permitted if pm is nullptr. system_error with an error condition of resource_deadlock_would_occur if on entry owns is true.

template <class Clock, class Duration>
bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);

Effects: 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 if an exception is required (30.2.2). system_error with an error condition of operation_not_permitted if pm is nullptr. system_error with an error condition of resource_deadlock_would_occur if on entry owns is true.

template <class Rep, class Period>
bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);

Effects: 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 if an exception is required (30.2.2). system_error with an error condition of operation_not_permitted if pm is nullptr. system_error with an error condition of resource_deadlock_would_occur if on entry owns is true.

void unlock();

Effects: pm→unlock_shared().

Postconditions: owns == false.

Throws: system_error when an exception is required (30.2.2).

Error conditions:

— operation_not_permitted — if on entry owns is false.

30.4.2.3.3 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: x.swap(y).

30.4.2.3.4 shared_lock observers

bool owns_lock() const noexcept;
  Returns: owns.

explicit operator bool () const noexcept;
  Returns: owns.

mutex_type* mutex() const noexcept;
  Returns: pm.

30.4.3 Generic locking algorithms

template <class L1, class L2, class... L3> int try_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.
  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 0-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.
  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.
  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().

30.4.4 Call once

The class once_flag is an opaque data structure that call_once uses to initialize data without causing a data race or deadlock.

30.4.4.1 Struct once_flag

constexpr once_flag() noexcept;
  Effects: Constructs an object of type once_flag.

  Synchronization: The construction of a once_flag object is not synchronized.

  Postcondition: 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.
30.4.4.2 Function call_once

\[\text{template\langle class Callable, class ...Args\rangle} \]
\[\text{void call_once(once_flag& flag, Callable&& func, Args&&... args);}\]

1 Requires: Callable and each Ti in Args shall satisfy the MoveConstructible requirements. \text{INVOKED DECAY_COPY( std::forward<Callable>(func)), DECAY_COPY (std::forward<Args>(args))...} (20.9.2) shall be a valid expression.

2 Effects: An execution of call_once that does not call its func is a passive execution. An execution of call_once that calls its func is an active execution. An active execution shall call \text{INVOKED DECAY_COPY( std::forward<Callable>(func)), DECAY_COPY (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. \[\text{Note: passive executions allow other threads to reliably observe the results produced by the earlier returning execution. — end note}\]

3 Synchronization: For any given once_flag: all active executions occur in a total order; completion of an active execution synchronizes with (1.10) the start of the next one in this total order; and the returning execution synchronizes with the return from all passive executions.

4 Throws: \text{system_error} when an exception is required (30.2.2), or any exception thrown by func.

5 \[\text{Example:}\]
\[\text{// global flag, regular function}\]
\[\text{void init();}\]
\[\text{std::once_flag flag;}\]
\[\text{void f() {}\]
\[\text{std::call_once(flag, init);}\]
\[\text{}}\]
\[\text{// function static flag, function object}\]
\[\text{struct initializer {}\]
\[\text{void operator(){};}\]
\[\text{}};}\]
\[\text{void g() {}\]
\[\text{static std::once_flag flag2;}\]
\[\text{std::call_once(flag2, initializer());}\]
\[\text{}}\]
\[\text{// object flag, member function}\]
\[\text{class information {}\]
\[\text{std::once_flag verified;}\]
\[\text{void verifier();}\]
\[\text{public:}\]
\[\text{void verify() { std::call_once(verified, &information::verifier, *this); } }\]
\[\text{}};}\]

— end example]

30.5 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 \text{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.

### Header `condition_variable` synopsis

```cpp
namespace std {
    class condition_variable;
    class condition_variable_any;

    void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);

    enum class cv_status { no_timeout, timeout };
}
```

6 Requires: `lk` is locked by the calling thread and either
   1. no other thread is waiting on `cond`, or
   2. `lk.mutex()` returns the same value for each of the lock arguments supplied by all concurrently waiting (via `wait`, `wait_for`, or `wait_until`) threads.

7 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();
   ```

8 Synchronization: The implied `lk.unlock()` call is sequenced after the destruction of all objects with thread storage duration associated with the current thread.

9 Note: The supplied lock will be held until the thread exits, and care must 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.

10 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`. 

§ 30.5
namespace std {
    class condition_variable {
        public:

            condition_variable();
            ~condition_variable();

            condition_variable(const condition_variable&) = delete;
            condition_variable& operator=(const condition_variable&) = delete;

            void notify_one() noexcept;
            void notify_all() noexcept;
            void wait(unique_lock<mutex>& lock);
            template <class Predicate>
                void wait(unique_lock<mutex>& lock, Predicate pred);
            template <class Clock, class Duration>
                cv_status wait_until(unique_lock<mutex>& lock,
                    const chrono::time_point<Clock, Duration>& abs_time);
            template <class Clock, class Duration, class Predicate>
                bool wait_until(unique_lock<mutex>& lock,
                                const chrono::time_point<Clock, Duration>& abs_time,
                                Predicate pred);

            template <class Rep, class Period>
                cv_status wait_for(unique_lock<mutex>& lock,
                                    const chrono::duration<Rep, Period>& rel_time);
            template <class Rep, class Period, class Predicate>
                bool wait_for(unique_lock<mutex>& lock,
                              const chrono::duration<Rep, Period>& rel_time,
                              Predicate pred);

            typedef implementation-defined native_handle_type; // See 30.2.3
            native_handle_type native_handle(); // See 30.2.3
        }
    }

1 The class condition_variable shall be a standard-layout class (Clause 9).

    condition_variable();

2 Effects: Constructs an object of type condition_variable.

3 Throws: system_error when an exception is required (30.2.2).

4 Error conditions:

    — resource_unavailable_try_again — if some non-memory resource limitation prevents initialization.

    ~condition_variable();

5 Requires: There shall be no thread blocked on *this. [Note: That is, all threads shall have been notified; they may subsequently block on the lock specified in the wait. This relaxes the usual rules, which would have required all wait calls to happen before destruction. Only the notification to unblock the wait must happen before destruction. The user must 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 \texttt{wait, wait\_for, or wait\_until} that take a predicate.

\textit{— end note}]

\textbf{Effects}: Destroys the object.

\begin{verbatim}
void notify_one() noexcept;
\end{verbatim}

\textbf{Effects}: If any threads are blocked waiting for \texttt{*this}, unblocks one of those threads.

\begin{verbatim}
void notify_all() noexcept;
\end{verbatim}

\textbf{Effects}: Unblocks all threads that are blocked waiting for \texttt{*this}.

\begin{verbatim}
void wait(unique_lock<mutex>& lock);
\end{verbatim}

\textbf{Requires}: \texttt{lock.owns\_lock()} is \texttt{true} and \texttt{lock.mutex()} is locked by the calling thread, and either

(9.1) no other thread is waiting on this \texttt{condition\_variable} object or

(9.2) \texttt{lock.mutex()} returns the same value for each of the \texttt{lock} arguments supplied by all concurrently waiting (via \texttt{wait, wait\_for, or wait\_until}) threads.

\textbf{Effects}:

(9.1) Atomically calls \texttt{lock.unlock()} and blocks on \texttt{*this}.

(9.2) When unblocked, calls \texttt{lock.lock()} (possibly blocking on the lock), then returns.

(9.3) The function will unblock when signaled by a call to \texttt{notify\_one()} or a call to \texttt{notify\_all()}, or spuriously.

\textbf{Remarks}: If the function fails to meet the postcondition, \texttt{std::terminate()} shall be called (15.5.1).

\textit{[Note: This can happen if the re-locking of the mutex throws an exception. — end note]}

\textbf{Postcondition}: \texttt{lock.owns\_lock()} is \texttt{true} and \texttt{lock.mutex()} is locked by the calling thread.

\textbf{Throws}: Nothing.

\begin{verbatim}
template <class Predicate>
void wait(unique_lock<mutex>& lock, Predicate pred);
\end{verbatim}

\textbf{Requires}: \texttt{lock.owns\_lock()} is \texttt{true} and \texttt{lock.mutex()} is locked by the calling thread, and either

(14.1) no other thread is waiting on this \texttt{condition\_variable} object or

(14.2) \texttt{lock.mutex()} returns the same value for each of the \texttt{lock} arguments supplied by all concurrently waiting (via \texttt{wait, wait\_for, or wait\_until}) threads.

\textbf{Effects}: Equivalent to:

\begin{verbatim}
while (!pred())
  wait(lock);
\end{verbatim}

\textbf{Remarks}: If the function fails to meet the postcondition, \texttt{std::terminate()} shall be called (15.5.1).

\textit{[Note: This can happen if the re-locking of the mutex throws an exception. — end note]}

\textbf{Postcondition}: \texttt{lock.owns\_lock()} is \texttt{true} and \texttt{lock.mutex()} is locked by the calling thread.

\textbf{Throws}: Timeout-related exceptions (30.2.4) or any exception thrown by \texttt{pred}.

\begin{verbatim}
template <class Clock, class Duration>
 cv\_status wait\_until(unique_lock<mutex>& lock,
 const chrono::time\_point<
 Clock, Duration>& abs\_time);
\end{verbatim}

\textbf{Requires}: \texttt{lock.owns\_lock()} is \texttt{true} and \texttt{lock.mutex()} is locked by the calling thread, and either

\section*{30.5.1}
— 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:**

(20.1) — Atomically calls `lock.unlock()` and blocks on `*this`.
(20.2) — When unblocked, calls `lock.lock()` (possibly blocking on the lock), then returns.
(20.3) — The function will unblock when signaled by a call to `notify_one()`, a call to `notify_all()`,
expiration of the absolute timeout (30.2.4) specified by `abs_time`, or spuriously.
(20.4) — 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, `std::terminate()` shall be called (15.5.1).
[Note: This can happen if the re-locking of the mutex throws an exception. — end note]

**Postcondition:** `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

**Returns:** `cv_status::timeout` if the absolute timeout (30.2.4) specified by `abs_time` expired, otherwise
`cv_status::no_timeout`.

**Throws:** Timeout-related exceptions (30.2.4).

```cpp
template <class Rep, class Period>
std::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
(25.1) — no other thread is waiting on this `condition_variable` object or
(25.2) — `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:

```cpp
return wait_until(lock, chrono::steady_clock::now() + rel_time);
```

**Returns:** `cv_status::timeout` if the relative timeout (30.2.4) specified by `rel_time` expired, otherwise
`cv_status::no_timeout`.

**Remarks:** If the function fails to meet the postcondition, `std::terminate()` shall be called (15.5.1).
[Note: This can happen if the re-locking of the mutex throws an exception. — end note]

**Postcondition:** `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

**Throws:** Timeout-related exceptions (30.2.4).

```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
(31.1) — no other thread is waiting on this `condition_variable` object or
(31.2) — `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())
    if (wait_until(lock, abs_time) == cv_status::timeout)
        return pred();
    return true;

Remarks: If the function fails to meet the postcondition, std::terminate() shall be called (15.5.1).
[Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Postcondition: 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 (30.2.4) 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
— 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, std::terminate() shall be called (15.5.1).
[Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Postcondition: 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 (30.2.4) or any exception thrown by pred.

30.5.2 Class condition_variable_any [thread.condition.condvarany]

A Lock type shall meet the BasicLockable requirements (30.2.5.2). [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 must 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 (30.2.2).

Error conditions:

(4.1) resource_unavailable_try_again — if some non-memory resource limitation prevents initialization.

(4.2) operation_not_permitted — if the thread does not have the privilege to perform the operation.

~condition_variable_any();

Requires: There shall be no thread blocked on *this. [Note: That is, all threads shall have been notified; they may subsequently block on the lock specified in the wait. This relaxes the usual rules, which would have required all wait calls to happen before destruction. Only the notification to unblock the wait must happen before destruction. The user must take care to ensure that no threads wait on *this once the destructor has been started, especially when the waiting threads are calling the wait functions in a loop or using the overloads of wait, wait_for, or wait_until that take a predicate. — end note]

Effects: Destroys the object.

void notify_one() noexcept;

Effects: If any threads are blocked waiting for *this, unblocks one of those threads.

void notify_all() noexcept;

Effects: Unblocks all threads that are blocked waiting for *this.

template <class Lock>
void wait(Lock& lock);

Note: if any of the wait functions exits via an exception, it is unspecified whether the Lock is held. One can use a Lock type that allows to query that, such as the unique_lock wrapper.

Effects:

§ 30.5.2
— Atomically calls `lock.unlock()` and blocks on `*this`.
— When unblocked, calls `lock.lock()` (possibly blocking on the lock) and returns.
— 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, `std::terminate()` shall be called (15.5.1).
[Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Postcondition: `lock` is locked by the calling thread.

Throws: Nothing.

```
template <class Lock, class Predicate>
void wait(Lock& lock, Predicate pred);
```

Effects: Equivalent to:
```
while (!pred())
    wait(lock);
```

```
template <class Lock, class Clock, class Duration>
    cv_status wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time);
```

Effects:
— Atomically calls `lock.unlock()` and blocks on `*this`.
— When unblocked, calls `lock.lock()` (possibly blocking on the lock) and returns.
— The function will unblock when signaled by a call to `notify_one()`, a call to `notify_all()`, expiration of the absolute timeout (30.2.4) 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, `std::terminate()` shall be called (15.5.1).
[Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Postcondition: `lock` is locked by the calling thread.

Returns: `cv_status::timeout` if the absolute timeout (30.2.4) specified by `abs_time` expired, otherwise `cv_status::no_timeout`.

Throws: Timeout-related exceptions (30.2.4).

```
template <class Lock, class Rep, class Period>
    cv_status wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time);
```

Effects: Equivalent to:
```
    return wait_until(lock, chrono::steady_clock::now() + rel_time);
```

Returns: `cv_status::timeout` if the relative timeout (30.2.4) specified by `rel_time` expired, otherwise `cv_status::no_timeout`.

Remarks: If the function fails to meet the postcondition, `std::terminate()` shall be called (15.5.1).
[Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Postcondition: `lock` is locked by the calling thread.

Throws: Timeout-related exceptions (30.2.4).

```
template <class Lock, class Clock, class Duration, class Predicate>
    bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time, Predicate pred);
```
Effects: Equivalent to:

```cpp
while (!pred())
    if (wait_until(lock, abs_time) == cv_status::timeout)
        return pred();
    return true;
```

[Note: There is no blocking if `pred()` is initially `true`, or if the timeout has already expired. — end note]

[Note: The returned value indicates whether the predicate evaluates to `true` regardless of whether the timeout was triggered. — end note]

```cpp
template <class Lock, class Rep, class Period, class Predicate>
bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time, Predicate pred);
```

Effects: Equivalent to:

```cpp
return wait_until(lock, chrono::steady_clock::now() + rel_time, std::move(pred));
```

### 30.6 Futures [futures]

#### 30.6.1 Overview [futures.overview]

30.6 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]

Header `<future>` synopsis

```cpp
namespace std {
    enum class future_errc {
        broken_promise = implementation-defined,
        future_already_retrieved = implementation-defined,
        promise_already_satisfied = implementation-defined,
        no_state = implementation-defined
    };

    enum class launch : unspecified {
        async = unspecified,
        deferred = unspecified,
        implementation-defined
    };

    enum class future_status {
        ready,
        timeout,
        deferred
    };

    template <> struct is_error_code_enum<future_errc> : public true_type { };  
    error_code make_error_code(future_errc e) noexcept;
    error_condition make_error_condition(future_errc e) noexcept;

    const error_category& future_category() noexcept;

class future_error;
```
template <class R> class promise;
template <class R> class promise<R&>;
template <> class promise<void>;

template <class R>
void swap(promise<R>& x, promise<R>& y) noexcept;

template <class R, class Alloc>
struct uses_allocator<promise<R>, Alloc>;

template <class R> class future;
template <class R> class future<R&>;
template <> class future<void>;

template <class R> class shared_future;
template <class R> class shared_future<R&>;
template <> class shared_future<void>;

template <class> class packaged_task; // undefined

template <class R, class... ArgTypes>
class packaged_task<R(ArgTypes...)>
class packaged_task<R(ArgTypes...)>;

template <class R, class... ArgTypes>
void swap(packaged_task<R(ArgTypes...)>&, packaged_task<R(ArgTypes...)>&) noexcept;

template <class R, class Alloc>
struct uses_allocator<packaged_task<R>, Alloc>;

template <class F, class... Args>
future<result_of_t<decay_t<F>(decay_t<Args>...)>>
async(F&& f, Args&&... args);

template <class F, class... Args>
future<result_of_t<decay_t<F>(decay_t<Args>...)>>
async(launch policy, F&& f, Args&&... args);
}

The enum type `launch` is a bitmask type (17.5.2.1.3) with `launch::async` and `launch::deferred` denoting individual bits. [Note: Implementations can provide bitmasks to specify restrictions on task interaction by functions launched by `async()` applicable to a corresponding subset of available launch policies. Implementations can extend the behavior of the first overload of `async()` by adding their extensions to the launch policy under the “as if” rule. — end note]

The enum values of `future_errc` are distinct and not zero.

### 30.6.2 Error handling

```cpp
const error_category& future_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 "future".

```cpp
error_code make_error_code(future_errc e) noexcept;
```
Returns: error_code(static_cast<int>(e), future_category()).

error_condition make_error_condition(future_errc e) noexcept;

Returns: error_condition(static_cast<int>(e), future_category()).

### 30.6.3 Class future_error

```cpp
namespace std {
    class future_error : public logic_error {
        public:
            future_error(error_code ec); // exposition only

            const error_code& code() const noexcept;
            const char* what() const noexcept;
    };
}
```

const error_code& code() const noexcept;

Returns: The value of ec that was passed to the object’s constructor.

const char* what() const noexcept;

Returns: An ntbs incorporating code().message().

### 30.6.4 Shared state

Many of the classes introduced in this sub-clause 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]

[Note: The result can be any kind of object including a function to compute that result, as used by async when policy is launch::deferred. — end note]

An asynchronous return object is an object that reads results from a shared state. A waiting function of an asynchronous return object is one that potentially blocks to wait for the shared state to be made ready. If a waiting function can return before the state is made ready because of a timeout (30.2.5), then it is a timed waiting function, otherwise it is a non-timed waiting function.

An asynchronous provider is an object that provides a result to a shared state. The result of a shared state is set by respective functions on the asynchronous provider. [Note: Such as promises or tasks. — end note]

The means of setting the result of a shared state is specified in the description of those classes and functions that create such a state object.

When an asynchronous return object or an asynchronous provider is said to release its shared state, it means:

- if the return object or provider holds the last reference to its shared state, the shared state is destroyed;
  and

- the return object or provider gives up its reference to its shared state; and

- these actions will not block for the shared state to become ready, except that it may block if all of the following are true: the shared state was created by a call to std::async, the shared state is not yet ready, and this was the last reference to the shared state.

When an asynchronous provider is said to make its shared state ready, it means:

- first, the provider marks its shared state as ready; and

§ 30.6.4
(6.2) — second, the provider unblocks any execution agents waiting for its shared state to become ready.

7 When an asynchronous provider is said to abandon its shared state, it means:

(7.1) — first, if that state is not ready, the provider
(7.1.1) — stores an exception object of type future_error with an error condition of broken Promise
within its shared state; and then
(7.1.2) — makes its shared state ready;
(7.2) — second, the provider releases its shared state.

8 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.

9 Calls to functions that successfully set the stored result of a shared state synchronize with (1.10) 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 (1.10) the successful return from a call to a waiting function on the shared state.

10 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 (3.7.2) is sequenced before making that shared state ready.

11 Access to the result of the same shared state may conflict (1.10). [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 (17.6.5.9). For example, concurrent accesses through references returned by shared_future::get (30.6.7) must either use read-only operations or provide additional synchronization. — end note]

### 30.6.5 Class template promise

```cpp
namespace std {
    template <class R>
    class promise {
        public:
            promise();
            template <class Allocator>
                promise(allocation_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(const R& r);
            void set_value_at_thread_exit(see below);

            § 30.6.5
        }
    }
} // namespace std
```

[30.6.5]
void set_exception_at_thread_exit(exception_ptr p);
};
template <class R>
void swap(promise<R>& x, promise<R>& y) noexcept;
template <class R, class Alloc>
struct uses_allocator<promise<R>, Alloc> {
   true_type {
      Reqd: Alloc shall be an Allocator (17.6.3.5).
   }
};

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.
Postcondition: rhs has no shared state.

~promise();
Effects: Abandons any shared state (30.6.4).

promise& operator=(promise&& rhs) noexcept;
Effects: Abandons any shared state (30.6.4) 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.
Postcondition: *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:
future_already_retrieved if get_future has already been called on a promise with the same shared state as *this.

— 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 (30.6.4).

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.

void set_exception(exception_ptr p);

Effects: atomically stores the exception pointer p in the shared state and makes that state ready (30.6.4).

Throws: future_error if its shared state already has a stored value or exception.

Error conditions:

— promise_already_satisfied if its shared state already has a stored value or exception.

— no_state if *this has no shared state.

void promise::set_value_at_thread_exit(const R& r);
void promise::set_value_at_thread_exit(R&& r);
void promise<R&>::set_value_at_thread_exit(R& r);
void promise<void>::set_value_at_thread_exit();

Effects: Stores the value r in the shared state without making that state ready immediately. Schedules that state to be made ready when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed.

Throws:

— 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.

void set_exception_at_thread_exit(exception_ptr p);

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:

§ 30.6.5
(26.1) promise_already_satisfied if its shared state already has a stored value or exception.
(26.2) no_state if *this has no shared state.

```
template <class R>
  void swap(promise<R>& x, promise<R>& y);
```

**30.6.6 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 (30.6.4) 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]

2 The effect of calling any member function other than the destructor, the move-assignment operator, or `valid` on a `future` object for which `valid() == false` is undefined. [Note: Implementations are encouraged to detect this case and throw an object of type `future_error` with an error condition of `future_errc::no_state`. — end note]

```
namespace std {
    template <class R>
    class future {
        public:
            future() noexcept;
            future(future &&) noexcept;
            future(const future& rhs) = delete;
            ~future();
            future& operator=(const future& rhs) = delete;
            future& operator=(future&&) noexcept;
            shared_future<R> share();

            // retrieving the value
            see below get();

            // functions to check state
            bool valid() const noexcept;

            void wait() const;
            template <class Rep, class Period>
                future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
            template <class Clock, class Duration>
                future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
    }
}
```

4 The implementation shall provide the template `future` and two specializations, `future<R&>` and `future<void>`. These differ only in the return type and return value of the member function `get`, as set out in its description, below.

```
future() noexcept;
```
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 (30.6.4);
(9.2) destroys *this.

future& operator=(future&& rhs) noexcept;

Effects:
(10.1) releases any shared state (30.6.4).
(10.2) move assigns the contents of rhs to *this.

Postconditions:
(11.1) valid() returns the same value as rhs.valid() prior to the assignment.
(11.2) rhs.valid() == false.

shared_future<R> share();

Returns: shared_future<R>(std::move(*this)).

Postcondition: valid() == false.

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.

Effects: wait()s until the shared state is ready, then retrieves the value stored in the shared state.

Returns:
(16.1) future::get() returns the value v stored in the object’s shared state as std::move(v).
(16.2) future<R&>::get() returns the reference stored as value in the object’s shared state.
(16.3) future<void>::get() returns nothing.

Throws: the stored exception, if an exception was stored in the shared state.

Postcondition: valid() == false.
>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 (30.6.8), otherwise blocks until the shared state is ready or until the relative timeout (30.2.4) specified by rel_time has expired.

>Returns:
(22.1) — future_status::deferred if the shared state contains a deferred function.
(22.2) — future_status::ready if the shared state is ready.
(22.3) — future_status::timeout if the function is returning because the relative timeout (30.2.4) specified by rel_time has expired.

Throws: timeout-related exceptions (30.2.4).

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 (30.6.8), otherwise blocks until the shared state is ready or until the absolute timeout (30.2.4) specified by abs_time has expired.

(Returns:
(25.1) — future_status::deferred if the shared state contains a deferred function.
(25.2) — future_status::ready if the shared state is ready.
(25.3) — future_status::timeout if the function is returning because the absolute timeout (30.2.4) specified by abs_time has expired.

Throws: timeout-related exceptions (30.2.4).

30.6.7 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 (30.6.4) 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 shared state. — end note]

The effect of calling any member function other than the destructor, the move-assignment operator, or valid() on a shared_future object for which valid() == false is undefined. [Note: Implementations are encouraged to 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);
      shared_future(future<R>&& rhs) noexcept;
  }
}

§ 30.6.7
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;`

*Effects:* constructs an *empty* `shared_future` object that does not refer to a shared state.

*Postcondition:* `valid() == false`.

`shared_future(const shared_future& rhs);`

*Effects:* constructs a `shared_future` object that refers to the same shared state as `rhs` (if any).

*Postcondition:* `valid()` returns the same value as `rhs.valid()`.

`shared_future(future<R>&& rhs) noexcept;`

*Effects:* move constructs a `shared_future` object that refers to the shared state that was originally referred to by `rhs` (if any).

*Postconditions:*

1. `valid()` returns the same value as `rhs.valid()` returned prior to the constructor invocation.
2. `rhs.valid() == false`.

`~shared_future();`

*Effects:*

1. releases any shared state (30.6.4);
2. destroys `*this`.

`shared_future& operator=(shared_future&& rhs) noexcept;`

*Effects:*

1. releases any shared state (30.6.4);
move assigns the contents of rhs to *this.

Postconditions:

— valid() returns the same value as rhs.valid() returned prior to the assignment.
— rhs.valid() == false.

shared_future& operator=(const shared_future& rhs);

Effects:

— releases any shared state (30.6.4);
— 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().

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.

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 (1.10).

Effects: wait()s 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 (30.6.8), otherwise blocks until the shared state is ready or until the relative timeout (30.2.4) 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 \((30.2.4)\) specified by \texttt{rel\_time} has expired.

Throws: timeout-related exceptions \((30.2.4)\).

\[
\text{template} <\text{class Clock, class Duration}>
\text{future\_status} \text{wait\_until} (\text{const} \text{chrono}::\text{time\_point<Clock, Duration>>& \text{abs\_time}) \text{const};
\]

Effects: none if the shared state contains a deferred function \((30.6.8)\), otherwise blocks until the shared state is ready or until the absolute timeout \((30.2.4)\) specified by \texttt{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 \((30.2.4)\) specified by \texttt{abs\_time} has expired.

Throws: timeout-related exceptions \((30.2.4)\).

### 30.6.8 Function template \texttt{async}

The function template \texttt{async} provides a mechanism to launch a function potentially in a new thread and provides the result of the function in a \texttt{future} object with which it shares a shared state.

\[
\text{template} <\text{class F, class... Args}>
\text{future<result\_of_t<\text{decay\_t<F}(\text{decay\_t<Args>...})>>} \text{async}(\text{F}&& \text{f}, \text{Args}&&... \text{args});
\]

\[
\text{template} <\text{class F, class... Args}>
\text{future<result\_of_t<\text{decay\_t<F}(\text{decay\_t<Args>...})>>} \text{async(\text{launch\_policy, F}&& \text{f}, \text{Args}&&... \text{args});}
\]

Requires: \texttt{F} and each \texttt{Ti} in \texttt{Args} shall satisfy the \texttt{MoveConstructible} requirements. \texttt{INVOLVE(DECAY\_COPY (std::forward<F>(\text{f})), DECAY\_COPY (std::forward<Args>(\text{args})...))} \((20.9.2, 30.3.1.2)\) shall be a valid expression.

Effects: The first function behaves the same as a call to the second function with a \texttt{policy} argument of \texttt{launch::async \mid launch::deferred} and the same arguments for \texttt{F} and \texttt{Args}. The second function creates a shared state that is associated with the returned \texttt{future} object. The further behavior of the second function depends on the \texttt{policy} argument as follows (if more than one of these conditions applies, the implementation may choose any of the corresponding policies):

\[\text{(3.1)}\]

- if \texttt{policy \& launch::async} is non-zero — calls \texttt{INVOLVE(DECAY\_COPY (std::forward<F>(\text{f})), DECAY\_COPY (std::forward<Args>(\text{args})...))} \((20.9.2, 30.3.1.2)\) as if in a new thread of execution represented by a \texttt{thread} object with the calls to \texttt{DECAY\_COPY()} being evaluated in the thread that called \texttt{async}. Any return value is stored as the result in the shared state. Any exception propagated from the execution of \texttt{INVOLVE (DECAY\_COPY (std::forward<F>(\text{f})), DECAY\_COPY (std::forward<Args>(\text{args})...))} is stored as the exceptional result in the shared state. The \texttt{thread} object is stored in the shared state and affects the behavior of any asynchronous return objects that reference that state.

\[\text{(3.2)}\]

- if \texttt{policy \& launch::deferred} is non-zero — Stores \texttt{DECAY\_COPY (std::forward<F>(\text{f})) and DECAY\_COPY (std::forward<Args>(\text{args})...)} in the shared state. These copies of \texttt{f} and \texttt{args} constitute a deferred function. Invocation of the deferred function evaluates \texttt{INVOLVE (std::move(g), std::move(xyz))} where \texttt{g} is the stored value of \texttt{DECAY\_COPY (std::forward<F>(\text{f})) and xyz} is the stored copy of \texttt{DECAY\_COPY (std::forward<Args>(\text{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 \((30.6.4)\) 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 \texttt{INVOLVE (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]

— If no value is set in the launch policy, or a value is set that is neither specified in this International
Standard or by the implementation, the behaviour is undefined.

Returns: An object of type future<result_of_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 std::async is moved
outside the local scope, other code that uses the future must be aware that the future’s destructor may
block for the shared state to become ready. — end note]

Synchronization: Regardless of the provided policy argument,

— the invocation of async synchronizes with (1.10) the invocation of f. [Note: This statement
applies even when the corresponding future object is moved to another thread. — end note]; and

— the completion of the function f is sequenced before (1.10) the shared state is made ready. [Note: f
might not be called at all, so its completion might never happen. — end note]

If the implementation chooses the launch::async policy,

— a call to a waiting function on an asynchronous return object that shares the shared state created
by this async call shall block until the associated thread has completed, as if joined, or else time
out (30.3.1.5);

— the associated thread completion synchronizes with (1.10) the return from the first function that
successfully detects the ready status of the shared state or with the return from the last function
that releases the shared state, whichever happens first.

Throws: system_error if policy == launch::async and the implementation is unable to start a
new thread.

Error conditions:

— resource_unavailable_try_again — if policy == launch::async and the system is unable
to start a new thread.

[Example:

```cpp
int work1(int value);
int work2(int value);
int work(int value) {
    auto handle = std::async([=]{ return work2(value); });
    int tmp = work1(value);
    return tmp + handle.get(); // #1
}
```

[Note: Line #1 might not result in concurrency because the async call uses the default policy, which may
use launch::deferred, in which case the lambda might not be invoked until the get() call; in that case,
work1 and work2 are called on the same thread and there is no concurrency. — end note] — end example]

30.6.9 Class template packaged_task [futures.task]

The class template packaged_task defines a type for wrapping a function or callable object so that the
return value of the function or callable object is stored in a future when it is invoked.

When the packaged_task object is invoked, its stored task is invoked and the result (whether normal or
exceptional) stored in the shared state. Any futures that share the shared state will then be able to access
the stored result.
namespace std {

template<class> class packaged_task; // undefined

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);
    template <class F, class Allocator>
        explicit packaged_task(allocator_arg_t, const Allocator& a, F&& f);

    ~packaged_task();

    // no copy
    packaged_task(const packaged_task&) = delete;
    packaged_task& operator=(const packaged_task&) = delete;

    // move support
    packaged_task(packaged_task&& rhs) noexcept;
    packaged_task& operator=(packaged_task&& rhs) noexcept;
    void swap(packaged_task& other) noexcept;

    bool valid() const noexcept;

    // result retrieval
    future<R> get_future();

    // execution
    void operator()(ArgTypes...);
    void make_ready_at_thread_exit(ArgTypes...);

    void reset();
};

template <class R, class ArgTypes>
void swap(packaged_task<R(ArgTypes...)>& x, packaged_task<R(ArgTypes...)>& y) noexcept;

template <class R, class Alloc>
struct uses_allocator<packaged_task<R>, Alloc>;
}  

30.6.9.1 packaged_task member functions [futures.task.members]

packaged_task() noexcept;  

Effects: constructs a packaged_task object with no shared state and no stored task.

template <class F>
    packaged_task(F&& f);
    template <class F, class Allocator>
        explicit packaged_task(allocation_arg_t, const Allocator& a, F&& f);

Requires: INVOKE(f, t1, t2, ..., tN, R), where t1, t2, ..., tN are values of the corresponding types in ArgTypes..., shall be a valid expression. Invoking a copy of f shall behave the same as invoking f.

Remarks: These constructors shall not participate in overload resolution if decay_t<F> is the same type as std::packaged_task<R(ArgTypes...)>. 

§ 30.6.9.1 1207
Effects: constructs a new `packaged_task` object with a shared state and initializes the object’s stored task with `std::forward<F>(f)`. The constructors that take an `Allocator` argument use it to allocate memory needed to store the internal data structures.

Throws: any exceptions thrown by the copy or move constructor of `f`, or `std::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`.

Postcondition: `rhs` has no shared state.

`packaged_task& operator=(packaged_task&& rhs) noexcept;`

Effects:

(8.1) releases any shared state (30.6.4).
(8.2) `packaged_task(std::move(rhs)).swap(*this)`.

~`packaged_task();`

Effects: Abandons any shared state. (30.6.4).

void swap(packaged_task& other) noexcept;

Effects: exchanges the shared states and stored tasks of `*this` and `other`.

Postcondition: `*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:

(15.1) `future_already_retrieved` if `get_future` has already been called on a `packaged_task` object with the same shared state as `*this`.
(15.2) `no_state` if `*this` has no shared state.

void operator()(ArgTypes... args);

Effects: `INVOKE(f, t1, t2, ..., tN, R)`, where `f` is the stored task of `*this` and `t1`, `t2`, ..., `tN` are the values in `args`.... If the task returns normally, the return value is stored as the asynchronous result in the shared state of `*this`, otherwise the exception thrown by the task is stored. The shared state of `*this` is made ready, and any threads blocked in a function waiting for the shared state of `*this` to become ready are unblocked.

Throws: a `future_error` exception object if there is no shared state or the stored task has already been invoked.

Error conditions:

(18.1) `promise_already_satisfied` if the stored task has already been invoked.
(18.2) — no_state if *this has no shared state.

```cpp
void make_ready_at_thread_exit(ArgTypes... args);
```

19 Effects: `INVOKE(f, t1, t2, ..., tN, R)`, where f is the stored task and t1, t2, ..., tN are the values in args.... If the task returns normally, the return value is stored as the asynchronous result in the shared state of *this, otherwise the exception thrown by the task is stored. In either case, this shall be done without making that state ready (30.6.4) 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.

20 Throws: `future_error` if an error condition occurs.

21 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.

```cpp
void reset();
```

22 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 (30.6.4). — end note]

23 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.

30.6.9.2 packaged_task globals [futures.task.nonmems]

```cpp
template <class R, class... ArgTypes>
void swap(packaged_task<R(ArgTypes...)>& x, packaged_task<R(ArgTypes...)>& y) noexcept;
```

1 Effects: x.swap(y)

```cpp
template <class R, class Alloc>
struct uses_allocator<packaged_task<R>, Alloc>
: true_type { }; 
```

2 Requires: Alloc shall be an Allocator (17.6.3.5).
Annex A  (informative)  
Grammar summary

1 This summary of C++ syntax 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 (6.8, 7.1, 10.2) 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

1 New context-dependent keywords are introduced into a program by typedef (7.1.3), namespace (7.3.1), class (Clause 9), enumeration (7.2), and template (Clause 14) declarations.

typedef-name:
    identifier
namespace-name:
    original-namespace-name
    namespace-alias
original-namespace-name:
    identifier
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 (9.1).

A.2 Lexical conventions

hex-quad:
   hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit
universal-character-name:
\u hex-quad
\U hex-quad hex-quad
preprocessing-token:
    header-name
    identifier
    pp-number
    character-literal
    user-defined-character-literal
    string-literal
    user-defined-string-literal
    preprocessing-op-or-punc
    each non-white-space character that cannot be one of the above
token:
  identifier
  keyword
  literal
  operator
  punctuator

header-name:
  < h-char-sequence >
  " q-char-sequence "

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

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

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

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

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

identifier:
  identifier-nondigit
  identifier identifier-nondigit
  identifier digit

identifier-nondigit:
  nondigit
  universal-character-name
  other implementation-defined characters

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 ? :: . .*
    + - * / % ˆ & |
    ! = < > += -= *= /= %=
    *= &= |= <<= >> >>= <<= == !=
    <= >= && || ++ -- , ->* ->
and and_eq bitand bitor compl not not_eq
or or_eq xor xor_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

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:
  0x hexadecimal-digit
  0X hexadecimal-digit
  hexadecimal-literal ’opt hexadecimal-digit

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-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
      \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:
      fractional-constant exponent-part\opt floating-suffix\opt
      digit-sequence exponent-part floating-suffix\opt
   fractional-constant:
      digit-sequence\opt . digit-sequence
      digit-sequence
   exponent-part:
      e sign\opt digit-sequence
      E sign\opt digit-sequence
   sign: one of
      + -
   digit-sequence:
      digit
      digit-sequence \opt digit
   floating-suffix: one of
      f l F L
   string-literal:
      encoding-prefix\opt s-char-sequence\opt"
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

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

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

ud-suffix:
  identifier
A.3 Basic concepts

translation-unit:
  declaration-seq_opt

A.4 Expressions

primary-expression:
  literal
  this
    ( expression )
  id-expression
  lambda-expression
  fold-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 lambda-declarator_opt compound-statement

lambda-introducer:
  [ lambda-capture_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

§ A.4
init-capture:
  identifier initializer
  & identifier initializer

lambda-declarator:
  ( parameter-declaration-clause ) mutable_opt
  exception-specification_opt attribute-specifier-seq_opt trailing-return-type_opt

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

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

postfix-expression:
  primary-expression
  postfix-expression [ expression ]
  postfix-expression [ 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:
  ::optnew new-placement opt new-type-id new-initializer opt
  ::optnew new-placement opt ( type-id ) new-initializer opt
new-placement:
  ( expression-list )
new-type-id:
  type-specifier-seq new-declarator opt
new-declarator:
  ptr-operator new-declarator opt
  noptr-new-declarator
noptr-new-declarator:
  [ expression ] attribute-specifier-seq opt
  noptr-new-declarator [ constant-expression ] attribute-specifier-seq opt
new-initializer:
  ( expression-list opt )
  braced-init-list
delete-expression:
  ::optdelete cast-expression
  ::optdelete [ ] 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
relational-expression:
  shift-expression
  relational-expression < shift-expression
  relational-expression > shift-expression
  relational-expression <= shift-expression
  relational-expression >= shift-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
  logical-or-expression ? expression : assignment-expression
throw-expression:
  throw assignment-expression opt
assignment-expression:
  conditional-expression
  logical-or-expression assignment-operator initializer-clause
  throw-expression
assignment-operator: one of
  = *= /= %= += -= >>= <<= &= ^= |=
expression:
  assignment-expression
  expression , assignment-expression
constant-expression:
  conditional-expression

A.5 Statements

statement:
  labeled-statement
  attribute-specifier-seq opt expression-statement
  attribute-specifier-seq opt compound-statement
  attribute-specifier-seq opt attribute-specifier-seq opt selection-statement
  attribute-specifier-seq opt attribute-specifier-seq opt iteration-statement
  attribute-specifier-seq opt attribute-specifier-seq opt jump-statement
  declaration-statement
  attribute-specifier-seq opt try-block
labeled-statement:
  attribute-specifier-seq opt identifier : statement
  attribute-specifier-seq opt case constant-expression : statement
  attribute-specifier-seq opt default : statement
expression-statement:
  expression opt ;
compound-statement:
  { statement-seq opt }
selection-statement:
  if ( condition ) statement
  if ( condition ) statement else statement
  switch ( condition ) statement

condition:
  expression
attribute-specifier-seq specified-decl declspec-seq declarator = initializer-clause
attribute-specifier-seq specified-decl declarator braced-init-list

iteration-statement:
  while ( condition ) statement
  do statement while ( expression ) ;
  for ( for-init-statement condition_opt; expression_opt) statement
  for ( for-range-declaration : for-range-initializer) statement

for-init-statement:
  expression-statement
  simple-declaration
for-range-declaration:
  attribute-specifier-seq specified-decl declspec-seq declarator

for-range-initializer:
  expression
  braced-init-list

jump-statement:
  break ;
  continue ;
  return expression_opt ;
  return braced-init-list ;
  goto identifier ;

declaration-statement:
  block-declaration

A.6 Declarations [gram.dcl]

declaration-seq:
  declaration
  declaration-seq declaration

declaration:
  block-declaration
  function-definition
  template-declaration
  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
alias-declaration:
  using identifier attribute-specifier-seqopt = type-id ;
simple-declaration:
  decl-specifier-seqopt init-declarator-listopt ;
  attribute-specifier-seq decl-specifier-seqopt init-declarator-list ;
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
  type-specifier
  function-specifier
  friend
typedef
constexpr
decl-specifier-seq:
  decl-specifier attribute-specifier-seqopt
decl-specifier decl-specifier-seq
storage-class-specifier:
  register
  static
  thread_local
  extern
  mutable
function-specifier:
  inline
  virtual
  explicit
typedef-name:
  identifier
type-specifier:
  trailing-type-specifier
  class-specifier
  enum-specifier
trailing-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
trailing-type-specifier-seq:
  trailing-type-specifier attribute-specifier-seqopt
  trailing-type-specifier trailing-type-specifier-seq
simple-type-specifier:
  nested-name-specifier_opt type-name
  nested-name-specifier template simple-template-id
char
cchar16_t
cchar32_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 identifier_opt enum-base_opt
  enum-key attribute-specifier-seq_opt nested-name-specifier identifier
  enum-base_opt
opaque-enum-declaration:
  enum-key attribute-specifier-seq_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

§ A.6 1221
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:
  nested-name-specifier

using-declaration:
  using typename\_opt nested-name-specifier unqualified-id ;

using-directive:
  attribute-specifier-seq\_opt using namespace nested-name-specifier\_opt namespace-name ;

asm-definition:
  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-list ] ]
  alignment-specifier

alignment-specifier:
  alignas ( type-id ...\_opt )
  alignas ( constant-expression ...\_opt )

attribute-list:
  attribute\_opt
  attribute-list , attribute\_opt
  attribute ...
  attribute-list , attribute ...

attribute:
  attribute-token attribute-argument-clause\_opt

attribute-token:
  identifier
  attribute-scoped-token
attribute-scoped-token:
  attribute-namespace : : identifier
attribute-namespace:
  identifier
attribute-argument-clause:
  ( balanced-token-seq )
balanced-token-seq:
  balanced-token_opt
  balanced-token-seq balanced-token
balanced-token:
  ( balanced-token-seq )
  [ balanced-token-seq ]
  { balanced-token-seq }
  any token other than a parenthesis, a bracket, or a brace

A.7 Declarators

init-declarator-list:
  init-declarator
  init-declarator-list , init-declarator
init-declarator:
  declarator initializer_opt
declarator:
  ptr-declarator
  ptr-declarator parameters-and-qualifiers trailing-return-type
ptr-declarator:
  nptr-declarator
  ptr-operator ptr-declarator
nptr-declarator:
  declarator-id attribute-specifier-seq_opt
  nptr-declarator parameters-and-qualifiers
  nptr-declarator [ constant-expression_opt ] attribute-specifier-seq_opt
  ( ptr-declarator )
parameters-and-qualifiers:
  ( parameter-declaration-clause ) cv-qualifier-seq_opt
  ref-qualifier_opt exception-specification_opt attribute-specifier-seq_opt
trailing-return-type:
  -> trailing-type-specifier-seq abstract-declarator_opt
ptr-operator:
  * attribute-specifier-seq_opt cv-qualifier-seq_opt
  & attribute-specifier-seq_opt
  && attribute-specifier-seq_opt
  nested-name-specifier * attribute-specifier-seq_opt cv-qualifier-seq_opt
cv-qualifier-seq:
  cv-qualifier cv-qualifier-seq_opt
cv-qualifier:
  const
  volatile
ref-qualifier:
  &
  &&
declarator-id:
  ... opt id-expression
type-id:
  type-specifier-seq abstract-declarator_opt
abstract-declarator:
  ptr-abstract-declarator
  nopt ptr-abstract-declarator_opt parameters-and-qualifiers trailing-return-type
  abstract-pack-declarator
ptr-abstract-declarator:
  nopt ptr-abstract-declarator
  ptr-operator ptr-abstract-declarator_opt
nopt ptr-abstract-declarator:
  nopt ptr-abstract-declarator_opt parameters-and-qualifiers
  nopt ptr-abstract-declarator_opt [ constant-expression_opt ] attribute-specifier-seq_opt
  ( ptr-abstract-declarator )
abstract-pack-declarator:
  nopt ptr-abstract-pack-declarator
  ptr-operator abstract-pack-declarator
nopt abstract-pack-declarator:
  nopt abstract-pack-declarator parameters-and-qualifiers
  nopt 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 declarator virt-specifier-seq_opt 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-list
initializer-clause:
  assignment-expression
  braced-init-list
initializer-list:
  initializer-clause ...
  initializer-list , initializer-clause ...
brace-or-equal-initializer:
  { initializer-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-nameSpecifier_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 declSpecifier-seq_opt member-declarator-list_opt ;
  function-definition
  using-declaration
  static_assert-declaration
  template-declaration
  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 brace-or-equal-initializer_opt
  identifier_opt attribute-specifier-seq_opt : constant-expression
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

§ A.9
base-specifier:
    attribute-specifier-seq_opt base-type-specifier
    attribute-specifier-seq_opt virtual access-specifier_opt base-type-specifier
    attribute-specifier-seq_opt access-specifier virtual_opt base-type-specifier

class-or-decltype:
    nested-name-specifier_opt class-name
decltype-specifier
base-type-specifier:
    class-or-decltype
access-specifier:
    private
    protected
    public

A.10 Special member functions

conversion-function-id:
    operator conversion-type-id
conversion-type-id:
    type-specifier-seq conversion-declarator_opt
conversion-declarator:
    ptr-operator conversion-declarator_opt
cctor-initializer:
    : mem-initializer-list
mem-initializer-list:
    mem-initializer ..._opt
    mem-initializer-list , mem-initializer ..._opt
mem-initializer:
    mem-initializer-id ( expression-list_opt )
    mem-initializer-id braced-init-list
mem-initializer-id:
    class-or-decltype
    identifier

A.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 < template-parameter-list > declaration
template-parameter-list:
    template-parameter
    template-parameter-list , template-parameter
template-parameter:
  type-parameter
  parameter-declaration

type-parameter:
  type-parameter-key ... opt identifier opt
  type-parameter-key identifier opt
  template < template-parameter-list > type-parameter-key ... opt identifier opt
  template < template-parameter-list > type-parameter-key identifier opt
  id-expression

type-parameter-key:
  class
typename

simple-template-id:
  typename
  template-name < template-argument-list opt>

template-id:
  simple-template-id
  operator-function-id < template-argument-list opt>
  literal-operator-id < template-argument-list opt>

template-name:
  identifier

template-argument-list:
  template-argument ...
  template-argument-list , template-argument ...

template-argument:
  constant-expression
  type-id
  id-expression

typename-specifier:
  typename nested-name-specifier identifier
typename nested-name-specifier template opt simple-template-id

explicit-instantiation:
  extern opt template declaration

explicit-specialization:
  template < > declaration

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

exception-specification:
  dynamic-exception-specification
  noexcept-specification

dynamic-exception-specification:
  throw ( type-id-list opt)
A.14 Preprocessing directives

[type-id-list:
  type-id ...opt
  type-id-list , type-id ...opt

noexcept-specification:
  noexcept ( constant-expression )
  noexcept]

§ A.14 1228
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
**Annex B  (informative)**

**Implementation quantities**

Because computers are finite, C++ implementations are inevitably limited in the size of the programs they can successfully process. Every implementation shall document those limitations where known. This documentation may cite fixed limits where they exist, say how to compute variable limits as a function of available resources, or say that fixed limits do not exist or are unknown.

The limits may constrain quantities that include those described below or others. The bracketed number following each quantity is recommended as the minimum for that quantity. However, these quantities are only guidelines and do not determine compliance.

1. Nesting levels of compound statements, iteration control structures, and selection control structures [256].
2. Nesting levels of conditional inclusion [256].
3. Pointer, array, and function declarators (in any combination) modifying a class, arithmetic, or incomplete type in a declaration [256].
4. Nesting levels of parenthesized expressions within a full-expression [256].
5. Number of characters in an internal identifier or macro name [1024].
6. Number of characters in an external identifier [1024].
7. External identifiers in one translation unit [65536].
8. Identifiers with block scope declared in one block [1024].
9. Macro identifiers simultaneously defined in one translation unit [65536].
10. Parameters in one function definition [256].
11. Arguments in one function call [256].
12. Parameters in one macro definition [256].
13. Arguments in one macro invocation [256].
14. Characters in one logical source line [65536].
15. Characters in a string literal (after concatenation) [65536].
16. Size of an object [262144].
17. Nesting levels for `#include` files [256].
18. Case labels for a `switch` statement (excluding those for any nested `switch` statements) [16384].
19. Data members in a single class [16384].
20. Enumeration constants in a single enumeration [4096].
(2.21) Levels of nested class definitions in a single member-specification [256].

(2.22) Functions registered by atexit() [32].

(2.23) Functions registered by at_quick_exit() [32].

(2.24) Direct and indirect base classes [16384].

(2.25) Direct base classes for a single class [1024].

(2.26) Members declared in a single class [4096].

(2.27) Final overriding virtual functions in a class, accessible or not [16384].

(2.28) Direct and indirect virtual bases of a class [1024].

(2.29) Static members of a class [1024].

(2.30) Friend declarations in a class [4096].

(2.31) Access control declarations in a class [4096].

(2.32) Member initializers in a constructor definition [6144].

(2.33) Scope qualifications of one identifier [256].

(2.34) Nested external specifications [1024].

(2.35) Recursive constexpr function invocations [512].

(2.36) Full-expressions evaluated within a core constant expression [1048576].

(2.37) Template arguments in a template declaration [1024].

(2.38) Recursively nested template instantiations, including substitution during template argument deduction (14.8.2) [1024].

(2.39) Handlers per try block [256].

(2.40) Throw specifications on a single function declaration [256].

(2.41) Number of placeholders (20.9.10.4) [10].


Annex C  (informative)
Compatibility

C.1  C++ and ISO C

This subclause lists the differences between C++ and ISO C, by the chapters of this document.

C.1.1  Clause 2: lexical conventions

2.11
Change: New Keywords
New keywords are added to C++; see 2.11.
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.

2.13.3
Change: Type of character literal is changed from int to char
Rationale: This is needed for improved overloaded function argument type matching. For example:

```c
int function( int i );
int function( char c );
function( 'x' );
```

It is preferable that this call match the second version of `function` rather than the first.
Effect on original feature: Change to semantics of well-defined feature. ISO C programs which depend on `sizeof('x') == sizeof(int)` will not work the same as C++ programs.
Difficulty of converting: Simple.
How widely used: Programs which depend upon `sizeof('x')` are probably rare.

Subclause 2.13.5:
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:

```
char* p = "abc";  // valid in C, invalid in C++
void f(char*) {
    char* p = (char*)"abc";  // OK: cast added
```

§ C.1.1
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 3: basic concepts

3.1 Change: C++ does not have “tentative definitions” as in C E.g., at file scope,

```c
int i;
int i;
```

is valid in C, invalid in C++. This makes it impossible to define mutually referential file-local static objects, if initializers are restricted to the syntactic forms of C. For example,

```c
struct X { int i; struct X* next; };
static struct X a;
static struct X b = { 0, &a };
static struct X a = { 1, &b };
```

Rationale: This avoids having different initialization rules for fundamental types and user-defined types.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation.

Rationale: 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.

3.3 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.

3.5 [also 7.1.6] 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 can be used as compile-time values in C++, this feature urges programmers to provide explicit initializer values for each const. This feature allows the user to put const objects in header files that are included in many compilation units.

Effect on original feature: Change to semantics of well-defined feature.

Difficulty of converting: Semantic transformation.

How widely used: Seldom

3.6 Change: Main 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

3.9

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 International Standard.

How widely used: Common.

C.1.3 Clause 4: standard conversions [diff.conv]

4.10

Change: Converting `void*` to a pointer-to-object type requires casting

```c
char a[10];
void* b=a;
void foo() {
    char* c=b;
}
```

ISO C will accept this usage of pointer to void being assigned to a pointer to object type. C++ will not.

Rationale: C++ tries harder than C to enforce compile-time type safety.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Could be automated. Violations will be diagnosed by the C++ translator. The fix is to add a cast. For example:

```c
char* c = (char*) b;
```

How widely used: This is fairly widely used but it is good programming practice to add the cast when assigning pointer-to-void to pointer-to-object. Some ISO C translators will give a warning if the cast is not used.

C.1.4 Clause 5: expressions [diff.expr]

5.2.2

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.

5.3.3, 5.4

Change: Types must be declared in declarations, not in expressions. In C, a `sizeof` expression or cast expression may create a new type. For example,

```c
p = (void*)(struct x {int i;} *)0;
```

declares a new type, `struct x`.

Rationale: This prohibition helps to clarify the location of declarations in the source code.
Effect on original feature: Deletion of a semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Seldom.
5.16, 5.18, 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 6: statements \[diff.stat\]
6.4.2, 6.6.4
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 run-time 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.

6.6.3
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 7: declarations \[diff.dcl\]
7.1.1
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.1.6
static struct S {  // valid C, invalid in C++
    int i;
};

Rationale: Storage class specifiers don’t have any meaning when associated with a type. In C++, class members can be declared with the static storage class specifier. Allowing storage class specifiers on type declarations could render the code confusing for users.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Syntactic transformation.

How widely used: Seldom.

7.1.3

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:

typedef struct name1 { /*...*/ } name1;  // valid C and C++
struct name { /*...*/ };                // valid C, invalid C++
typedef int name;                     // valid C, invalid C++

Rationale: For ease of use, C++ doesn’t require that a type name be prefixed with the keywords class, struct or union when used in object declarations or type casts.

Example:

class name { /*...*/ };                // i has type 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.

7.1.6 [see also 3.5]

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.

7.1.6

Change: Banning implicit int

In C++ a decl-specifier-seq must contain a type-specifier, unless it is followed by a declarator for a constructor, a destructor, or a conversion function. In the following example, the left-hand column presents valid C; the right-hand column presents equivalent C++:

```c
void f(const parm); void f(const int parm);
const n = 3; const int n = 3;
main() int main()
    /* ... */ /* ... */
```

§ C.1.6 1236
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.

7.1.6.4

Change: The keyword auto cannot be used as a storage class specifier.

```c
void f() {
    auto int x;  // valid C, invalid C++
}
```

Rationale: Allowing the use of auto to deduce the type of a variable from its initializer results in undesired interpretations of auto as a storage class specifier in certain contexts.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Syntactic transformation.

How widely used: Rare.

7.2

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:

```c
enum color { red, blue, green };
enum color c = 1;  // valid C, invalid C++
```

Rationale: The type-safe nature of C++.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Syntactic transformation. (The type error produced by the assignment can be automatically corrected by applying an explicit cast.)

How widely used: Common.

7.2

Change: In C++, the type of an enumerator is its enumeration. In C, the type of an enumerator is int.

Example:

```c
enum e { A };
sizeof(A) == sizeof(int)  // in C
sizeof(A) == sizeof(e)    // in C++
/* and sizeof(int) is not necessarily equal to sizeof(e) */
```

Rationale: In C++, an enumeration is a distinct type.

Effect on original feature: Change to semantics of well-defined feature.

Difficulty of converting: Semantic transformation.

How widely used: Seldom. The only time this affects existing C code is when the size of an enumerator is taken. Taking the size of an enumerator is not a common C coding practice.

§ C.1.6
C.1.7 Clause 8: declarators

8.3.5 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++
```

Rationale: This is to avoid erroneous function calls (i.e., function calls with the wrong number or type of arguments).

Effect on original feature: Change to semantics of well-defined feature. This feature was marked as “obsolescent” in C.

Difficulty of converting: Syntactic transformation. The function declarations using C incomplete declaration style must be completed to become full prototype declarations. A program may need to be updated further if different calls to the same (non-prototype) function have different numbers of arguments or if the type of corresponding arguments differed.

How widely used: Common.

8.3.5 [see 5.3.3] 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 compilation units, C++ relies on name equivalence when C relies on structural equivalence. Regarding parameter types: since the type defined in an 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 definitions is seen as poor coding style.

8.4 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.

8.5.2 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 routines, 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 9: classes

9.1 [see also 7.1.3]

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.

9.6

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.

9.7

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;
};
```
```c
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 of 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 3.3.

**How widely used:** Seldom.

9.9

**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 `T` 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 12: special member functions**

12.8

**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;
x3 = x1; // also invalid C++
```
**Rationale:** Several alternatives were debated at length. Changing the parameter to `volatile const X&` would greatly complicate the generation of efficient code for class objects. Discussion of providing two alternative signatures for these implicitly-defined operations raised unanswered concerns about creating ambiguities and complicating the rules that specify the formation of these operators according to the bases and members.

**Effect on original feature:** Deletion of semantically well-defined feature.

**Difficulty of converting:** Semantic transformation. If volatile semantics are required for the copy, a user-declared constructor or assignment must be provided. [Note: This user-declared constructor may be explicitly defaulted. — end note] If non-volatile semantics are required, an explicit `const_cast` can be used.

**How widely used:** Seldom.

### C.1.10 Clause 16: preprocessing directives

**Change:** Whether `_STDC_` is defined and if so, what its value is, are implementation-defined

**Rationale:** C++ is not identical to ISO C. Mandating that `_STDC_` be defined would require that translators make an incorrect claim. Each implementation must choose the behavior that will be most useful to its marketplace.

**Effect on original feature:** Change to semantics of well-defined feature.

**Difficulty of converting:** Semantic transformation.

**How widely used:** Programs and headers that reference `_STDC_` are quite common.

### C.2 C++ and ISO C++ 2003

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 2: lexical conventions

**Change:** New kinds of string literals

**Rationale:** Required for new features.

**Effect on original feature:** Valid C++ 2003 code may fail to compile or produce different results in this International Standard. Specifically, macros named `R`, `u8`, `u8R`, `u`, `uR`, `U`, `UR`, or `LR` will not be expanded when adjacent to a string literal but will be interpreted as part of the string literal. For example,

```cpp
#define u8 "abc"
const char* s = u8"def";    // Previously "abcdef", now "def"
```

**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.

```cpp
#define _x "there"
"hello"_x    // #1
```

Previously, #1 would have consisted of two separate preprocessing tokens and the macro `_x` would have been expanded. In this International Standard, #1 consists of a single preprocessing tokens, so the macro is not expanded.

**Change:** New keywords
Rationale: Required for new features.
Effect on original feature: Added to Table 3, 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.

2.13.2
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 4: standard conversions

Change: Only literals are integer null pointer constants
Rationale: Removing surprising interactions with templates and constant expressions
Effect on original feature: Valid C++ 2003 code may fail to compile or produce different results in this International Standard, 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 5: expressions

Change: Specify rounding for results of integer / and %
Rationale: Increase portability, C99 compatibility.
Effect on original feature: Valid C++ 2003 code that uses integer division rounds the result toward 0 or toward negative infinity, whereas this International Standard always rounds the result toward 0.

C.2.4 Clause 7: declarations

Change: Remove auto as a storage class specifier
Rationale: New feature.
Effect on original feature: Valid C++ 2003 code that uses the keyword auto as a storage class specifier may be invalid in this International Standard. In this International Standard, auto indicates that the type of a variable is to be deduced from its initializer expression.

C.2.5 Clause 8: declarators

Change: Narrowing restrictions in aggregate initializers
Rationale: Catches bugs.
Effect on original feature: Valid C++ 2003 code may fail to compile in this International Standard. For example, the following code is valid in C++ 2003 but invalid in this International Standard because double to int is a narrowing conversion:

```cpp
int x[] = { 2.0 };  
```

C.2.6 Clause 12: special member functions

Change: Implicitly-declared special member functions are defined as deleted when the implicit definition
would have been ill-formed.

**Rationale:** Improves template argument deduction failure.

**Effect on original feature:** A valid C++ 2003 program that uses one of these special member functions in a context where the definition is not required (e.g., in an expression that is not potentially evaluated) becomes ill-formed.

12.4 (destructors)

**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 `noexcept` or `noexcept(true)`. For a throwing virtual destructor of a derived class, `std::terminate()` can be avoided only if the base class virtual destructor has an exception specification that is not `noexcept` and not `noexcept(true)`.

### C.2.7 Clause 14: templates

14.1

**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.

14.3

**Change:** Remove whitespace requirement for nested closing template right angle brackets

**Rationale:** Considered a persistent but minor annoyance. Template aliases representing nonclass types would exacerbate whitespace issues.

**Effect on original feature:** Change to semantics of well-defined expression. A valid C++ 2003 expression containing a right angle bracket (">") followed immediately by another right angle bracket may now be treated as closing two templates. For example, the following code is valid in C++ 2003 because “>>” is a right-shift operator, but invalid in this International Standard because “>>” closes two templates.

```cpp
template <class T> struct X { };
template <int N> struct Y { };
X< Y< 1 >> 2 > > x;
```

14.6.4.2

**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 17: library introduction

17 – 30

**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.

17.6.1.2

**Change:** New headers

**Rationale:** New functionality.
Effect on original feature: The following C++ headers are new: `<array>`, `<atomic>`, `<chrono>`, `<codecvt>`, `<condition_variable>`, `<forward_list>`, `<future>`, `<initializer_list>`, `<mutex>`, `<random>`, `<ratio>`, `<regex>`, `<scoped_allocator>`, `<system_error>`, `<thread>`, `<tuple>`, `<typeindex>`, `<type_traits>`, `<unordered_map>`, and `<unordered_set>`. In addition the following C compatibility headers are new: `<ccomplex>`, `<cfenv>`, `<cinttypes>`, `<cstdalign>`, `<cstdbool>`, `<cstdint>`, `<ctgmath>`, and `<cuchar>`. Valid C++ 2003 code that includes headers with these names may be invalid in this International Standard.

17.6.3.2
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>`.

17.6.4.2.2
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.

17.6.5.3
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 18: language support library [diff.cpp03.language.support]

18.6.1.1
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
}```
18.6.1.1
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 19: diagnostics library
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 20: general utilities library
Effect on original feature: Valid C++ 2003 code, compiled without traceable pointer support, that interacts with newer C++ code using regions declared reachable may have different runtime behavior.

C.2.12 Clause 21: strings library
Rationale: Invalidation is subtly different with reference-counted strings. This change regularizes behavior for this International Standard.
Effect on original feature: Valid C++ 2003 code may execute differently in this International Standard.

C.2.13 Clause 23: containers library
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.
23.2
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:

- not all containers provide size(); use empty() instead of size() == 0;
- not all containers are empty after construction (array);
- not all containers have constant complexity for swap() (array).

23.2
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.

23.2.3, 23.2.4
Change: Signature changes: from void return types
Rationale: Old signature threw away useful information that may be expensive to recalculate.
Effect on original feature: The following member functions have changed:

- erase(iter) for set, multiset, map, multimap
- erase(begin, end) for set, multiset, map, multimap
- insert(pos, num, val) for vector, deque, list, forward_list
- insert(pos, beg, end) for vector, deque, list, forward_list

Valid C++ 2003 code that relies on these functions returning void (e.g., code that creates a pointer to member function that points to one of these functions) will fail to compile with this International Standard.

23.2.3, 23.2.4
Change: Signature changes: from iterator to const_iterator parameters
Rationale: Overspecification. Effects: The signatures of the following member functions changed from taking an iterator to taking a const_iterator:

- insert(iter, val) for vector, deque, list, set, multiset, map, multimap
- insert(pos, beg, end) for vector, deque, list, forward_list
- erase(iter) for set, multiset, map, multimap
- erase(begin, end) for set, multiset, map, multimap
- all forms of list::splice
- all forms of list::merge

Valid C++ 2003 code that uses these functions may fail to compile with this International Standard.

23.2.3, 23.2.4
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.2.13
C.2.14 Clause 25: algorithms library

25.1
Change: Result state of inputs after application of some algorithms
Rationale: Required by new feature.
Effect on original feature: A valid C++ 2003 program may detect that an object with a valid but unspecified state has a different valid but unspecified state with this International Standard. For example, std::remove and std::remove_if may leave the tail of the input sequence with a different set of values than previously.

C.2.15 Clause 26: numerics library

26.4
Change: Specified representation of complex numbers
Rationale: Compatibility with C99.
Effect on original feature: Valid C++ 2003 code that uses implementation-specific knowledge about the binary representation of the required template specializations of std::complex may not be compatible with this International Standard.

C.2.16 Clause 27: Input/output library

27.7.2.1.3, 27.7.3.4, 27.5.5.4
Change: Specify use of explicit in existing boolean conversion operators
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:

- passing a value to a function that takes an argument of type bool;
- using operator== to compare to false or true;
- returning a value from a function with a return type of bool;
- initializing members of type bool via aggregate initialization;
- initializing a const bool& which would bind to a temporary.

27.5.3.1.1
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.

27.5.3
Change: Flag types in std::ios_base are now bitmasks with values defined as constexpr static members
Rationale: Required for new features.
Effect on original feature: Valid C++ 2003 code that relies on std::ios_base flag types being represented as std::bitset or as an integer type may fail to compile with this International Standard. For example:

```cpp
#include <iostream>

int main() {
    int flag = std::ios_base::hex;
    std::cout.setf(flag); // error: setf does not take argument of type int
    return 0;
}
```

§ C.2.16
C.3  C++ and ISO C++ 2011  

This subclause lists the differences between C++ and ISO C++ 2011 (ISO/IEC 14882:2011, Programming Languages — C++), by the chapters of this document.

C.3.1 Clause 2: lexical conventions  

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:  

```
#define M(x, ...) __VA_ARGS__
int x[2] = { M(1'2,3'4) };  
// int x[2] = { 3'4 }; — this International Standard
```

C.3.2 Clause 3: basic concepts  

Change: New usual (non-placement) deallocator  
Rationale: Required for sized deallocation.  
Effect on original feature: Valid C++ 2011 code could declare a global placement allocation function and deallocation function as follows:  

```
void operator new(std::size_t, std::size_t);
void operator delete(void*, std::size_t) noexcept;
```

In this International Standard, however, the declaration of operator delete might match a predefined usual (non-placement) operator delete (3.7.4). If so, the program is ill-formed, as it was for class member allocation functions and deallocation functions (5.3.4).

C.3.3 Clause 7: declarations  

Change: constexpr non-static member functions are not implicitly const member functions.  
Rationale: Necessary to allow constexpr member functions to mutate the object.  
Effect on original feature: Valid C++ 2011 code may fail to compile in this International Standard. For example, the following code is valid in C++ 2011 but invalid in this International Standard because it declares the same member function twice with different return types:  

```
struct S {
    constexpr const int &f();
    int &f();
};
```

C.3.4 Clause 27: input/output library  

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 2: lexical conventions

2.2 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.

C.4.2 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.

C.5 C standard library

This subclause summarizes the contents of the C++ standard library included from the Standard C library. It also summarizes the explicit changes in definitions, declarations, or behavior from the Standard C library noted in other subclauses (17.6.1.2, 18.2, 21.8).

The C++ standard library provides 57 standard macros from the C library, as shown in Table 150.

The header names (enclosed in `< and >`) indicate that the macro may be defined in more than one header. All such definitions are equivalent (3.2).

<table>
<thead>
<tr>
<th>Table 150 — Standard macros</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert</td>
</tr>
<tr>
<td>BUFSIZ</td>
</tr>
<tr>
<td>CLOCKS_PER_SEC</td>
</tr>
<tr>
<td>EDOM</td>
</tr>
<tr>
<td>EILSEQ</td>
</tr>
<tr>
<td>EOF</td>
</tr>
<tr>
<td>ERANGE</td>
</tr>
<tr>
<td>errno</td>
</tr>
<tr>
<td>EXIT_FAILURE</td>
</tr>
<tr>
<td>EXIT_SUCCESS</td>
</tr>
<tr>
<td>FILENAME_MAX</td>
</tr>
<tr>
<td>POPEN_MAX</td>
</tr>
</tbody>
</table>

The C++ standard library provides 57 standard values from the C library, as shown in Table 151.

The C++ standard library provides 20 standard types from the C library, as shown in Table 152.

The C++ standard library provides 2 standard structs from the C library, as shown in Table 153.

The C++ standard library provides 209 standard functions from the C library, as shown in Table 154.
Table 151 — Standard values

<table>
<thead>
<tr>
<th>CHAR_BIT</th>
<th>FLT_DIG</th>
<th>INT_MIN</th>
<th>MB_LEN_MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR_MAX</td>
<td>FLT_EPSILON</td>
<td>LDBL_DIG</td>
<td>SCHAR_MAX</td>
</tr>
<tr>
<td>CHAR_MIN</td>
<td>FLT_MANT_DIG</td>
<td>LDBL_EPSILON</td>
<td>SCHAR_MIN</td>
</tr>
<tr>
<td>DBL_DIG</td>
<td>FLT_MAX</td>
<td>LDBL_MANT_DIG</td>
<td>SHRT_MAX</td>
</tr>
<tr>
<td>DBL_EPSILON</td>
<td>FLT_MAX_10_EXP</td>
<td>LDBL_MAX</td>
<td>SHRT_MIN</td>
</tr>
<tr>
<td>DBL_MANT_DIG</td>
<td>FLT_MAX_EXP</td>
<td>LDBL_MAX_10_EXP</td>
<td>UCHAR_MAX</td>
</tr>
<tr>
<td>DBL_MAX</td>
<td>FLT_MIN</td>
<td>LDBL_MAX_EXP</td>
<td>UINT_MAX</td>
</tr>
<tr>
<td>DBL_MAX_10_EXP</td>
<td>FLT_MIN_10_EXP</td>
<td>LDBL_MIN</td>
<td>ULONG_MAX</td>
</tr>
<tr>
<td>DBL_MAX_EXP</td>
<td>FLT_MIN_EXP</td>
<td>LDBL_MIN_10_EXP</td>
<td>USHRT_MAX</td>
</tr>
<tr>
<td>DBL_MIN</td>
<td>FLT_RADIX</td>
<td>LDBL_MIN_EXP</td>
<td></td>
</tr>
<tr>
<td>DBL_MIN_10_EXP</td>
<td>FLT_ROUNDS</td>
<td>LONG_MAX</td>
<td></td>
</tr>
<tr>
<td>DBL_MIN_EXP</td>
<td>INT_MAX</td>
<td>LONG_MIN</td>
<td></td>
</tr>
</tbody>
</table>

Table 152 — Standard types

| clock_t | ldiv_t | size_t <cstdio> | va_list |
| div_t | mbstate_t | size_t <cstdlib> | wctrans_t |
| FILE | ptdiff_t | size_t <cstring> | wctype_t |
| fpos_t | sig_atomic_t | size_t <cstring> | wint_t <cwchar> |
| jmp_buf | size_t <cstddef> | time_t <ctime> | wint_t <cwctype> |

Table 153 — Standard structs

lconv tm

C.5.1 Modifications to headers

For compatibility with the Standard C library, the C++ standard library provides the C headers enumerated in D.5, but their use is deprecated in C++.

C.5.2 Modifications to definitions

C.5.2.1 Types char16_t and char32_t

The types char16_t and char32_t are distinct types rather than typedefs to existing integral types.

C.5.2.2 Type wchar_t

wchar_t is a keyword in this International Standard (2.11). It does not appear as a type name defined in any of <cstdlib>, <cstring>, <ctime>, or <cwchar> (21.8).

C.5.2.3 Header <iso646.h>

The tokens and, and_eq, bitand, bitor, compl, not_eq, not, or, or_eq, xor, and xor_eq are keywords in this International Standard (2.11). They do not appear as macro names defined in <ciso646>.

C.5.2.4 Macro NULL

The macro NULL, defined in any of <locale>, <cstdlib>, <cstdio>, <cstring>, <ctime>, or <cwchar>, is an implementation-defined C++ null pointer constant in this International Standard (18.2).

C.5.3 Modifications to declarations

Header <cstring>:

The following functions have different declarations:
| C.5.3 1251 |

Table 154 — Standard functions

<table>
<thead>
<tr>
<th>abort</th>
<th>fmod</th>
<th>iswalnum</th>
<th>modf</th>
<th>strlen</th>
<th>wcsstrcat</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>fopen</td>
<td>iswalpha</td>
<td>perror</td>
<td>strncat</td>
<td>wcschr</td>
</tr>
<tr>
<td>acos</td>
<td>fprintf</td>
<td>iswcntrl</td>
<td>pow</td>
<td>strncmp</td>
<td>wcscmp</td>
</tr>
<tr>
<td>asctime</td>
<td>fprintf</td>
<td>iswctype</td>
<td>printf</td>
<td>strncpy</td>
<td>wcscoll</td>
</tr>
<tr>
<td>asin</td>
<td>fputs</td>
<td>iswdigit</td>
<td>putc</td>
<td>strpbrk</td>
<td>wcscpy</td>
</tr>
<tr>
<td>atan</td>
<td>fputs</td>
<td>iswgraph</td>
<td>putchar</td>
<td>strchr</td>
<td>wcscspn</td>
</tr>
<tr>
<td>atan2</td>
<td>fputs</td>
<td>iswlower</td>
<td>puts</td>
<td>strspn</td>
<td>wcftime</td>
</tr>
<tr>
<td>atexit</td>
<td>fread</td>
<td>iswprintf</td>
<td>putwc</td>
<td>strstr</td>
<td>wcslen</td>
</tr>
<tr>
<td>atof</td>
<td>free</td>
<td>iswpunct</td>
<td>putwchar</td>
<td>strtod</td>
<td>wcscat</td>
</tr>
<tr>
<td>atoi</td>
<td>freopen</td>
<td>iswspace</td>
<td>qsort</td>
<td>strtok</td>
<td>wcscncmp</td>
</tr>
<tr>
<td>atol</td>
<td>frexp</td>
<td>iswupper</td>
<td>raise</td>
<td>strtol</td>
<td>wcscpy</td>
</tr>
<tr>
<td>bsearch</td>
<td>fscanf</td>
<td>iswxdigit</td>
<td>rand</td>
<td>strtol</td>
<td>wcspbrk</td>
</tr>
<tr>
<td>btowc</td>
<td>fseek</td>
<td>iswxdigit</td>
<td>realloc</td>
<td>strxfrm</td>
<td>wcsrchr</td>
</tr>
<tr>
<td>calloc</td>
<td>fseek</td>
<td>labs</td>
<td>remove</td>
<td>swprintf</td>
<td>wcsrtombs</td>
</tr>
<tr>
<td>ceil</td>
<td>ftell</td>
<td>ldexp</td>
<td>rename</td>
<td>swscanf</td>
<td>wcsspn</td>
</tr>
<tr>
<td>clearerr</td>
<td>fwide</td>
<td>div</td>
<td>rewind</td>
<td>system</td>
<td>wcsmx</td>
</tr>
<tr>
<td>clock</td>
<td>fprintf</td>
<td>localeconv</td>
<td>scanf</td>
<td>tan</td>
<td>wcstod</td>
</tr>
<tr>
<td>cos</td>
<td>fwrite</td>
<td>localtime</td>
<td>setbuf</td>
<td>tanh</td>
<td>wcstok</td>
</tr>
<tr>
<td>cosh</td>
<td>fscanf</td>
<td>log</td>
<td>setlocale</td>
<td>time</td>
<td>wcstol</td>
</tr>
<tr>
<td>ctime</td>
<td>getc</td>
<td>log10</td>
<td>setvbuf</td>
<td>tmpfile</td>
<td>wcstombs</td>
</tr>
<tr>
<td>difftime</td>
<td>getchar</td>
<td>longjmp</td>
<td>signal</td>
<td>tmpnam</td>
<td>wcstoul</td>
</tr>
<tr>
<td>div</td>
<td>getenv</td>
<td>malloc</td>
<td>sin</td>
<td>tolower</td>
<td>wcxfmm</td>
</tr>
<tr>
<td>exit</td>
<td>getwc</td>
<td>mbcbd</td>
<td>sinh</td>
<td>toupper</td>
<td>wtob</td>
</tr>
<tr>
<td>exp</td>
<td>getwchar</td>
<td>mbrienc</td>
<td>sprintf</td>
<td>towtrans</td>
<td>wctomb</td>
</tr>
<tr>
<td>fabs</td>
<td>gmtime</td>
<td>mbtowc</td>
<td>sqrt</td>
<td>tolower</td>
<td>wctrans</td>
</tr>
<tr>
<td>fclose</td>
<td>isalnum</td>
<td>mbsinit</td>
<td>srand</td>
<td>towupper</td>
<td>wctype</td>
</tr>
<tr>
<td>feof</td>
<td>isalpha</td>
<td>mbsrtowcs</td>
<td>sscanf</td>
<td>ungetc</td>
<td>wmemchr</td>
</tr>
<tr>
<td>ferror</td>
<td>iscntrl</td>
<td>mbstowcs</td>
<td>strcat</td>
<td>ungetwc</td>
<td>wmemcmp</td>
</tr>
<tr>
<td>fflush</td>
<td>isdigit</td>
<td>mbtowc</td>
<td>strchr</td>
<td>vfprintf</td>
<td>wmemcmpy</td>
</tr>
<tr>
<td>fgetc</td>
<td>isdigit</td>
<td>memchr</td>
<td>strcmp</td>
<td>vfwprintf</td>
<td>wmemmove</td>
</tr>
<tr>
<td>fgetpos</td>
<td>islower</td>
<td>memcmp</td>
<td>strcoll</td>
<td>vfprintf</td>
<td>wmemset</td>
</tr>
<tr>
<td>fgetw</td>
<td>isprint</td>
<td>memcpyp</td>
<td>strcspn</td>
<td>vsprintf</td>
<td>wprintf</td>
</tr>
<tr>
<td>fgetwc</td>
<td>ispunct</td>
<td>memmove</td>
<td>strcspn</td>
<td>vsprintf</td>
<td>wscanf</td>
</tr>
<tr>
<td>fgetws</td>
<td>isspace</td>
<td>memmove</td>
<td>strerror</td>
<td>vsprintf</td>
<td>wscanf</td>
</tr>
<tr>
<td>floor</td>
<td>isupper</td>
<td>mktime</td>
<td>strftime</td>
<td>wcrtomb</td>
<td>wctomb</td>
</tr>
</tbody>
</table>
(1.1) — strchr
(1.2) — strpbrk
(1.3) — strrchr
(1.4) — strstr
(1.5) — memchr

21.8 describes the changes.

C.5.4 Modifications to behavior [diff.mods.to.behavior]

1 Header <cstdlib>: The following functions have different behavior:

(1.1) — atexit
(1.2) — exit
(1.3) — abort

18.5 describes the changes.

2 Header <csetjmp>: The following functions have different behavior:

(2.1) — longjmp

18.10 describes the changes.

C.5.4.1 Macro offsetof(type, member-designator) [diff.offsetof]

1 The macro offsetof, defined in <cstddef>, accepts a restricted set of type arguments in this International Standard. 18.2 describes the change.

C.5.4.2 Memory allocation functions [diff.malloc]

1 The functions malloc, calloc, and realloc are restricted in this International Standard. 20.7.13 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 the Standard, but having been identified as a candidate for removal from future revisions. An implementation may declare library names and entities described in this section with the deprecated attribute (7.6.5).

D.1 Increment operator with bool operand
1 The use of an operand of type bool with the ++ operator is deprecated (see 5.3.2 and 5.2.6).

D.2 register keyword
1 The use of the register keyword as a storage-class-specifier (7.1.1) is deprecated.

D.3 Implicit declaration of copy functions
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 (12.4, 12.8). In a future revision of this International Standard, these implicit definitions could become deleted (8.4).

D.4 Dynamic exception specifications
1 The use of dynamic-exception-specifications is deprecated.

D.5 C standard library headers
1 For compatibility with the C standard library and the C Unicode TR, the C++ standard library provides the 26 C headers, as shown in Table 155.

| <assert.h> | <inttypes.h> | <signal.h> | <stdio.h> | <wchar.h> |
| <complex.h> | <iso646.h> | <stdalign.h> | <stdlib.h> | <wctype.h> |
| <ctype.h> | <limits.h> | <stdarg.h> | <string.h> | |
| <errno.h> | <locale.h> | <stdbool.h> | <tgmath.h> | |
| <fenv.h> | <math.h> | <stddef.h> | <time.h> | |
| <float.h> | <setjmp.h> | <stdint.h> | <uchar.h> | |

2 Every 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 cname header is placed within the global namespace scope. It is unspecified whether these names are first declared or defined within namespace scope (3.3.6) of the namespace std and are then injected into the global namespace scope by explicit using-declarations (7.3.3).

3 [Example: The header <stdlib.h> 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 Old iostreams members

The following member names are in addition to names specified in Clause 27:

```cpp
namespace std {
    class ios_base {
    public:
        typedef T1 io_state;
        typedef T2 open_mode;
        typedef T3 seek_dir;
        typedef implementation-defined streamoff;
        typedef implementation-defined streampos;
        // remainder unchanged
    };
}
```

1. The type `io_state` is a synonym for an integer type (indicated here as `T1`) that permits certain member functions to overload others on parameters of type `iosate` and provide the same behavior.

2. The type `open_mode` is a synonym for an integer type (indicated here as `T2`) that permits certain member functions to overload others on parameters of type `openmode` and provide the same behavior.

3. The type `seek_dir` is a synonym for an integer type (indicated here as `T3`) that permits certain member functions to overload others on parameters of type `seekdir` and provide the same behavior.

4. The type `streamoff` is an implementation-defined type that satisfies the requirements of `off_type` in 27.2.2.

5. The type `streampos` is an implementation-defined type that satisfies the requirements of `pos_type` in 27.2.2.

6. An implementation may provide the following additional member function, which has the effect of calling `sbumpc()` (76.3.2.3):

```cpp
namespace std {
    template<class charT, class traits = char_traits<charT> >
    class basic_streambuf {
    public:
        void stossc();
        // remainder unchanged
    };
}
```

8. An implementation may provide the following member functions that overload signatures specified in Clause 27:

```cpp
namespace std {
    template<class charT, class traits> class basic_ios {
    public:
        void clear(io_state state);
        void setstate(io_state state);
        void exceptions(io_state);
        // remainder unchanged
    };
}
```

```cpp
class ios_base {
    public:
        // remainder unchanged
    };
```

```cpp
template<class charT, class traits = char_traits<charT> >
class basic_streambuf {
    public:
```
pos_type pubseekoff(off_type off, ios_base::seek_dir way,
   ios_base::open_mode which = ios_base::in | ios_base::out);
pos_type pubseekpos(pos_type sp,
   ios_base::open_mode which);
   // remainder unchanged
};

template <class charT, class traits = char_traits<charT> >
class basic_filebuf : public basic_streambuf<charT,traits> {
 public:
   basic_filebuf<charT,traits>* open
      (const char* s, ios_base::open_mode mode);
      // remainder unchanged
};

template <class charT, class traits = char_traits<charT> >
class basic_ifstream : public basic_istream<charT,traits> {
 public:
   void open(const char* s, ios_base::open_mode mode);
      // remainder unchanged
};

template <class charT, class traits = char_traits<charT> >
class basic_ofstream : public basic_ostream<charT,traits> {
 public:
   void open(const char* s, ios_base::open_mode mode);
      // remainder unchanged
};

§ 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 = 0);
      strstreambuf(signed char* gnext_arg, streamsize n,
               signed char* pbeg_arg = 0);
      strstreambuf(const signed char* gnext_arg, streamsize n);
      strstreambuf(const signed char* gnext_arg, streamsize n,
               signed char* pbeg_arg = 0);
      strstreambuf(const unsigned char* gnext_arg, streamsize n);
      strstreambuf(const unsigned char* gnext_arg, streamsize n,
               unsigned char* pbeg_arg = 0);
      strstreambuf(const unsigned char* gnext_arg, streamsize n);

      virtual -strstreambuf();
   }
The class `strstreambuf` associates the input sequence, and possibly the output sequence, with an object of some character array type, whose elements store arbitrary values. The array object has several attributes.

1. For the sake of exposition, these are represented as elements of a bitmask type (indicated here as `T1`) called `strstate`. The elements are:

   - **allocated**, set when a dynamic array object has been allocated, and hence should be freed by the destructor for the `strstreambuf` object;
   - **constant**, set when the array object has `const` elements, so the output sequence cannot be written;
   - **dynamic**, set when the array object is allocated (or reallocated) as necessary to hold a character sequence that can change in length;
   - **frozen**, set when the program has requested that the array object not be altered, reallocated, or freed.

   — *end note*

2. For the sake of exposition, the maintained data is presented here as:

   - `strstate strmode`, the attributes of the array object associated with the `strstreambuf` object;
   - `int alsize`, 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.

§ D.7.1 1256
Each object of class `strstreambuf` has a *seekable area*, delimited by the pointers `seeklow` and `seekhigh`. If `gnext` is a null pointer, the seekable area is undefined. Otherwise, `seeklow` equals `gbeg` and `seekhigh` is either `pend`, if `pend` is not a null pointer, or `gend`.

D.7.1.1 `strstreambuf` constructors

```c
explicit strstreambuf(streamsize alsize_arg = 0);
```

*Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 156.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strmode</code></td>
<td>dynamic</td>
</tr>
<tr>
<td><code>alsize</code></td>
<td><code>alsize_arg</code></td>
</tr>
<tr>
<td><code>palloc</code></td>
<td>a null pointer</td>
</tr>
<tr>
<td><code>pfree</code></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 `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 157.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strmode</code></td>
<td>dynamic</td>
</tr>
<tr>
<td><code>alsize</code></td>
<td>an unspecified value</td>
</tr>
<tr>
<td><code>palloc</code></td>
<td><code>palloc_arg</code></td>
</tr>
<tr>
<td><code>pfree</code></td>
<td><code>pfree_arg</code></td>
</tr>
</tbody>
</table>

```c
strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = 0);
strstreambuf(signed char* gnext_arg, streamsize n,
signed char* pbeg_arg = 0);
strstreambuf(unsigned char* gnext_arg, streamsize n,
unsigned char* pbeg_arg = 0);
```

*Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 158.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strmode</code></td>
<td>0</td>
</tr>
<tr>
<td><code>alsize</code></td>
<td>an unspecified value</td>
</tr>
<tr>
<td><code>palloc</code></td>
<td>a null pointer</td>
</tr>
<tr>
<td><code>pfree</code></td>
<td>a null pointer</td>
</tr>
</tbody>
</table>

`gnext_arg` shall point to the first element of an array object whose number of elements \( N \) is determined as follows:
If \( n > 0 \), \( N \) is \( n \).

If \( n == 0 \), \( N \) is \( \text{std::strlen(gnext_arg)} \).

If \( n < 0 \), \( N \) is \( \text{INT_MAX} \).

If \( \text{pbeg\_arg} \) is a null pointer, the function executes:

\[
\text{setg(gnext\_arg, gnext\_arg, gnext\_arg + N)};
\]

Otherwise, the function executes:

\[
\text{setg(gnext\_arg, gnext\_arg, pbeg\_arg);}\\
\text{setp(pbeg\_arg, pbeg\_arg + N)};
\]

\[
\text{strstreambuf(const char* gnext\_arg, streamsize n);}\\
\text{strstreambuf(const signed char* gnext\_arg, streamsize n);}\\
\text{strstreambuf(const unsigned char* gnext\_arg, streamsize n)};
\]

Effects: Behaves the same as \( \text{strstreambuf((char*)gnext\_arg, n)} \), except that the constructor also sets constant in \( \text{strmode} \).

virtual \(-\text{strstreambuf}()\);

Effects: Destroys an object of class \( \text{strstreambuf} \). The function frees the dynamically allocated array object only if \( \text{strmode & allocated} != 0 \) and \( \text{strmode & frozen} == 0 \). (\D.7.1.3\) describes how a dynamically allocated array object is freed.)

D.7.1.2 Member functions

\[
\text{void freeze(bool freezefl = true);}\\
\]

Effects: If \( \text{strmode & dynamic} \) is non-zero, alters the freeze status of the dynamic array object as follows:

(1.1) If \( \text{freezefl} \) is \( \text{true} \), the function sets \( \text{frozen} \) in \( \text{strmode} \).

(1.2) Otherwise, it clears \( \text{frozen} \) in \( \text{strmode} \).

\[
\text{char* str();}\\
\]

Effects: Calls \( \text{freeze()} \), then returns the beginning pointer for the input sequence, \( \text{gbeg} \).

Remarks: The return value can be a null pointer.

\[
\text{int pcount() const;}\\
\]

Effects: If the next pointer for the output sequence, \( \text{pnext} \), 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, \( \text{pnext} - \text{pbeg} \).

D.7.1.3 \text{strstreambuf overridden virtual functions}

\[
\text{int\_type overflow(int\_type c = EOF);}\\
\]

Effects: Appends the character designated by \( c \) to the output sequence, if possible, in one of two ways:

(1.1) If \( c != \text{EOF} \) 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 \( *\text{pnext}++ \).

Returns \( \text{(unsigned char)}c \).

\( \text{§ D.7.1.3} \)

338) The function signature \( \text{strlen(const char*)} \) is declared in \( \text{<cstring>} \) \( \text{(21.8)} \). The macro \( \text{INT\_MAX} \) is defined in \( \text{<climits>} \) \( \text{(18.3)} \).
If \( c == \text{EOF} \), there is no character to append.

Returns a value other than \( \text{EOF} \).

Returns \( \text{EOF} \) to indicate failure.

Remarks: The function can alter the number of write positions available as a result of any call.

To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements \( n \) to hold the current array object (if any), plus at least one additional write position. How many additional write positions are made available is otherwise unspecified. If \( \text{palloc} \) is not a null pointer, the function calls \( (*\text{palloc})(n) \) to allocate the new dynamic array object. Otherwise, it evaluates the expression \( \text{new charT}[n] \). In either case, if the allocation fails, the function returns \( \text{EOF} \). Otherwise, it sets \( \text{allocated} \) in \( \text{strmode} \).

To free a previously existing dynamic array object whose first element address is \( p \): If \( \text{pfree} \) is not a null pointer, the function calls \( (*\text{pfree})(p) \). Otherwise, it evaluates the expression \( \text{delete[]p} \).

If \( \text{strmode} \& \text{dynamic} == 0 \), or if \( \text{strmode} \& \text{frozen} != 0 \), the function cannot extend the array (realloc it with greater length) to make a write position available.

\[
\text{int_type pbackfail(int_type c = EOF);}\
\]

Puts back the character designated by \( c \) to the input sequence, if possible, in one of three ways:

- If \( c != \text{EOF} \), if the input sequence has a putback position available, and if \( (\text{char})c == \text{gnext}[-1] \), assigns \( \text{gnext} - 1 \) to \( \text{gnext} \).
  
  Returns \( c \).

- If \( c != \text{EOF} \), if the input sequence has a putback position available, and if \( \text{strmode} \& \text{constant} \) is zero, assigns \( c \) to \( *--\text{gnext} \).
  
  Returns \( c \).

- If \( c == \text{EOF} \) and if the input sequence has a putback position available, assigns \( \text{gnext} - 1 \) to \( \text{gnext} \).
  
  Returns a value other than \( \text{EOF} \).

Returns \( \text{EOF} \) to indicate failure.

Remarks: If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

\[
\text{int_type underflow();}\
\]

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 \( (\text{unsigned char})*\text{gnext} \).

- Otherwise, if the current write next pointer \( \text{pnext} \) is not a null pointer and is greater than the current read end pointer \( \text{gend} \), makes a read position available by assigning to \( \text{gend} \) a value greater than \( \text{gnext} \) and no greater than \( \text{pnext} \).

  Returns \( (\text{unsigned char}*)\text{gnext} \).

Returns \( \text{EOF} \) to indicate failure.

Remarks: The function can alter the number of read positions available as a result of any call.

\[
\text{pos_type seekoff(off_type off, seekdir way, openmode which = in | out);}\
\]
Table 159 — seekoff positioning

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(which &amp; ios::in) != 0</td>
<td>positions the input sequence</td>
</tr>
<tr>
<td>(which &amp; ios::out) != 0</td>
<td>positions the output sequence</td>
</tr>
<tr>
<td>(which &amp; (ios::in</td>
<td>ios::out)) == (ios::in</td>
</tr>
<tr>
<td>Otherwise</td>
<td>the positioning operation fails.</td>
</tr>
</tbody>
</table>

**Effects:** Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 159.

For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines `newoff` as indicated in Table 160.

Table 160 — newoff values

<table>
<thead>
<tr>
<th>Condition</th>
<th>newoff Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>way == ios::beg</td>
<td>0</td>
</tr>
<tr>
<td>way == ios::cur</td>
<td>the next pointer minus the beginning pointer (xnext - xbeg).</td>
</tr>
<tr>
<td>way == ios::end</td>
<td>seekhigh minus the beginning pointer (seekhigh - xbeg).</td>
</tr>
</tbody>
</table>

If (newoff + off) < (seeklow - xbeg) or (seekhigh - xbeg) < (newoff + off), the positioning operation fails. Otherwise, the function assigns `xbeg + newoff + off` to the next pointer `xnext`.

**Returns:** `pos_type(newoff)`, constructed from the resultant offset `newoff` (of type `off_type`), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is `pos_type(off_type(-1))`.

```cpp
pos_type seekpos(pos_type sp, ios_base::openmode which = ios_base::in | ios_base::out);
```

**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).

1. (17.1) If (which & ios::in) != 0, positions the input sequence.
2. (17.2) If (which & ios::out) != 0, positions the output sequence.
3. (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()`.

1. (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

An implementation should consider `alsize` in making this decision.
Otherwise, the function adds newoff to the beginning pointer xbeg and stores the result in the next pointer xnext.

19

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);  

20

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();

        private:
            strstreambuf sb; // exposition only
    };
}

1 The class istrstream supports the reading of objects of class strstreambuf. It supplies a strstreambuf object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

(1.1) — sb, the strstreambuf object.

D.7.2.1 istrstream constructors

explicit istrstream(const char* s);
explicit istrstream(char* s);

1 Effects: Constructs an object of class istrstream, initializing the base class with istream(&sb) and initializing sb with strstreambuf(s,0). s shall designate the first element of an NTBS.

istrstream(const char* s, streamsize n);

2 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

strstreambuf* rdbuf() const;

1 Returns: const_cast<strstreambuf*>(&sb).

char* str();

2 Returns: rdbuf()->str().
D.7.3 Class ostrstream

namespace std {
    class ostrstream : public basic_ostream<char> {
        public:
            ostrstream();
            ostrstream(char* s, int n, ios_base::openmode mode = ios_base::out);
            virtual ~ostrstream();

            strstreambuf* rdbuf() const;
            void freeze(bool freezefl = true);
            char* str();
            int pcount() const;
        private:
            strstreambuf sb;  // exposition only
        }
    }

1 The class ostrstream supports the writing of objects of class strstreambuf. It supplies a strstreambuf object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

(1.1) — sb, the strstreambuf object.

D.7.3.1 ostrstream constructors

1 ostrstream();

   Effects: Constructs an object of class ostrstream, initializing the base class with ostream(&sb) and initializing sb with strstreambuf().

2 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)).\[340\]

D.7.3.2 Member functions

1 strstreambuf* rdbuf() const;

   Returns: (strstreambuf*)&sb.

2 void freeze(bool freezefl = true);

   Effects: Calls rdbuf()->freeze(freezefl).\[\]

3 char* str();

   Returns: rdbuf()->str().

4 int pcount() const;

   Returns: rdbuf()->pcount().\[\]

340) The function signature strlen(const char*) is declared in <cstring> (21.8).

§ D.7.3.2
D.7.4 Class strstream

namespace std {
    class strstream : public basic_iostream<char> {
        public:
            // Types
            typedef char char_type;
            typedef typename char_traits<char>::int_type int_type;
            typedef typename char_traits<char>::pos_type pos_type;
            typedef typename char_traits<char>::off_type off_type;

            // constructors/destructor
            strstream();
            strstream(char* s, int n, ios_base::openmode mode = ios_base::in|ios_base::out);
            virtual ~strstream();

            // Members:
            strstreambuf* rdbuf() const;
            void freeze(bool freezeFl = true);
            int pcount() const;
            char* str();
        private:
            strstreambuf sb; // exposition only
        }
    }

1 The class strstream supports reading and writing from objects of class strstreambuf. It supplies a strstreambuf object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

(1.1) — sb, the strstreambuf object.

D.7.4.1 strstream constructors

strstream();

1 Effects: Constructs an object of class strstream, initializing the base class with iostream(&sb).

strstream(char* s, int n, ios_base::openmode mode = ios_base::in|ios_base::out);

2 Effects: Constructs an object of class strstream, initializing the base class with iostream(&sb) and initializing sb with one of the two constructors:

(2.1) — If (mode & app) == 0, then s shall designate the first element of an array of n elements. The constructor is strstreambuf(s,n,s).

(2.2) — If (mode & app) ! = 0, then s shall designate the first element of an array of n elements that contains an NTBS whose first element is designated by s. The constructor is strstreambuf(s,n,s + std::strlen(s)).

D.7.4.2 strstream destructor

virtual ~strstream();

1 Effects: Destroys an object of class strstream.

§ D.7.4.2
strstreambuf* rdbuf() const;

Returns: &sb.

D.7.4.3 strstream operations

void freeze(bool freezefl = true);

Effects: Calls rdbuf()->freeze(freezefl).

char* str();

Returns: rdbuf()->str().

int pcount() const;

Returns: rdbuf()->pcount().

D.8 Violating exception-specifications

D.8.1 Type unexpected_handler

typedef void (*unexpected_handler)();

The type of a handler function to be called by unexpected() when a function attempts to throw an exception not listed in its dynamic-exception-specification.

Required behavior: An unexpected_handler shall not return. See also 15.5.2.

Default behavior: The implementation’s default unexpected_handler calls std::terminate().

D.8.2 set_unexpected

unexpected_handler set_unexpected(unexpected_handler f) noexcept;

Effects: Establishes the function designated by f as the current unexpected_handler.

Remark: It is unspecified whether a null pointer value designates the default unexpected_handler.

Returns: The previous unexpected_handler.

D.8.3 get_unexpected

unexpected_handler get_unexpected() noexcept;

Returns: The current unexpected_handler. [Note: This may be a null pointer value. — end note]

D.8.4 unexpected

[[noreturn]] void unexpected();

Remarks: Called by the implementation when a function exits via an exception not allowed by its exception-specification (15.5.2), in effect after evaluating the throw-expression (D.8.1). May also be called directly by the program.

Effects: Calls the current unexpected_handler function. [Note: A default unexpected_handler is always considered a callable handler in this context. — end note]

D.9 uncaught_exception

bool uncaught_exception() noexcept;

Returns: uncaught_exceptions() > 0.
Annex E  (normative)
Universal character names for identifier characters

E.1 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
3004-3007, 3021-302F, 3031-303F
3040-D7FF
F900-FD3D, FD40-FDCF, FDFO-FE44, FE47-FFFF
10000-1FFFF, 20000-2FFFF, 30000-3FFFF, 40000-4FFFF, 50000-5FFFF,
60000-6FFFF, 70000-7FFFF, 80000-8FFFF, 90000-9FFFF, A0000-AFFFF,
B0000-BFFFF, C0000-CFFFF, D0000-DFFFF, E0000-EFFFF

E.2 Ranges of characters disallowed initially
0300-036F, 1DC0-1DFF, 20D0-20FF, FE20-FE2F
Annex F  (informative)
Cross references

This annex lists each section label and the corresponding section number, in alphabetical order by label. All of the section labels are the same as in the 2003 standard, except:

— labels that begin with `lib.` in the 2003 standard have had the `lib.` removed so that they do not all appear in the same part of this list. For example, in the 2003 standard, the non-modifying sequence algorithms were found in a section with the label `[lib.alg.nonmodifying]`. The label for that section is now `[alg.nonmodifying]`.

— the label for Annex B has been changed from `[limits]` to `[impllimits]`. The label `[limits]` refers to section 18.3.2.

<table>
<thead>
<tr>
<th>Section Label</th>
<th>Section Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>accumulate</td>
<td>26.7.2</td>
</tr>
<tr>
<td>adjacent.difference</td>
<td>26.7.5</td>
</tr>
<tr>
<td>adjustfield.manip</td>
<td>27.5.6.2</td>
</tr>
<tr>
<td>alg.adjacent.find</td>
<td>25.2.8</td>
</tr>
<tr>
<td>alg.all_of</td>
<td>25.2.1</td>
</tr>
<tr>
<td>alg.any_of</td>
<td>25.2.2</td>
</tr>
<tr>
<td>alg.binary.search</td>
<td>25.4.3</td>
</tr>
<tr>
<td>alg.c.library</td>
<td>25.5</td>
</tr>
<tr>
<td>alg.copy</td>
<td>25.3.1</td>
</tr>
<tr>
<td>alg.count</td>
<td>25.2.9</td>
</tr>
<tr>
<td>alg.equal</td>
<td>25.2.11</td>
</tr>
<tr>
<td>alg.find</td>
<td>25.2.5</td>
</tr>
<tr>
<td>alg.find.end</td>
<td>25.2.6</td>
</tr>
<tr>
<td>alg.find.first.of</td>
<td>25.2.7</td>
</tr>
<tr>
<td>alg.foreach</td>
<td>25.2.4</td>
</tr>
<tr>
<td>alg.generate</td>
<td>25.3.7</td>
</tr>
<tr>
<td>alg.heap.operations</td>
<td>25.4.6</td>
</tr>
<tr>
<td>alg.is_permutation</td>
<td>25.2.12</td>
</tr>
<tr>
<td>alg.lex.comparison</td>
<td>25.4.8</td>
</tr>
<tr>
<td>alg.merge</td>
<td>25.4.4</td>
</tr>
<tr>
<td>alg.min.max</td>
<td>25.4.7</td>
</tr>
<tr>
<td>alg.modifying.operations</td>
<td>25.3</td>
</tr>
<tr>
<td>alg.move</td>
<td>25.3.2</td>
</tr>
<tr>
<td>alg.none_of</td>
<td>25.2.3</td>
</tr>
<tr>
<td>alg.nonmodifying</td>
<td>25.2</td>
</tr>
<tr>
<td>alg.nth.element</td>
<td>25.4.2</td>
</tr>
<tr>
<td>alg.partitions</td>
<td>25.3.13</td>
</tr>
<tr>
<td>alg.permutation.generators</td>
<td>25.4.9</td>
</tr>
<tr>
<td>alg.random.shuffle</td>
<td>25.3.12</td>
</tr>
<tr>
<td>alg.remove</td>
<td>25.3.8</td>
</tr>
<tr>
<td>alg.replace</td>
<td>25.3.5</td>
</tr>
<tr>
<td>alg.reverse</td>
<td>25.3.10</td>
</tr>
<tr>
<td>alg.rotate</td>
<td>25.3.11</td>
</tr>
<tr>
<td>alg.search</td>
<td>25.2.13</td>
</tr>
<tr>
<td>alg.set.operations</td>
<td>25.4.5</td>
</tr>
<tr>
<td>alg.sort</td>
<td>25.4.1</td>
</tr>
<tr>
<td>alg.sorting</td>
<td>25.4</td>
</tr>
<tr>
<td>alg.swap</td>
<td>25.3.3</td>
</tr>
<tr>
<td>alg.transform</td>
<td>25.3.4</td>
</tr>
<tr>
<td>alg.unique</td>
<td>25.3.9</td>
</tr>
<tr>
<td>algorithm.stable</td>
<td>17.6.5.7</td>
</tr>
<tr>
<td>algorithms</td>
<td>25</td>
</tr>
<tr>
<td>algorithms.general</td>
<td>25.1</td>
</tr>
<tr>
<td>alloc.errors</td>
<td>18.6.2</td>
</tr>
<tr>
<td>allocator.adaptor</td>
<td>20.13</td>
</tr>
<tr>
<td>allocator.adaptor.cnstr</td>
<td>20.13.3</td>
</tr>
<tr>
<td>allocator.adaptor.members</td>
<td>20.13.4</td>
</tr>
<tr>
<td>allocator.adaptor.syn</td>
<td>20.13.1</td>
</tr>
<tr>
<td>allocator.adaptor.types</td>
<td>20.13.2</td>
</tr>
<tr>
<td>allocator.globals</td>
<td>20.7.9.2</td>
</tr>
<tr>
<td>allocator.members</td>
<td>20.7.9.1</td>
</tr>
<tr>
<td>allocator.requirements</td>
<td>17.6.3.5</td>
</tr>
<tr>
<td>allocator.tag</td>
<td>20.7.6</td>
</tr>
<tr>
<td>allocator.traits</td>
<td>20.7.8</td>
</tr>
<tr>
<td>allocator.traits.members</td>
<td>20.7.8.2</td>
</tr>
<tr>
<td>allocator.traits.types</td>
<td>20.7.8.1</td>
</tr>
<tr>
<td>allocator.uses</td>
<td>20.7.7</td>
</tr>
<tr>
<td>allocator.uses.construction</td>
<td>20.7.7.2</td>
</tr>
<tr>
<td>allocator.uses.trait</td>
<td>20.7.7.1</td>
</tr>
<tr>
<td>alt.headers</td>
<td>17.6.4.4</td>
</tr>
<tr>
<td>arithmetic.operations</td>
<td>20.9.5</td>
</tr>
<tr>
<td>array</td>
<td>23.3.2</td>
</tr>
<tr>
<td>array.cons</td>
<td>23.3.2.2</td>
</tr>
</tbody>
</table>
array.data 23.3.2.5
array.fill 23.3.2.6
array.overview 23.3.2.1
array.size 23.3.2.4
array.special 23.3.2.3
array.swap 23.3.2.7
array.tuple 23.3.2.9
array.zero 23.3.2.8
assertions 19.3
associative 23.4
associative.general 23.4.1
associative.map.syn 23.4.2
associative.reqmts 23.4.4
associative.reqmts.except 23.2.4.1
associative.set.syn 23.4.3
atomics 29
atomics.fences 29.8
atomics.flag 29.7
atomics.general 29.1
atomics.lockfree 29.4
atomics.order 29.3
atomics.syn 29.2
atomics.types.generic 29.5
atomics.types.operations 29.6
atomics.types.operations.arith 29.6.3
atomics.types.operations.general 29.6.1
atomics.types.operations.pointer 29.6.4
atomics.types.operations.req 29.6.5
atomics.types.operationstempl 29.6.2

basic.ios.cons 27.5.5.2
basic.ios.members 27.5.5.3
basic.life 3.8
basic.link 3.5
basic.lookup 3.4
basic.lookup.argdep 3.4.2
basic.lookup.classref 3.4.5
basic.lookup.elab 3.4.4
basic.lookup.qual 3.4.3
basic.lookup.udir 3.4.6
basic.lookup.unqual 3.4.1
basic.lval 3.10
basic.namespace 7.3
basic.scope 3.3
basic.scope.block 3.3.3
basic.scope.class 3.3.7
basic.scope.declarative 3.3.1
basic.scope.enum 3.3.8
basic.scope.hiding 3.3.10
basic.scope.namespace 3.3.6
basic.scope.pdecl 3.3.2
basic.scope.proto 3.3.4
basic.scope.temp 3.3.9
basic.start 3.6
basic.start.init 3.6.2
basic.start.main 3.6.1
basic.start.term 3.6.3
basic.stc 3.7
basic.stc.auto 3.7.3
basic.stc.dynamichandle 3.7.4
basic.stc.dynamichandle.allocation 3.7.4.1
basic.stc.dynamichandle.deallocation 3.7.4.2
basic.stc.dynamichandle.safety 3.7.4.3
basic.stc.inherit 3.7.5
basic.stc.static 3.7.1
basic.stc.thread 3.7.2
basic.string 21.4
basic.string.hash 21.6
basic.string.literals 21.7
basic.type.qualifier 3.9.3
basic.types 3.9
bidirectional.iterators 24.2.6
binary.search 25.4.3.4
bitset.types 17.5.2.1.3
bitset.cons 20.6.1
bitset.hash 20.6.3
bitset.members 20.6.2
bitset.operands 20.6.4
bitset.operations 20.9.8
byte.string 17.5.2.1.4.1

Cross references 1267
| conv        | 4        | dcl.decl  | 8     |
| conv.array  | 4.2      | dcl.enum  | 7.2   |
| conv.bool   | 4.12     | dcl.fct   | 8.3.5 |
| conv.double | 4.8      | dcl.fct.def| 8.4  |
| conv.fint   | 4.9      | dcl.fct.def.default| 8.4.2 |
| conv.fprom  | 4.6      | dcl.fct.def.delete| 8.4.3 |
| conv.func   | 4.3      | dcl.fct.def.general| 8.4.1 |
| conv.integral | 4.7    | dcl.fct.default| 8.3.6 |
| conv.lval   | 4.1      | dcl.fct.spec| 7.1.2 |
| conv.mem    | 4.11     | dcl.friend| 7.1.4 |
| conv.prom   | 4.5      | dcl.init  | 8.5   |
| conv.ptr    | 4.10     | dcl.init.aggr| 8.5.1 |
| conv.qual   | 4.4      | dcl.init.list| 8.5.4 |
| conv.rank   | 4.13     | dcl.init.ref| 8.5.3 |
| conventions | 17.5.2    | dcl.link  | 7.5   |
| conversions | 22.3.3.2  | dcl.meaning| 8.3   |
| conversions.buffer | 22.3.3.2.3 | dcl.mptr  | 8.3.3 |
| conversions.character | 22.3.3.2.1 | dcl.name  | 8.1   |
| conversions.string | 22.3.3.2.2 | dcl.ptr   | 8.3.1 |
| cpp         | 16       | dcl.ref   | 8.3.2 |
| cpp.concat  | 16.3.3   | dcl.spec  | 7.1   |
| cpp.cond    | 16.1     | default.allocate | 20.7.9 |
| cpp.error   | 16.5     | definitions| 17.3  |
| cpp.include | 16.2     | defns.additional| 17.4  |
| cpp.line    | 16.4     | defns.arbitrary.stream| 17.3 |
| cpp.null    | 16.7     | defns.argument| 1.3  |
| cpp.pragma  | 16.6     | defns.argument.macro| 1.3 |
| cpp.pragma.op | 16.9 | defns.argumenttempl | 1.3 |
| cpp.predefined | 16.8 | defns.argument.throw| 1.3 |
| cpp.replace | 16.3     | defns.block| 17.3  |
| cpp.rescan  | 16.3.4   | defns.blocked| 17.3 |
| cpp.scope   | 16.3.5   | defns.character| 17.3 |
| cpp.stringize| 16.3.2   | defns.character.container| 17.3 |
| cpp.subst   | 16.3.1   | defns.comparison| 17.3 |
| cst dint    | 18.4     | defns.component| 17.3 |
| cst dint.syn | 18.4.1   | defns.cond.supp| 1.3  |
| D           |          | defns.deadlock| 17.3  |
| date.time   | 20.12.8  | defns.default.behavior.func| 17.3 |
| dcl.align   | 7.6.2    | defns.default.behavior.impl| 17.3 |
| dcl.ambig.res | 8.2    | defns.diagnostics| 1.3  |
| dcl.array   | 8.3.4    | defns.dynamics.type| 1.3  |
| dcl.asm     | 7.4      | defns.dynamics.type.prvalue| 1.3 |
| dcl.attr    | 7.6      |        |      |
| dcl.attr.depend | 7.6.4  |        |      |
| dcl.attr.deprecated | 7.6.5 |        |      |
| dcl.attr.grammar | 7.6.1  |        |      |
| dcl.attr.noreturn | 7.6.3  |        |      |
| dcl.constexpr | 7.1.5   |        |      |
| dcl.dcl     | 7        |        |      |

Cross references 1269
diff.cpp11.dcl.dcl C.3.3
diff.cpp11.input.output C.3.4
diff.cpp11.lex C.3.1
diff.cpp14 C.4
diff.cpp14.depr C.4.2
diff.cpp14.lex C.4.1
diff.del C.1.6
diff.decl C.1.7
diff.expr C.1.4
diff.header.iso646.h C.5.2.3
diff.iso C.1
diff.lex C.1.1
diff.library C.5
diff.malloc C.5.4.2
diff.mods.to.behavior C.5.4
diff.mods.to.declarations C.5.3
diff.mods.to.declarations C.5.2
diff.mods.to.headers C.5.1
diff.null C.5.2.4
diff.ofsetof C.5.4.1
diff.special C.1.9
diff.stat C.1.5
diff.wchar.t C.5.2.2
domain.error 19.2.2

e expr.comma 5.19
e expr.cond 5.16
e expr.const 5.20
e expr.const.cast 5.2.11
e expr.delete 5.3.5
e expr.dynamic.cast 5.2.7
e expr.eq 5.10
e expr.log.and 5.14
e expr.log.or 5.15
e expr.mptr.oper 5.5
e expr.mul 5.6
e expr.new 5.3.4
e expr.or 5.13
e expr.post 5.2
e expr.post.incr 5.2.6
e expr.pre.incr 5.3.2
e expr.prim 5.1
e expr.prim.fold 5.1.3
e expr.prim.general 5.1.1
e expr.prim.lambda 5.1.2
e expr.pseudo 5.2.4
e expr.ref 5.2.5
e expr.reinterpret.cast 5.2.10
e expr.rel 5.9
e expr.shift 5.8
e expr.sizeof 5.3.3
e expr.static.cast 5.2.9
e expr.sub 5.2.1
e expr.throw 5.17
e expr.type.conv 5.2.3
e expr.typeid 5.2.8
e expr.unary 5.3
e expr.unary.noexcept 5.3.7
e expr.unary.op 5.3.1
e expr.xor 5.12
e ext.manip 27.7.5
e extern.names 17.6.4.3.2
e extern.types 17.6.4.3.3

F
facet.ctype.char.dtor 22.4.1.3.1
facet.ctype.char.members 22.4.1.3.2
facet.ctype.char.statistics 22.4.1.3.3
facet.ctype.char.virtuals 22.4.1.3.4
facet.ctype.special 22.4.1.3
facet.num.get.members 22.4.2.1.1
facet.num.get.virtauls 22.4.2.1.2
facet.num.put.members 22.4.2.2.1
facet.num.put.virtuals 22.4.2.2.2
facet.mumpunct 22.4.3

Cross references

1271
Cross references

© ISO/IEC

G
get.new.handler 18.6.2.5
get.terminate 18.8.3.3
get.unexpected D.8.3
global.functions 17.6.5.4
gram A
gram.basic A.3
gram.class A.8
gram.cpp A.14
gram.dcl A.6
gram.decl A.7
gram.derived A.9
gram.except A.13
gram.expr A.4
gram.key A.1
gram.lex A.2
gram.over A.11
gram.special A.10
gram.stmt A.5
gram.temp A.12
gslice.access 26.6.6.3
gslice.array.assign 26.6.7.2
gslice.array.comp.assign 26.6.7.3
gslice.array.fill 26.6.7.4
gslice.cons 26.6.6.2

H
Cross references
istringstream.members 27.8.3.3
iterator.basic 24.4.2
iterator.container 24.8
iterator.iterators 24.2.2
iterator.operations 24.4.4
iterator.primitives 24.4
iterator.requirements 24.2
iterator.requirements.general 24.2.1
iterator.synopsis 24.3
iterator.traits 24.4.1
iterators 24
iterators.general 24.1

J

K

L

language.support 18
length.error 19.2.4
lex 2
lex.bool 2.13.6
lex.ccon 2.13.3
lex.charset 2.3
lex.comment 2.7
lex.digraph 2.5
lex.ext 2.13.8
lex.fcon 2.13.4
lex.header 2.8
lex.icon 2.13.2
lex.key 2.11
lex.literal 2.13
lex.literal.kinds 2.13.1
lex.name 2.10
lex.nullptr 2.13.7
lex_operators 2.12
lex.phases 2.2
lex.ppnumber 2.9
lex.pptoken 2.4
lex.separate 2.1
lex.string 2.13.5
lex.token 2.6
lib.types.movedfrom 17.6.5.15
library 17
library.c 17.2
library.general 17.1
limits 18.3.2
limits.numeric 18.3.2.1
limits.syn 18.3.2.2
list 23.3.5

list.capacity 23.3.5.3
list.cons 23.3.5.2
list.modifiers 23.3.5.4
list.ops 23.3.5.5
list.overview 23.3.5.1
list.special 23.3.5.6
locale 22.3.1
locale.categories 22.4
locale.category 22.3.1.1.1
locale.codecvt 22.4.1.4
locale.codecvt.byname 22.4.1.5
locale.codecvt.members 22.4.1.4.1
locale.codecvt.virtuals 22.4.1.4.2
locale.collate 22.4.4.1
locale.collate.byname 22.4.4.2
locale.collate.members 22.4.4.1.1
locale.collate.virtuals 22.4.4.1.2
locale.cons 22.3.1.2
locale.convenience 22.3.3
locale.ctype 22.4.1.1
locale.ctype.byname 22.4.1.2
locale.ctype.members 22.4.1.1.1
locale.ctype.virtuals 22.4.1.1.2
locale.facet 22.3.1.1.2
locale.global.templates 22.3.2
locale.id 22.3.1.1.3
locale.members 22.3.1.3
locale.messages 22.4.7.1
locale.messages.byname 22.4.7.2
locale.messages.members 22.4.7.1.1
locale.messages.virtuals 22.4.7.1.2
locale.money.get 22.4.6.1
locale.money.get.members 22.4.6.1.1
locale.money.get.virtuals 22.4.6.1.2
locale.money.put 22.4.6.2
locale.money.put.members 22.4.6.2.1
locale.money.put.virtuals 22.4.6.2.2
locale.moneypunct 22.4.6.3
locale.moneypunct.byname 22.4.6.4
locale.moneypunct.members 22.4.6.3.1
locale.moneypunct.virtuals 22.4.6.3.2
locale.num.put 22.4.2.2
locale.num.get 22.4.2.1
locale.numpunct 22.4.3.1
locale.numpunct.byname 22.4.3.2
locale.operators 22.3.1.4
locale.statics 22.3.1.5
locale.stdio 22.5
locale.syn 22.2
locale.time.ctime 22.4.5.1
locale.time.getbyname 22.4.5.2
protection.within.classes  17.6.5.10
ptr.align     20.7.5
push.heap     25.4.6.1

Q
queue        23.6.3
queue.cons   23.6.3.2
queue.cons.alloc  23.6.3.3
queue.defn    23.6.3.1
queue.ops     23.6.3.4
queue.special 23.6.3.5
queue.syn     23.6.2
quoted.manip  27.7.6

R
rand         26.5
rand.adapt   26.5.4
rand.adapt.disc  26.5.4.2
rand.adapt.general  26.5.4.1
rand.adapt.ibits  26.5.4.3
rand.adapt.shuf   26.5.4.4
rand.device     26.5.6
rand.dist      26.5.8
rand.dist.bern  26.5.8.3
rand.dist.bern.bernoulli  26.5.8.3.1
rand.dist.bern.bin   26.5.8.3.2
rand.dist.bern.geo   26.5.8.3.3
rand.dist.bern.negbin 26.5.8.3.4
rand.dist.general  26.5.8.1
rand.dist.norm    26.5.8.5
rand.dist.norm.cauchy  26.5.8.5.4
rand.dist.norm.chisq  26.5.8.5.3
rand.dist.norm.f    26.5.8.5.5
rand.dist.norm.lognormal  26.5.8.5.2
rand.dist.norm.normal 26.5.8.5.1
rand.dist.norm.t     26.5.8.5.6
rand.dist.pois      26.5.8.4
rand.dist.pois.exp   26.5.8.4.2
rand.dist.pois.extreme  26.5.8.4.5
rand.dist.pois.gamma 26.5.8.4.3
rand.dist.pois.poisson 26.5.8.4.1
rand.dist.pois.weibull 26.5.8.4.4
rand.dist.samp      26.5.8.6
rand.dist.samp.discrete 26.5.8.6.1
rand.dist.samp.pconst  26.5.8.6.2
rand.dist.samp.plinear 26.5.8.6.3
rand.dist.uni       26.5.8.2
rand.dist.uni.int   26.5.8.2.1
rand.dist.uni.real  26.5.8.2.2
rand.eng          26.5.3
rand.eng.lcong    26.5.3.1
rand.eng.mers     26.5.3.2
rand.eng.sub      26.5.3.3
rand.predef       26.5.5
rand.req          26.5.1
rand.req.adapt    26.5.1.5
rand.req.dist     26.5.1.6
rand.req.eng      26.5.1.4
rand.req.gent     26.5.1.1
rand.req.seedseq  26.5.1.2
rand.req.urg      26.5.1.3
rand.synopsis     26.5.2
rand.util         26.5.7
rand.util.canonical 26.5.7.2
rand.util.seedseq 26.5.7.1
random.access.iterators 24.2.7
range.error       19.2.7
ratio             20.11
ratio.arithmetic  20.11.4
ratio.comparison  20.11.5
ratio.general     20.11.1
ratio.ratio       20.11.3
ratio.si          20.11.6
ratio.syn         20.11.2
re               28
re.alg           28.11
re.alg.match     28.11.2
re.alg.replace   28.11.4
re.alg.search    28.11.3
re.badexp        28.6
re.const          28.5
re.def           28.2
re.err           28.5.3
re.except        28.11.1
re.general       28.1
re.grammar       28.13
re.iter           28.12
re.matchflag     28.5.2
re.regex          28.8
re.regex.assign   28.8.3
re.regex.const    28.8.1
re.regex.construct 28.8.2
re.regex.locale   28.8.5
re.regex.mnswap   28.8.7.1
re.regex.nnmemb   28.8.7
re.regex.operations 28.8.4
re.regex.swap     28.8.6
re.regiter        28.12.1
re.regiter.cnstr  28.12.1.1
re.regiter.comp   28.12.1.2

Cross references 1277
structure.see.also 17.5.1.5
structure.specifications 17.5.1.4
structure.summary 17.5.1.2
support.dynamic 18.6
support.exception 18.8
support.general 18.1
support.initlist 18.9
support.initlist.access 18.9.2
support.initlist.cons 18.9.1
support.initlist.range 18.9.3
support.limits 18.3
support.limits.general 18.3.1
support.rtti 18.7
support.runtime 18.10
support.start.term 18.5
support.types 18.2
swappable.requirements 17.6.3.2
syntax 1.6
syserr 19.5
syserr.compare 19.5.4
syserr.errcat 19.5.1
syserr.errcat.derived 19.5.1.4
syserr.errcat.nonvirtuals 19.5.1.3
syserr.errcat.objects 19.5.1.5
syserr.errcat.overview 19.5.1.1
syserr.errcat.virtuals 19.5.1.2
syserr.errcode 19.5.2
syserr.errcode.constructors 19.5.2.2
syserr.errcode.modifiers 19.5.2.3
syserr.errcode.nonmembers 19.5.2.5
syserr.errcode.observers 19.5.2.4
syserr.errcode.overview 19.5.2.1
syserr.errcondition 19.5.3
syserr.errcondition.constructors 19.5.3.2
syserr.errcondition.modifiers 19.5.3.3
syserr.errcondition.nonmembers 19.5.3.5
syserr.errcondition.observers 19.5.3.4
syserr.errcondition.overview 19.5.3.1
syserr.hash 19.5.5
syserr.syserr 19.5.6
syserr.syserr.members 19.5.6.2
syserr.syserr.overview 19.5.6.1
temp.arg.type 14.3.1
temp.class 14.5.1
temp.class.order 14.5.5.2
temp.class.spec 14.5.5
temp.class.spec.match 14.5.5.1
temp.class.spec.mfunc 14.5.5.3
temp.decls 14.5
temp.deduct 14.8.2
temp.deduct.call 14.8.2.1
temp.deduct.conv 14.8.2.3
temp.deduct.decl 14.8.2.6
temp.deduct.funcaddr 14.8.2.2
temp.deduct.partial 14.8.2.4
temp.deduct.type 14.8.2.5
temp.dep 14.6.2
temp.dep.candidate 14.6.4.2
temp.dep.constexpr 14.6.2.3
temp.dep.expr 14.6.2.2
temp.dep.res 14.6.4
temp.dep.temp 14.6.2.4
temp.dep.type 14.6.2.1
temp.expl.spec 14.7.3
temp.explicit 14.7.2
temp.fct 14.5.6
temp.fct.spec 14.8
temp.friend 14.5.4
temp.func.order 14.5.6.2
temp.inject 14.6.5
temp.inst 14.7.1
temp.local 14.6.1
temp.mem 14.5.2
temp.mem.class 14.5.1.2
temp.mem.enum 14.5.1.4
temp.mem.func 14.5.1.1
temp.names 14.2
temp.nondep 14.6.3
temp.over 14.8.3
temp.over.link 14.5.6.1
temp.param 14.1
temp.point 14.6.4.1
temp.res 14.6
temp.spec 14.7
temp.static 14.5.1.3
temp.type 14.4
temp.variadic 14.5.3
template.bitset 20.6
template.array 26.6.7
template.array.overview 26.6.7.1
template.array.overview 26.6.9
template.array.overview 26.6.9.1
template.array 26.6.8

Cross references 1280
type.descriptions 17.5.2.1
unord.multiset 23.5.7
unord.multiset.cnstr 23.5.7.2
unord.multiset.overview 23.5.7.1
unord.multiset.swap 23.5.7.3
unord.req 23.2.5
unord.req.exception 23.2.5.1
unord.set 23.5.6
unord.set.cnstr 23.5.6.2
unord.set.overview 23.5.6.1
unord.set.swap 23.5.6.3
unord.set.syn 23.5.3
upper.bound 25.4.3.2
using 17.6.2
using.headers 17.6.2.2
using.linkage 17.6.2.3
using.overview 17.6.2.1
usrlit.suffix 17.6.4.3.4
util.dynamic.safety 20.7.4
util.smartptr 20.8.2
util.smartptr.enab 20.8.2.5
util.smartptr.getdeleter 20.8.2.2.10
util.smartptr.hash 20.8.2.7
util.smartptr.ownerless 20.8.2.4
util.smartptr.shared 20.8.2.2
util.smartptr.shared.assign 20.8.2.2.3
util.smartptr.shared.atomic 20.8.2.6
util.smartptr.shared.cast 20.8.2.2.9
util.smartptr.shared.chomp 20.8.2.2.7
util.smartptr.shared.const 20.8.2.2.1
util.smartptr.shared.create 20.8.2.2.6
util.smartptr.shared.dest 20.8.2.2.2
util.smartptr.shared.io 20.8.2.2.11
util.smartptr.shared.mod 20.8.2.2.4
util.smartptr.shared.obs 20.8.2.2.5
util.smartptr.shared.spec 20.8.2.2.8
util.smartptr.weak 20.8.2.3
util.smartptr.weak.assign 20.8.2.3.3
util.smartptr.weak.const 20.8.2.3.1
util.smartptr.weak.dest 20.8.2.3.2
util.smartptr.weak.mod 20.8.2.3.4
util.smartptr.weak.obs 20.8.2.3.5
util.smartptr.weak.spec 20.8.2.3.6
util.smartptr.weakptr 20.8.2.1
utilities 20
utility 20.2
utility.arg.requirements 17.6.3.1
utility.exchange 20.2.3
utility.requirements 17.6.3
utility.swap 20.2.2
V
valarray.access 26.6.2.4
valarray.assign 26.6.2.3
valarray.binary 26.6.3.1
valarray.cassign 26.6.2.7
valarray.comparison 26.6.3.2
valarray.cons 26.6.2.2
valarray.members 26.6.2.8
valarray.nonnmembers 26.6.3
valarray.range 26.6.10
valarray.special 26.6.3.4
valarray.sub 26.6.2.5
valarray.syn 26.6.1
valarray.transcend 26.6.3.3
valarray.unary 26.6.2.6
value.error.codes 17.6.5.14
vector 23.3.6

vector.bool 23.3.7
vector.capacity 23.3.6.3
vector.cons 23.3.6.2
vector.data 23.3.6.4
vector.modifiers 23.3.6.5
vector.overview 23.3.6.1
vector.special 23.3.6.6

W
wide.stream.objects 27.4.3

X
xref F

Y

Z
Index

!, see operator, logical negation
!*., see inequality operator
(), see operator, function call, see declarator, function
*, see operator, indirection, see multiplication operator, see declarator, pointer
+, see operator, unary plus, see addition operator
++., see comma operator
-, see operator, unary minus, see subtraction operator
->, see operator, class member access
->*, see pointer to member operator
=., see operator, class member access
.*, see pointer to member operator
...., see ellipsis
/, see division operator
:
   field declaration, 240
      label specifier, 137
::., see scope resolution operator
::*., see declarator, pointer to member operator
<, see less than operator
   template and, 336, 338
<=, see less than or equal to operator
<<, see left shift operator
=., see assignment operator
==., see equality operator
>, see greater than operator
>=, see greater than or equal to operator
>>, see right shift operator
?:, see conditional expression operator
[]., see operator, subscripting, see declarator, array
"..., see preprocessing directives, source-file inclusion
# operator, 430, 431
## operator, 431
#define, 430
#define, 430
#define, 430
#define, 430
#define, 430
#define, 430
#elif, 428
#else, 428
#endif, 428
#error, see preprocessing directives, error

#if, 428, 464
#define, 428
#include, 429, 449
#line, see preprocessing directives, line control
#pragma, see preprocessing directives, pragma
#undef, 433, 461
%, see remainder operator
&., see operator, address-of, see bitwise AND operator
|., see declarator, reference
&&, see logical AND operator
^, see bitwise exclusive OR operator
__DATE__, 436
__FILE__, 436
__LINE__, 436
__STDC__, 436
   implementation-defined, 436
   __STDCPP STRICT POINTER SAFETY__, 437
      implementation-defined, 437
   __STDCPP Threads__, 437
      implementation-defined, 437
   __STDC HOSTED__, 436
      implementation-defined, 436
   __STDC_ISO 10646__, 436
      implementation-defined, 436
   __STDC_MB_MIGHT_NEQ WC__, 436
      implementation-defined, 436
   __STDC_VERSION__, 436
      implementation-defined, 436
   __TIME__, 436
   __VA_ARGS__, 430
   __cplusplus, 436
~., see destructor
\, see backslash
{
      block statement, 137
         class declaration, 226
            class definition, 226
               enum declaration, 166
                  initializer list, 214
         see character, underscore
|., see bitwise inclusive OR operator
||, see logical OR operator
~, see operator, one’s complement
0, see also zero, null
null character, 30
string terminator, 30

abort, 65, 143
abstract-declarator, 192, 1224
abstract-pack-declarator, 192, 1224
access control, 256–266
  base class, 259
  base class member, 243
  class member, 105
  default, 256
  default argument, 257
  friend function, 261
  member name, 256
  multiple access, 266
  nested class, 266
  overloading and, 303
private, 256
protected, 256, 264
public, 256
union default member, 227
using-declaration and, 178
virtual function, 265
access-specifier, 243, 1226
access control
  anonymous union, 239
  member function and, 267
  overloading resolution and, 247
access specifier, 257, 259
addition operator, 124
additive-expression, 125, 1217
address, 77, 127
address of member function
  unspecified, 465
aggregate, 214
aggregate initialization, 214
algorithm
  stable, 442, 465
alias
  namespace, 173
alias template, 363
alias-declaration, 147, 1220
alignment
  extended, 80
  fundamental, 80
alignment-specifier, 185, 1222
alignment requirement
  implementation-defined, 80
allocation
  alignment storage, 118
  implementation defined bit-field, 240
  unspecified, 232
allocation functions, 66
allowing all exceptions, see exception specification,
  allowing all exceptions
allowing an exception, see exception specification,
  allowing an exception
alternative token, see token, alternative
ambiguity
  base class member, 246
  class conversion, 249
  declaration type, 149
  declaration versus cast, 193
  declaration versus expression, 145
  function declaration, 212
  member access, 246
  overloading function, 303
  parentheses and, 116
Amendment 1, 461
and-expression, 128, 1218
anonymous union, 239
appertain, 186
argv, 62
argument, 2, 463–465, 500
  access checking and default, 257
  binding of default, 205
  evaluation of default, 205, 206
  example of default, 204, 205
  function call expression, 2
  function-like macro, 2
  overloaded operator and default, 325
  reference, 104
  scope of default, 206
  template, 339
  template instantiation, 2
  throw expression, 2
  type checking of default, 205
arguments
  implementation-defined order of evaluation of
    function, 206
argument and name hiding
  default, 206
argument and virtual function
  default, 206
argument list
  empty, 201
  variable, 201
argument passing, 104
  reference and, 218
argument substitution, see macro, argument sub-
  stitution
argument type

Cross references 1285
unknown, 201
argv, 62
arithmetic
    pointer, 125
    unsigned, 75
array, 201
    bound, 199
    const, 78
    delete, 121
    multidimensional, 200
    new, 117
    overloading and pointer versus, 301
    sizeof, 115
    storage of, 200
array
    as aggregate, 777
    contiguous storage, 776
    initialization, 777, 778
    tuple interface to, 779
    zero sized, 779
array size
    default, 199
arrow operator, see operator, class member access as-if rule, 8
asm
    implementation-defined, 182
asm-definition, 182, 1222
assembler, 182
<assert.h>, 450
assignment
    and lvalue, 131
    conversion by, 131
    copy, see assignment operator, copy
    move, see assignment operator, move, 441
    reference, 218
assignment operator
    copy, 267, 291–294
    hidden, 293
    implicitly declared, 291
    implicitly defined, 293
    inaccessible, 294
    trivial, 293
    virtual bases and, 294
    move, 267, 291–294
    hidden, 293
    implicitly declared, 292
    implicitly defined, 293
    inaccessible, 294
    trivial, 293
    overloaded, 326
assignment-expression, 131, 1218
assignment-operator, 131, 1218
associative containers
    exception safety, 764
    requirements, 764
    unordered, see unordered associative containers
asynchronous provider, 1196
asynchronous return object, 1196
atexit, 65
atomic operations, see operation, atomic attribute, 185–189
    alignment, 186
    carries dependency, 188
    deprecated, 189
    noreturn, 188
    syntax and semantics, 185
attribute, 185, 1222
attribute-argument-clause, 186, 1223
attribute-declaration, 147, 1220
attribute-list, 185, 1222
attribute-namespce, 186, 1223
attribute-scoped-token, 186, 1223
attribute-specifier, 185, 1222
attribute-specifier-seq, 185, 1222
attribute-token, 185, 1222
automatic storage duration, 66
awk, 1106
backslash character, 27
bad_alloc, 119
bad_cast, 107
bad_exception, 425
bad_typeid, 108
bad_typeid::what
    implementation-defined, 489
balanced-token, 186, 1223
balanced-token-seq, 186, 1223
base class
    dependent, 373
    overloading and, 302
    base class subobject, 7
base-clause, 243, 1225
base-specifier, 243, 1226
base-specifier-list, 243, 1225
base-type-specifier, 243, 1226
BaseCharacteristic, 603
base class, 243, 244
    direct, 243
    indirect, 243
    private, 259
    protected, 259

Cross references
public, 259
base class virtual, see virtual base class
 basic_ios::failure argument
   implementation-defined, 1028
begin
 unordered associative containers, 772
behavior
 conditionally-supported, 2, 5
default, 440, 444
implementation-defined, 3, 8
locale-specific, 3
observable, 8, 9
on receipt of signal, 8
required, 441, 444
undefined, 4, 5, 8
unspecified, 4, 8
Ben, 303
Bernoulli distributions, 962–965
bernoulli_distribution
 discrete probability function, 962
binary fold, 101
binary function, 586
binary left fold, 102
binary operator
 overloaded, 326
binary right fold, 102
binary-digit, 24, 1212
binary-literal, 24, 1212
BinaryTypeTrait, 603
binary operator
 interpretation of, 326
bind directly, 220
binding
 reference, 218
binomial_distribution
 discrete probability function, 963
bit-field, 240
 address of, 240
alignment of, 240
implementation-defined sign of, 1239
implementation defined alignment of, 240
type of, 240
unnamed, 240
zero width of, 240
bitmask
 empty, 446
block, 439
 initialization in, 144
block scope, 41
block statement, see statement, compound
block-declaration, 147, 1219
block structure, 144
body
 function, 207
Boolean, 240
Boolean literal, 31
boolean literal, see literal, boolean
boolean-literal, 31, 1214
Boolean type, 76
bound arguments, 596
bound, of array, 199
brace-or-equal-initializer, 210, 1224
braced-init-list, 210, 1224
bucket
 unordered associative containers, 772
bucket_count
 unordered associative containers, 772
bucket_size
 unordered associative containers, 772
buckets, 765
byte, 6, 115
C
 linkage to, 183
 standard, 1
 standard library, 1
 Unicode TR, 1
c-char, 26, 1213
c-char-sequence, 26, 1213
call
 operator function, 325
 pseudo destructor, 105
call signature, 585
call wrapper, 585
 forwarding, 585
 simple, 585
type, 585
Callable, 599
callable object, 585, 599
callable type, 585
capture, 94, 1215
capture-default, 94, 1215
capture-list, 94, 1215
captured, 98
 by copy, 99
 by reference, 99
carries a dependency, 12
carry
 subtract_with_carry_engine, 951
<cassert>, 450
cast
 base class, 110
const, 112, 122
derived class, 110
dynamic, 107, 488
  construction and, 288
destruction and, 288
integer to pointer, 111
lvalue, 109, 111
pointer-to-function, 111
pointer-to-member, 110, 112
pointer to integer, 111
reference, 109, 112
reinterpret, 111, 122
  integer to pointer, 111
  lvalue, 111
  pointer to integer, 111
  pointer-to-function, 111
  pointer-to-member, 112
  reference, 112
static, 109, 122
  lvalue, 109
  reference, 109
  undefined pointer-to-function, 111
cast-expression, 122, 1217
casting, 105
casting away constness, 113
catch, 414
cauchy_distribution
  probability density function, 972
cbegin
  unordered associative containers, 772
cend
  unordered associative containers, 772
cerrno, 461
char
  implementation-defined sign of, 75
char-like object, 648
char-like type, 648
char16_t, 26
char16_t character, 26
char32_t, 26
char32_t character, 26
char_class_type
  regular expression traits, 1096
character, 439
  decimal-point, 446
  multibyte, 3
  signed, 75
  source file, 17
  underscore, 23
  in identifier, 23
character literal, see literal, character
character set, 18–19
  basic execution, 6
  basic source, 17, 18
character string literal, 431
character_literal, 26, 1213
character string, 29
checking
  point of error, 366
  syntax, 366
chi_squared_distribution
  probability density function, 972
class, 77, 226–242
  abstract, 254
  base, 461, 466
  cast to incomplete, 123
  constructor and abstract, 255
  definition, 36
  derived, 466
  linkage of, 59
  linkage specification, 184
  member function, see member function, class
  pointer to abstract, 254
  polymorphic, 249
  scope of enumerator, 168
  standard-layout, 227
  trivial, 227
  unnamed, 154
class-head, 226, 1225
class-head-name, 226, 1225
class-key, 226, 1225
class-name, 226, 1225
class-or-decltype, 243, 1226
class-specifier, 226, 1225
class-virt-specifier, 226, 1225
class local, see local class
class name
  elaborated, 162, 229, 230
  point of declaration, 230
  scope of, 229
typedef, 153, 154, 230
class nested, see nested class
class object
  assignment to, 131
  member, 232
  sizeof, 115
class object copy, see copy constructor
class object initialization, see constructor
clear
  unordered associative containers, 771
clocale, 446
closure object, 94
Cross references 1288
closure type, 94
collating element, 1095
comment, 19, 21
/* */, 21
//, 21
common initial sequence, 233
comparison
    pointer, 127
    pointer to function, 127
    undefined pointer, 125
compatible, see exception specification, compatible
compilation
    separate, 17
compiler control line, see preprocessing directives
complete object, 7
complete object of, 7
completely defined, 231
component, 440
compound-statement, 137, 1218
concatenation
    macro argument, see ##
    string, 30
condition, 138, 1219
conditions
    rules for, 138
conditional-expression
    throw-expression in, 129
conditional-expression, 129, 1218
conditionally-supported behavior
    see behavior, conditionally-supported, 1
conflict, 12
conformance requirements, 5, 8
    class templates, 5
    classes, 5
    general, 5
    library, 5
    method of description, 5
consistency
    linkage, 150
    linkage specification, 184
type declaration, 62
const, 77
    cast away, 113
    constructor and, 236, 268
destructor and, 236, 275
    linkage of, 59
    overloading and, 302
const_cast, see cast, const
constexpr function, 154
    construction, 286–288
dynamic cast and, 288
    member access, 286
    move, 440
    pointer to member or base, 286
    typeid operator, 287
    virtual function call, 287
constructor, 267
    address of, 269
    array of class objects and, 280
    converting, 273
    copy, 267, 270, 288–291, 447
    elision, 294
    implicitly declared, 289
    implicitly defined, 291
    inaccessible, 294
    trivial, 290
default, 267, 268
exception handling, see exception handling,
    constructors and destructors
    explicit call, 269
    implicitly called, 269
    implicitly defined, 269
    inheritance of, 268
    inheriting, 296–299
    move, 267, 288–291
    elision, 294
    implicitly declared, 290
    implicitly defined, 291
    inaccessible, 294
    trivial, 290
    non-trivial, 268
    random number distribution requirement, 943
    random number engine requirement, 940
union, 238
    unspecified argument to, 119
constructor, conversion by, see conversion, user-defined
cionario
    default, see default constructor
const-object
    undefined change to, 158
counter
        contiguous, 749
context
  non-deduced, 405
contextually converted to bool, see conversion, contextual
contiguous container, 749
contiguous iterators, 853
continue
  and handler, 414
  and try block, 414
control line, see preprocessing directives
control-line, 426, 1228
conventions, 445
  lexical, 17–33
conversion
  argument, 201
  array-to-pointer, 83
bool, 85
boolean, 87
class, 272
contextual, 82
contextual to bool, 82
deduced return type of user-defined, 275
derived-to-base, 315
floating point, 85
floating to integral, 86
function-to-pointer, 83
implementation defined pointer integer, 111
implicit, 82, 272
implicit user-defined, 272
inheritance of user-defined, 274
integer rank, 87
integral, 85
integral to floating, 86
lvalue-to-rvalue, 83, 1235
narrowing, 224
overload resolution and pointer, 324
overload resolution and, 312
pointer, 86
pointer to member, 86
  void*, 87
qualification, 84
return type, 143
standard, 82–87
static user-defined, 275
to signed, 85
to unsigned, 85
type of, 273
user-defined, 272, 273
usual arithmetic, 89
virtual user-defined, 275
conversion operator, see conversion, user defined
conversion rank, 316
conversion-declarator, 273, 1226
conversion-function-id, 273, 1226
conversion-type-id, 273, 1226
conversion explicit type, see casting
conversion function, see conversion, user-defined copy
  class object, see constructor, copy; assignment, copy
copy constructor
  random number engine requirement, 940
copy elision, see constructor, copy, elision; constructor, move, elision
copy-initialization, 213
CopyInsertable into X, 750
count
  unordered associative containers, 772
<cstdio>, 201
<cstdlib>, 115, 125
<cstring>, 478
<ctime>, 65, 449
<string>, 447
ctor-initializer, 281, 1226
<char>, 461
cv-decomposition, 84
cv-qualification signature, 84
cv-qualifier, 77
top-level, 78
cv-qualifier, 192, 1223
cv-qualifier-seq, 192, 1223
<char>, 461
<char>, 461
d-char, 29, 1214
d-char-sequence, 28, 1214
DAG
  multiple inheritance, 245, 246
  non-virtual base class, 246
  virtual base class, 245, 246
data race, 14
data member, see member
data race, 14
deadlock, 440
deallocation, see delete
deallocation functions, 66
decay, see conversion, array to pointer; conversion, function to pointer
DECAY_COPY, 1162
decimal-literal, 24, 1212
decl-specifier, 149, 1220
decl-specifier-seq, 149, 1220

Cross references
dynamic cast and, 288
member access, 286
pointer to member or base, 286
typeid operator, 287
virtual function call, 287
destructor, 275, 447
default, 275
exception handling, see exception handling, constructors and destructors
explicit call, 276
implicit call, 276
implicitly defined, 276
program termination and, 276
pure virtual, 276
union, 238
virtual, 276
diagnosable rules, 5
diagnostic message, see message, diagnostic
digit, 22, 1211
digit-sequence, 28, 1213
digraph, see token, alternative, 20
directed acyclic graph, see DAG
directive, preprocessing, see preprocessing directives
discard
  random number engine requirement, 940
discard_block_engine
generation algorithm, 952
state, 952
textual representation, 953
transition algorithm, 952
discarded-value expression, 90
discrete probability function
  bernoulli_distribution, 962
  binomial_distribution, 963
  discrete_distribution, 975
  geometric_distribution, 964
  negative_binomial_distribution, 965
  poisson_distribution, 965
  uniform_int_distribution, 960
discrete_distribution
  discrete probability function, 975
  weights, 975
distribution, see random number distribution
dominance
  virtual base class, 248
dot operator, see operator, class member access
dynamic binding, see virtual function
dynamic initialization, 63
dynamic type, see type, dynamic
dynamic-exception-specification, 419, 1227
dynamic_cast, see cast, dynamic
ECMA-262, 1
ECMAScript, 1106, 1139
egrep, 1106
elaborated-type-specifier, 162, 1221
elaborated type specifier, see class name, elaborated
elif-group, 426, 1228
elif-groups, 426, 1228
elision
copy, see constructor, copy, elision; constructor, move, elision
copy constructor, see constructor, copy, elision
move constructor, see constructor, move, elision
ellipsis
  conversion sequence, 104, 317
  overload resolution and, 312
else-group, 426, 1228
EmplaceConstructible into X from args, 750
empty future object, 1201
empty shared_future object, 1203
empty-declaration, 147, 1220
encoding
  multibyte, 30
  encoding-prefix, 26, 1213
end
  unordered associative containers, 772
end-of-file, 540
endif-line, 426, 1228
engine, see random number engine
engine adaptor, see random number engine adaptor
engines with predefined parameters
default_random_engine, 956
knuth_b, 956
minstd_rand, 955
minstd_rand0, 955
mt19937, 955
mt19937_64, 956
ranlux24, 956
ranlux24_base, 956
ranlux48, 956
ranlux48_base, 956
entity, 34
enum, 77
  overloading and, 301
Cross references

1292
type of, 166, 167
underlying type, 167
enum-base, 166, 1221
enum-head, 166, 1221
enum-key, 166, 1221
enum-name, 166, 1221
enum-specifier, 166, 1221
everification, 166, 167
linkage of, 59
scoped, 167
unscoped, 167
enumeration type
conversion to, 110
static_cast conversion to, 110
enumerator
definition, 36
value of, 167
everifier, 166, 1222
everifier-definition, 166, 1221
neverifier-list, 166, 1221
everification name
typedef, 154
environment
program, 62
epoch, 628
equal_range
unordered associative containers, 772
equality-expression, 127, 1217
equivence
template type, 344
type, 152, 229
equivalent-key group, 764
equivalently-valued, 459
equivalent parameter declarations, 301
overloading and, 301
Erasable from X, 751
erase
unordered associative containers, 771
escape-sequence, 26, 1213
escape character, see backslash
escape sequence
undefined, 27
Evaluation, 10
evaluation
order of argument, 104
unspecified order of, 10, 64
unspecified order of argument, 104
unspecified order of function call, 104
example
array, 199
class definition, 232
cost, 195
constant pointer, 195
constructor, 269
constructor and initialization, 280
declaration, 35, 202
declarator, 192
definition, 35
delete, 278
derived class, 243
destructor and delete, 278
eclipse, 201
enumeration, 168
explicit destructor call, 277
explicit qualification, 247
friend, 230
friend function, 261
function declaration, 202
function definition, 207
linkage consistency, 150
local class, 242
member function, 234, 261
nested type name, 242
nested class, 240
nested class definition, 241, 266
nested class forward declaration, 241
pointer to member, 198
pure virtual function, 254
scope of delete, 278
scope resolution operator, 247
static member, 237
subscripting, 199
typedef, 152
type name, 192
unnamed parameter, 207
variable parameter list, 201
virtual function, 251, 252
exception
arithmetic, 88
undefined arithmetic, 88
exception handling, 414–425
constructors and destructors, 417
exception object, 416
constructor, 416
destructor, 416
function try block, 415
goto, 414
handler, 414, 416–419, 466
array in, 417
Cross references 1293
incomplete type in, 417
match, 417–419
pointer to function in, 417
rvalue reference in, 417
memory, 416
nearest handler, 416
rethrow, 130, 131, 416
rethrowing, 416
switch, 414
terminate() called, 131, 416, 421
throwing, 130, 415, 416
try block, 414
unexpected() called, 421
exception object, see exception handling, exception object
exception specification, 419–424
allowing all exceptions, 421
allowing an exception, 421
compatible, 420
incomplete type and, 419
noexcept
custom expression and, 419
non-throwing, 419
virtual function and, 420
exception-declaration, 414, 1227
exception-specification, 419, 1227
exception::what message
implementation-defined, 490
exclusive-or-expression, 128, 1218
execution agent, 1160
exit, 62, 64, 143
explicit-instantiation, 384, 1227
explicit-specialization, 386, 1227
explicitly captured, 97
explicit type conversion, see casting
exponent-part, 28, 1213
exponential_distribution
probability density function, 966
expression, 88–136
additive operators, 124
alignof, 122
assignment and compound assignment, 131
bitwise AND, 128
bitwise exclusive OR, 128
bitwise inclusive OR, 128
cast, 105, 122–123
class member access, 105
comma, 132
conditional operator, 129
custom, 132
const cast, 112
decrement, 106, 115
delete, 120
dynamic cast, 107
equality operators, 127
fold, 101–102
function call, 103
increment, 106, 115
lambda, 94–101
left-shift-operator, 126
logical AND, 128
logical OR, 129
multiplicative operators, 124
new, 115
noexcept, 122
order of evaluation of, 9
parenthesized, 92
pointer-to-member, 123
pointer to member constant, 114
postfix, 102–113
primary, 91–101
pseudo-destructor call, 105
reference, 88
reinterpret cast, 111
relational operators, 126
right-shift-operator, 126
rvalue reference, 88
sizeof, 115
static cast, 109
throw, 130
type identification, 108
unary, 113–122
unary operator, 113
expression, 132, 1218
expression-list, 102, 1216
expression-statement, 137, 1218
extend, see namespace, extend
extended alignment, 80
extended integer type, 75
extended signed integer type, 75
extended unsigned integer type, 75
extern, 149
linkage of, 150
extern "C", 450, 461
extern "C++", 450, 461
external linkage, 59
extreme_value_distribution
probability density function, 969
file, source, see source file
final overrider, 250
find
unordered associative containers, 771
finite state machine, 1095
fisher_f_distribution
probability density function, 973
floating literal, see literal, floating
floating-literal, 27, 1213
floating-point literal, see literal, floating
floating-suffix, 28, 1213
floating point type, 76
implementation-defined, 76
fold
binary, 101
unary, 101
fold-expression, 101, 1216
fold-operator, 101, 1216
for
scope of declaration in, 141
for-init-statement, 139, 1219
for-range-declaration, 140, 1219
for-range-initializer, 140, 1219
format specifier, 1095
forwarding call wrapper, 585
forwarding reference, 400
fractional-constant, 28, 1213
free store, 278
freestanding implementation, 5
free store, see also new, delete
friend
virtual and, 252
access specifier and, 263
class access and, 261
inheritance and, 263
local class and, 264
template and, 352
friend function
access and, 261
inline, 263
linkage of, 263
member function and, 261
friend function
nested class, 241
full-expression, 9
function, see also friend function; member function; inline function; virtual function, 201
allocation, 67, 117
comparison, 440
conversion, 273
deallocation, 67, 121, 278
definition, 36
global, 461, 464, 465
handler, 440
linkage specification overloaded, 184
modifier, 440
observer, 441
operator, 325
overload resolution and, 304
plain old, 495
pointer to member, 124
replacement, 441
reserved, 441
viable, 304
virtual member, 461, 465
function object, 582
binders, 594–597
mem_fn, 597
reference_wrapper, 586
type, 582
wrapper, 597–602
function pointer type, 77
function try block, see exception handling, function try block
function, overloaded, see overloading
function, virtual, see virtual function
function-definition, 207, 1224
function-like macro, see macro, function-like
function-specifier, 151, 1220
function-try-block, 414, 1227
functions
candidate, 378
function argument, see argument
function call, 104
recursive, 104
undefined, 111
function call operator
overloaded, 326
function parameter, see parameter
function prototype, 41
function return, see return
function return type, see return type
fundamental alignment, 80
fundamental type
destructor and, 277
fundamental type conversion, see conversion, user-defined
future
shared state, 1196
gamma_distribution
probability density function, 967
generate
seed sequence requirement, 938
generated destructor, see destructor, default
Cross references
cross references

C, 461, 464, 1253
C library, 450
C++ library, 448
name, 21
header-name, 21, 1211
hex-quad, 18, 1210
hexadecimal-digit, 24, 1212
hexadecimal-escape-sequence, 26, 1213
hexadecimal-literal, 24, 1212

hash
instantiation restrictions, 602
hash code, 765
hash function, 764
hash tables, see unordered associative containers
hash_function
unordered associative containers, 769
hasher
unordered associative containers, 765
header
discard_block_engine, 952
independent_bits_engine, 953
linear_congruential_engine, 948
mersenne_twister_engine, 949
shuffle_order_engine, 954
subtract_with_carry_engine, 951
generic lambda
definition of, 163
geometric_distribution
discrete probability function, 964
global, 42
global namespace, 42
global namespace scope, 42
global scope, 42
gvalue, 79
goto
and handler, 414
and try block, 414
initialization and, 144
grammar
regular expression, 1139
grep, 1106
group, 426, 1228
group-part, 426, 1228
h-char, 21, 1211
h-char-sequence, 21, 1211
handler, see exception handling, handler
handler, 414, 1227
handler-seq, 414, 1227
happens before, 13
hash
instantiation restrictions, 602
hash code, 765
hash function, 764
hash tables, see unordered associative containers
hash_function
unordered associative containers, 769
hasher
unordered associative containers, 765
header
C, 461, 464, 1253
C library, 450
C++ library, 448
name, 21
header-name, 21, 1211
hex-quad, 18, 1210
hexadecimal-digit, 24, 1212
hexadecimal-escape-sequence, 26, 1213
hexadecimal-literal, 24, 1212

Cross references
generation algorithm, 953
state, 953
textual representation, 954
transition algorithm, 953
indeterminately sequenced, 10
indeterminate value, 212
indirection, 113
inheritance, 243
init-capture, 94, 1216
init-declarator, 191, 1223
init-declarator-list, 191, 1223
initialization, 62, 210–225
aggregate, 214
array, 214
array of class objects, 217, 280
automatic, 144
automatic object, 210
base class, 281, 282
character array, 217
character array, 217
class member, 212
class object, see also constructor, 214, 279–285
const, 158, 214
cost member, 282
constant, 63
constructor and, 279, 280
copy, 213
default, 211
default constructor and, 279
definition and, 148
direct, 213
dynamic, 62
explicit, 280
jump past, 144
list-initialization, 220–225
local static, 144
member, 281
member function call during, 285
member object, 282
non-vacuous, 69
order of, 62, 244
order of base class, 284
order of member, 284
order of virtual base class, 283
overloaded assignment and, 280
parameter, 103
reference, 197, 218
reference member, 282
run-time, 62
static and thread, 62
static member, 237
static object, 62
static object, 210, 211
union, 217, 239
virtual base class, 291
initializer
base class, 207
member, 207
pack expansion, 285
scope of member, 284
temporary and declarator, 271
initializer, 210, 1224
initializer-clause, 210, 1224
initializer-list, 210, 1224
initializer-list constructor
seed sequence requirement, 938
<initializer_list>, 494
injected-class-name, 226
inline
linkage of, 59
inline function, 151
insert
unordered associative containers, 770, 771
instantiation
dependent member of the current, 373
explicit, 384
member of the current, 373
point of, 377
template implicit, 380
instantiation units, 18
integer literal, see literal, integer
integer representation, 69
integer-literal, 24, 1212
integer-suffix, 24, 1212
integer type, 76
integral type, 76
sizeof, 75
inter-thread happens before, 13
internal linkage, 59
interval boundaries
piecewise_constant_distribution, 977
piecewise_linear_distribution, 978
invocation
macro, 431
iscctype
regular expression traits, 1097
iteration-statement, 139, 143, 1219
Jessie, 273
jump-statement, 142, 1219
key_eq
  unordered associative containers, 769
key_equal
  unordered associative containers, 766
key_type
  unordered associative containers, 765
keyword, 23
label, 143
  case, 137, 139
  default, 137, 139
  scope of, 42, 137
labeled-statement, 137, 1218
lambda-capture, 94, 1215
lambda-declarator, 94, 1216
lambda-expression, 94, 1215
lambda-introducer, 94, 159, 1215
lattice, see DAG, subobject layout
  bit-field, 240
  class object, 232, 244
layout-compatible, 75, 168
layout-compatible type, 75
left shift
  undefined, 126
left shift operator, 126
lexical conventions, see conventions, lexical library
  C standard, 439, 446, 448, 450, 1249, 1250, 1253
  C++ standard, 438, 461, 463, 466
library clauses, 5
lifetime, 69
limits
  implementation, 3
<limits>, 469
line splicing, 17
linear_congruential_engine
  generation algorithm, 948
  modulus, 949
  state, 948
  textual representation, 949
  transition algorithm, 948
linkage, 34, 59–62
  const and, 59
  external, 59, 450, 461
  implementation-defined object, 185
  inline and, 59
  internal, 59
  no, 59, 60
  static and, 59

Cross references

linkage specification, see specification, linkage
linkage-specification, 182, 1222
literal, 24–33, 91
  base of integer, 25
  binary, 25
  boolean, 31
  char16_t, 26
  char32_t, 26
  character, 26
  constant, 24
  decimal, 25
  double, 28
  float, 28
  floating, 27, 28
  hexadecimal, 25
  char, 27
  integer, 24, 25
  long, 25
  long double, 28
  multicharacter, 26
  implementation-defined value of, 26
  narrow-character, 26
  octal, 25
  pointer, 31
  string, 28, 29
  char16_t, 29
  char32_t, 29, 30
  narrow, 29
  type of, 29
  undefined change to, 30
  wide, 29, 30
  type of character, 26
  type of floating point, 28
  type of integer, 25
  unsigned, 25
  user defined, 31
literal, 24, 1212
literal type, 74
literal-operator-id, 328, 1226
load_factor
  unordered associative containers, 773
local lambda expression, 97
local variable, 41
local_iterator, 766
  unordered associative containers, 766
locale, 1095, 1096, 1098, 1106
locale-specific behavior, see behavior, locale-specific
local class
  friend, 264
  member function in, 234
  scope of, 242
local scope, see block scope
local variable
destruction of, 142, 144
lock-free executions, 11
logical-and-expression, 128, 1218
logical-or-expression, 129, 1218
lognormal_distribution
probability density function, 971
long
typedef and, 149
long-long-suffix, 25, 1213
long-suffix, 25, 1213
lookup
argument-dependent, 49
class member, 58
class member, 52
elaborated type specifier, 57
member name, 246
name, 34, 45–59
namespace aliases and, 59
namespace member, 53
qualified name, 51–56
template name, 364
unqualified name, 45
using-directives and, 59
lookup_classname
regular expression traits, 1141
lookup_classname
regular expression traits, 1097
lookup_collatename
regular expression traits, 1097
low-order bit, 6
lowercase, 446
lparen, 427, 1228
lvalue, 79, 1235
lvalue reference, 76, 196
macro
argument substitution, 431
function-like, 430
arguments, 431
masking, 464
name, 430
object-like, 430
pragma operator, 437
predefined, 436
replacement, 430–435
replacement list, 430
rescanning and replacement, 432
scope of definition, 432
main(), 62
implementation-defined linkage of, 62
implementation-defined parameters to, 62
parameters to, 62
return from, 62, 64
match_results
as sequence, 1123
matched, 1095
max
random number distribution requirement, 944
uniform random number generator requirement, 939
max_bucket_count
unordered associative containers, 772
max_load_factor
unordered associative containers, 773
mean
normal_distribution, 970
poisson_distribution, 966
mem-initializer, 281, 1226
mem-initializer-id, 281, 1226
mem-initializer-list, 281, 1226
member
class static, 66
enumerator, 169
static, 236
template and static, 347
member access operator
overloaded, 327
member function
call undefined, 234
class, 233
const, 235
constructor and, 269
destructor and, 276
friend, 263
inline, 233
local class, 242
nested class, 266
nonstatic, 234
overload resolution and, 304
static, 236, 237
this, 235
union, 238
volatile, 235
member names, 42
member subobject, 7
member-declaration, 231, 1225
member-declarator, 231, 1225
member-declarator-list, 231, 1225
member-specification, 230, 1225
members, 42
Cross references

N4296

© ISO/IEC

1299
member data
  static, 237
member pointer to, see pointer to member
memory location, 6
memory model, 6–7
memory management, see also new, delete

merseene_twister_engine
  generation algorithm, 949
  state, 949
  textual representation, 950
  transition algorithm, 949
message
  diagnostic, 2, 5
min
  random number distribution requirement, 944
  uniform random number generator require-
  ment, 939
modification order, 12
more specialized
  function template, 404
most derived class, 7
most derived object, 7
  bit-field, 7
  zero size subobject, 7
move
  class object, see constructor, move; assign-
  ment, move
MoveInsertable into X, 750
multi-pass guarantee, 856
multibyte character, see character, multibyte
multicharacter literal, see literal, multicharacter
multiple threads, see threads, multiple
multiple inheritance, 243, 244
  virtual and, 252
multiplicative-expression, 124, 1217
mutable, 149
mutable iterator, 852
mutex types, 1168

name, 22, 34, 92
  address of cv-qualified, 114
  dependent, 370, 377
  elaborated
    enum, 162
  global, 42
  length of, 22
  macro, see macro, name
  point of declaration, 40
  predefined macro, see macro, predefined
  qualified, 51
  reserved, 460
  scope of, 39
  unqualified, 45
name hiding
  function, 303
  overloading versus, 303
  using-declaration and, 178
named-namespace-definition, 169, 1222
namespace, 448, 1253
  alias, 173
  definition, 169
  extend, 170
  global, 23
  member definition, 172
  unnamed, 171
namespace-alias, 173, 1222
namespace-alias-definition, 173, 1222
namespace-body, 170, 1222
namespace-definition, 169, 1222
namespaces, 169–182
name class, see class name
name hiding, 40, 45, 93, 144
  class definition, 229
  user-defined conversion and, 272
name space
  label, 137
narrowing conversion, 224
NDEBUG, 450
negative_binomial_distribution
  discrete probability function, 965
nested-name-specifier, 93, 1215
nested-namespace-definition, 169, 1222
nested class
  local class, 242
  scope of, 240
new, 66, 115, 117
  array of class objects and, 119
  constructor and, 119
  default constructor and, 119
  exception and, 119
  initialization and, 119
operator, 461, 462
  scoping and, 116
  storage allocation, 116
  type of, 278
  unspecified constructor and, 119
  unspecified order of evaluation, 119
new-declarator, 116, 1217
new-expression, 116, 1217
new-initializer, 116, 1217

Cross references
new-line, 427, 1229
new-placement, 116, 1217
new-type-id, 116, 1217
new_handler, 67
no linkage, 59
noexcept-expression, 122, 1217
noexcept-specification, 419, 1228
non-directive, 427, 1228
non-throwing exception specification, 419
nondigit, 22, 1211
nonzero-digit, 24, 1212
noptr-abstract-declarator, 192, 1224	noptr-abstract-pack-declarator, 192, 1224	noptr-declarator, 191, 1223
noptr-new-declarator, 116, 1217
normal distributions, 970–975
normal_distribution
mean, 970
probability density function, 970
standard deviation, 970
normative references, see references, normative notation
syntax, 6
NTBS, 447, 1083, 1261, 1262
  static, 447
NTCTS, 441
NTMBS, 447
  static, 447
number
  hex, 27
  octal, 27
numeric_limits, 469
  specializations for arithmetic types, 76
object, see also object model, 7, 34
  byte copying and, 73
  complete, 7
  definition, 36
  delete, 121
destructor static, 64
destructor and placement of, 277
linkage specification, 185
local static, 66
undefined deleted, 68
unnamed, 269
object expression, 105
object model, 7–8
object pointer type, 77
object representation, 73
object type, 7, 74
  incompletely-defined, 73
object, exception, see exception handling, exception object
object-like macro, see macro, object-like
object class, see also class object
object lifetime, 69–73
object temporary, see temporary
object type, 74
observable behavior, see behavior, observable
octal-digit, 24, 1212
octal-escape-sequence, 26, 1213
octal-literal, 24, 1212
odr-used, 36
one-definition rule, 36–39
opaque-enum-declaration, 166, 1221
operation
  atomic, 11–15
operator, 23–24, 325
  **=, 131
  +=, 115, 131
  -=, 131
  /=, 131
  <<=, 131
  >>=, 131
  %*, 131
  &*, 131
  *=, 131
  |=, 131
  address-of, 113
  assignment, 131, 447
  bitwise, 128
  bitwise AND, 128
  bitwise exclusive OR, 128
  bitwise inclusive OR, 128
cast, 113, 192
class member access, 105
comma, 132
conditional expression, 129
copy assignment, see assignment, copy
decrement, 106, 113, 115
division, 124
equality, 127
function call, 103, 325
greater than, 126
greater than or equal to, 126
increment, 106, 113, 115
indirection, 113
inequality, 127
less than, 126
less than or equal to, 126
logical AND, 128

Cross references 1301
logical negation, 113, 114
logical OR, 129
move assignment, see assignment, move
multiplication, 124
multiplicative, 124
one’s complement, 113, 114
overloaded, 88, 325
pointer to member, 123
pragma, see macro, pragma operator
precedence of, 9
relational, 126
remainder, 124
scope resolution, 93, 117, 233, 243, 253
side effects and comma, 132
side effects and logical AND, 128
side effects and logical OR, 129
sizeof, 113, 115
subscripting, 102, 325
unary, 113
unary minus, 113, 114
unary plus, 113, 114
operator, 325, 1226
operator delete, see also delete, 117, 121, 278
operator new, see also new, 117
operator overloading, see overloading, operator
operator !=
  random number distribution requirement, 944
  random number engine requirement, 940
operator()
  random number distribution requirement, 943, 944
  random number engine requirement, 940
  uniform random number generator requirement, 939
operator-function-id, 325, 1226
operator<<
  random number distribution requirement, 944
  random number engine requirement, 941
operator==
  random number distribution requirement, 944
  random number engine requirement, 940
operator>>
  random number distribution requirement, 944
  random number engine requirement, 941
operator , see delete
operator left shift, see left shift operator
operator right shift, see right shift operator
operator use
  scope resolution, 237
optimization of temporary, see elimination of temporary

order of evaluation in expression, see expression, order of evaluation of
ordered, 63
ordering
  function template partial, 361
order of execution
  base class constructor, 269
  base class destructor, 276
  constructor and static objects, 280
  constructor and array, 279
destructor, 276
destructor and array, 276
member constructor, 269
member destructor, 276
ordinary character literal, 26
over-aligned type, 80
overflow, 88
  undefined, 88
overloaded function, see overloading
overloaded operator, see overloading, operator
overloaded function
  address of, 323
overloaded function
  address of, 114
overloaded operator
  inheritance of, 325
overloading, 201, 229, 300–322, 359
  access control and, 303
  address of overload function, 323
  argument lists, 304–311
  assignment operator, 326
  binary operator, 326
  built-in operators and, 329
  candidate functions, 304–311
  declaration matching, 302
  declarations, 300
  example of, 300
  function call operator, 326
  member access operator, 327
  operator, 325–329
  prohibited, 300
resolution, 303–323
  best viable function, 312–325
  contexts, 304
  function call syntax, 306–307
  function template, 411
  implicit conversions and, 314–323
  initialization, 310, 311
  operators, 307
  scoping ambiguity, 247
template, 361

Cross references

1302
template name, 364
viable functions, 312–325
subscripting operator, 327
unary operator, 326
user-defined literal, 328
using directive and, 181
using-declaration and, 178
overloads
  floating point, 935
overrider
  final, 250
own, 556
pair
tuple interface to, 519
param
  random number distribution requirement, 943
  seed sequence requirement, 938
param_type
  random number distribution requirement, 943
parameter
  catch clause, 3
  function, 3
  function-like macro, 3
  reference, 196
  scope of, 41
  template, 3, 35
  void, 201
parameter declaration, 35
parameter-declaration, 201, 1224
parameter-declaration-clause, 200, 1224
parameter-declaration-list, 200, 1224
parameterized type, see template parameters
  macro, 430
parameters-and-qualifiers, 191, 1223
parameter list
  variable, 104, 201
period, 446
phases of translation, see translation, phases
piecewise construction, 520
piecewise_constant_distribution
  interval boundaries, 977
  probability density function, 976
  weights, 977
piecewise_linear_distribution
  interval boundaries, 978
  probability density function, 978
  weights at boundaries, 978
placement new-expression, 119
placement syntax

new, 119
pm-expression, 123, 1217
POD class, 228
POD struct, 228
POD union, 228
POF, 495
point of declaration, 40
pointer, see also void*
  safely-derived, 68–69
  to traceable object, 467
  to traceable object, 68
  zero, 86
pointer literal, see literal, pointer
pointer, integer representation of safely-derived, 69
pointer-literal, 31, 1214
pointer to member, 77, 123
Poisson distributions, 965–970
poisson_distribution
  discrete probability function, 965
  mean, 966
POSIX, 1
  extended regular expressions, 1106
  regular expressions, 1106
postfix-expression, 102, 1216
postfix ++ and --
  overloading, 327
postfix ++, 106
postfix --, 106
potential exception, 421
potential results, 36
potential scope, 39
potentially evaluated, 36
potentially concurrent, 14
pp-number, 22, 1211
pp-tokens, 427, 1229
precedence of operator, see operator, precedence of
prefix
  L, 26, 30
prefix ++ and --
  overloading, 327
prefix ++, 115
prefix --, 115
preprocessing directive, 426
  conditional inclusion, 427
preprocessing directives, 426–437
  error, 435
  header inclusion, 429
  line control, 435
  macro replacement, see macro, replacement

Cross references
null, 436
pragma, 435
source-file inclusion, 429
preprocessing-file, 426, 1228
preprocessing-op-or-punc, 23, 1212
preprocessing-token, 19, 1210
primary equivalence class, 1096
primary-expression, 91, 1215
private, see access control, private
probability density function
cauchy_distribution, 972
chi_squared_distribution, 972
exponential_distribution, 966
extreme_value_distribution, 969
fisher_f_distribution, 973
gamma_distribution, 971
lognormal_distribution, 971
normal_distribution, 970
piecewise_constant_distribution, 976
piecewise_linear_distribution, 978
student_t_distribution, 974
uniform_real_distribution, 961
weibull_distribution, 968
program, 59
ill-formed, 2
start, 62–64
termination, 64–65
well-formed, 4, 8
program execution, 8–11
abstract machine, 8
as-if rule, see as-if rule
promotion
bool to int, 85
floating point, 85
integral, 84
protected, see access control, protected
protection, see access control, 466
prvalue, 79
pseudo-destructor-name, 105
pseudo-destructor-name, 102, 1216
ptr-abstract-declarator, 192, 1224
ptr-declarator, 191, 1223
ptr-operator, 192, 1223
ptrdiff_t, 125
implementation defined type of, 125
public, see access control, public
punctuator, 23–24
pure-specifier, 231, 1225
q-char, 21, 1211
q-char-sequence, 21, 1211
qualification
explicit, 51
qualified-id, 93, 1215
qualified-name-specifier, 173, 1222
r-char, 28, 1214
r-char-sequence, 28, 1214
random number distribution
bernoulli_distribution, 962
binomial_distribution, 963
chi_squared_distribution, 972
discrete_distribution, 966
exponential_distribution, 966
extreme_value_distribution, 969
fisher_f_distribution, 973
gamma_distribution, 967
geometric_distribution, 964
lognormal_distribution, 971
negate_binomial_distribution, 965
normal_distribution, 970
piecewise_constant_distribution, 976
piecewise_linear_distribution, 978
poisson_distribution, 965
requirements, 942–945
student_t_distribution, 974
uniform_int_distribution, 960
uniform_real_distribution, 961
random number distributions
Bernoulli, 962–965
normal, 970–975
Poisson, 965–970
sampling, 975–980
uniform, 960–962
random number engine
linear_congruential_engine, 948
mersenne_twister_engine, 949
requirements, 939–941
subtract_with_carry_engine, 951
with predefined parameters, 955–956
random number engine adaptor
discard_block_engine, 952
independent_bits_engine, 953
shuffle_order_engine, 954
with predefined parameters, 955–956
random number generation, 936–980
distributions, 960–980
engines, 947–955
predefined engines and adaptors, 955–956
requirements, 937–945
synopsis, 945–947
utilities, 958–960

Cross references
random number generator, see uniform random number generator

random_device
implementation leeway, 957
raw string literal, 29
raw-string, 28, 1214
reaching scope, 97
ready, 1123, 1197
redefinition
typedef, 152
ref-qualifier, 192, 1223
reference, 76
assignment to, 131
call by, 104
forwarding, 400
lvalue, 76
null, 197
rvalue, 76
sizeof, 115
reference collapsing, 197
reference-compatible, 218
reference-related, 218
references
normative, 1
regex_iterator
end-of-sequence, 1134
regex_token_iterator
end-of-sequence, 1136
regex_traits
specializations, 1109
region
declarative, 34, 39
register, 149
regular expression, 1095–1141
grammar, 1139
matched, 1095
requirements, 1096
regular expression traits, 1139
char_class_type, 1096
isctype, 1097
lookup_classname, 1141
lookup_classname, 1097
lookup_collatename, 1097
requirements, 1096, 1109
transform, 1141
transform, 1097
transform_primary, 1141
transform_primary, 1097
translate, 1141
translate, 1097
translate_nocase, 1141

translate_nocase, 1097
rehash
unordered associative containers, 773
reinterpret_cast, see cast, reinterpret
relational-expression, 126, 1217
relaxed pointer safety, 69
release sequence, 12
remainder operator, see remainder operator
replacement
macro, see macro, replacement
replacement-list, 427, 1229
representation
object, 73
value, 73
requirements, 443
Allocator, 454
container, 745, 765, 777, 778, 1123
not required for unordered associated containers, 764
CopyAssignable, 451
CopyConstructible, 451
DefaultConstructible, 451
Destructible, 451
EqualityComparable, 451
Hash, 454
iterator, 852
LessThanComparable, 451
MoveAssignable, 451
MoveConstructible, 451
NullablePointer, 454
numeric type, 924
random number distribution, 942–945
random number engine, 939–941
regular expression traits, 1096, 1109
seed sequence, 937–938
sequence, 1123
uniform random number generator, 939
unordered associative container, 765
reraise, see exception handling, rethrow
rescanning and replacement, see macro, rescanning and replacement
reserved identifier, 23
reset, 556
reset
random number distribution requirement, 943
resolution, see overloading, resolution
restriction, 463, 464, 466
address of bit-field, 240
anonymous union, 239
bit-field, 240
constructor, 268, 269

Cross references 1305
Cross references 1306
visible, 14
sign, 28, 1213
signal, 8
signature, 3, 4
signed
typedef and, 149
signed integer type, 75
similar types, 84
simple call wrapper, 585
simple-capture, 94, 1215
simple-declaration, 147, 1220
simple-escape-sequence, 26, 1213
simple-template-id, 337, 1227
simple-type-specifier, 159, 1221
size
seed sequence requirement, 938
size_t, 115
smart pointers, 567–582
source file, 17, 449, 461
source file character, see character, source file
space
white, 19
specialization
class template, 339
class template partial, 355
template, 379
template explicit, 386
special member function, see constructor, destructor, inline function, user-defined conversion, virtual function
specification
linkage, 182–185
extern, 182
implementation-defined, 183
nesting, 183
template argument, 392
specifications
C standard library exception, 466
C++, 467
implementation-defined exception, 467
specifier, 149–164
friend, 466
constexpr, 154
constructor, 154, 155
function, 154
cv-qualifier, 158
declaration, 149
explicit, 152
friend, 154
function, 151
inline, 151
static, 150
storage class, 149
type, see type specifier
typedef, 152
virtual, 152
specifier access, see access specifier
stable algorithm, 442, 465
stack unwinding, 417
see exception handling, constructors and destructors, 417
standard
structure of, 5–6
standard deviation
normal_distribution, 970
standard-layout types, 74
standard-layout class, 227
standard-layout struct, 228
standard-layout union, 228
standard integer type, 75
standard signed integer type, 75
standard unsigned integer type, 75
start
program, 62
startup
program, 450, 462
state
discard_block_engine, 952
independent_bits_engine, 953
linear_congruential_engine, 948
mersenne_twister_engine, 949
object, 441
shuffle_order_engine, 954
subtract_with_carry_engine, 951
statement, 137–146
continue in for, 141
break, 142, 143
compound, 137
continue, 142, 143
declaration, 143
declaration in if, 138
declaration in switch, 138
declaration in for, 141
declaration in switch, 139
declaration in while, 140
do, 139, 141
empty, 137
expression, 137
for, 139, 141
goto, 137, 142, 143
if, 138, 139
iteration, 139–142
jump, 142
labeled, 137
null, 137
for, 141
selection, 138–139
switch, 138, 139, 143
while, 139, 140
statement, 137, 1218
statement-seq, 137, 1218
static, 149
destruction of local, 144
linkage of, 59, 150
overloading and, 300
static initialization, 63
static storage duration, 66
static type, see type, static
static_assert, 148
static_assert-declaration, 147, 1220
static_cast, see cast, static
<stddef.h>, 26, 30
<stdio.h>, 498
storage-class-specifier, 149, 1220
storage class, 34
storage duration, 65–69
  automatic, 65, 66
class member, 69
dynamic, 65–69, 116
local object, 66
register, 66
static, 65, 66
thread, 65, 66
storage management, see new, delete
stream
  arbitrary-positional, 439
  repositionable, 441
streambuf
  implementation-defined, 1010
strict pointer safety, 69
string
  distinct, 30
  null-terminated byte, 447
  null-terminated character type, 441
  null-terminated multibyte, 447
sizeof, 30
type of, 29
string literal, see literal, string
string-literal, 28, 1213
stringize, see #
struct
  standard-layout, 228
struct

class versus, 227
structure, 227
structure tag, see class name
student_t_distribution
  probability density function, 974
sub-expression, 1096
subobject, see also object model, 7
subscripting operator
  overloaded, 327
subsequence rule
  overloading, 321
subtract_with_carry_engine
  carry, 951
generation algorithm, 951
state, 951
textual representation, 951
transition algorithm, 951
subtraction
  implementation defined pointer, 125
subtraction operator, 124
suffix
  E, 28
e, 28
F, 28
f, 28
L, 25, 28
l, 25, 28
U, 25
u, 25
summary
  x C++ 2003, 1241
  x C++ 2011, 1248
  x C++ 2014, 1249
  compatibility with ISO C, 1232
swappable, 452
swappable with, 452
switch
  and handler, 414
  and try block, 414
  synchronize with, 12
synonym, 173
type name as, 152
syntax
  class member, 105
target object, 585
template, 333–413
  definition of, 333
  function, 392
  member function, 347
  primary, 355
static data member, 333
variable, 333
template, 333
template parameter, 35
template-argument, 338, 1227
template-argument-list, 338, 1227
template-declaration, 333, 1226
template-id, 337, 1227
template-name, 338, 1227
template-parameter, 334, 1227
template-parameter-list, 333, 1226
template name
linkage of, 334
template parameter scope, 44
temporary, 270
constructor for, 270
destruction of, 270
destructor for, 270
elimination of, 270, 294
implementation-defined generation of, 270
order of destruction of, 271
\texttt{terminate()}, 424, 425
called, 131, 416, 421, 424
termination
program, 62, 65
terminology
pointer, 77
text-line, 427, 1228
textual representation
\texttt{discard\_block\_engine}, 953
\texttt{independent\_bits\_engine}, 954
\texttt{shuffle\_order\_engine}, 955
\texttt{subtract\_with\_carry\_engine}, 951
this, 91, 235
type of, 235
\texttt{this} pointer, see this
thread, 11
thread of execution, 11
thread storage duration, 66
thread, blocked, 439
\texttt{thread\_local}, 149
threads
multiple, 11–15
throw, 130
\texttt{throw\_expression}, 130, 1218
throwing, see exception handling, throwing
\texttt{timed\_mutex\_types}, 1171
token, 21
alternative, 20
preprocessing, 19–20, 1210
token, 21, 1211

Cross references

traceable pointer object, 68, 467
\texttt{trailing\_return\_type}, 192, 1223
\texttt{trailing\_type\_specifier}, 157, 1220
\texttt{trailing\_type\_specifier\_seq}, 157, 1220
traits, 442
transfer ownership, 556
\texttt{transform}
regular expression traits, 1141
\texttt{transform}
regular expression traits, 1097
\texttt{transform\_primaryl}
regular expression traits, 1141
\texttt{transform\_primary}
regular expression traits, 1141
\texttt{transform\_primaryl}
regular expression traits, 1097
TransformationTrait, 603
transition algorithm
\texttt{discard\_block\_engine}, 952
\texttt{independent\_bits\_engine}, 953
\texttt{linear\_congruential\_engine}, 948
\texttt{mersenne\_twister\_engine}, 949
\texttt{shuffle\_order\_engine}, 954
\texttt{subtract\_with\_carry\_engine}, 951
\texttt{translate}
regular expression traits, 1141
\texttt{translate}
regular expression traits, 1097
\texttt{translate\_nocase}
regular expression traits, 1141
\texttt{translate\_nocase}
regular expression traits, 1097
translation
phases, 17–18
separate, see compilation, separate
\texttt{translation\_unit}, 17
\texttt{translation\_units}, 59
\texttt{translation\_unit}, 59, 1215
\texttt{translation\_unit}, 59
name and, 34
trigraph sequence, 1249
trivial types, 74
trivially copyable class, 227
trivially copyable types, 74
trivial class, 227
trivial class type, 119
trivial type, 119
truncation, 86
try, 414
\texttt{try\_block}, see exception handling, try block
\texttt{try\_block}, 414, 1227
tuple
and pair, 519

Type, 34, 73–78
arithmetic, 76
array, 76, 201
bitmask, 445, 446
Boolean, 75
char, 75
char16_t, 76
char32_t, 76
character, 75
character container, 439
class and, 226
compound, 76
cost, 157
destination, 213
double, 76
dynamic, 2
enumerated, 77, 445
example of incomplete, 74
extended integer, 75
extended signed integer, 75
extended unsigned integer, 75
float, 76
floating point, 75
function, 76, 200, 201
fundamental, 75
sizeof, 75
incomplete, 36, 37, 40, 73, 83, 102, 103, 105, 107, 108, 113, 115, 121, 131, 243
incompletely-defined object, 73
int, 75
integral, 75
long, 75
long double, 76
long long, 75
narrow character, 75
over-aligned, 80
POD, 74
pointer, 76
polymorphic, 249
referenceable, 441
short, 75
signed char, 75
signed integer, 75
similar, see similar types
standard integer, 75
standard signed integer, 75
standard unsigned integer, 75
static, 4
trivially copyable, 73

underlying wchar_t, 76
unsigned, 75
unsigned char, 75
unsigned int, 75
unsigned long, 75
unsigned long long, 75
unsigned short, 75
unsigned integer, 75
void, 76
volatile, 157
wchar_t, 76
type generator, see template
type specifier
auto, 162
const, 158
elaborated, 162
simple, 159
volatile, 158
type-id, 192, 1224
type-id-list, 419, 1228
type-name, 159, 1221
type-parameter, 334, 1227
type-parameter-key, 334, 1227
type-specifier
bool, 159
wchar_t, 159
type-specifier, 157, 1220
type-specifier-seq, 157, 1220
type_info, 108
typedef
function, 202
typedef
overloading and, 301
typedef-name, 152, 1220
typedef, 108
construction and, 287
destruction and, 287
<typename>, 487
typename, 162
typename-specifier, 365, 1227
types
implementation-defined, 445
implementation-defined exception, 467	
type checking
argument, 104
type conversion, explicit, see casting
type name, 192
nested, 242
scope of, 242
type pun, 112
type specifier
auto, 159
char, 159
char16_t, 159
char32_t, 159
decltype, 159, 161
double, 159
elaborated, 57
enum, 162
float, 159
int, 159
long, 159
short, 159
signed, 159
unsigned, 159
void, 159
volatile, 159

ud-suffix, 31, 1214
unary fold, 101
unary function, 586
unary left fold, 101
unary operator
  overloaded, 326
unary right fold, 101
unary-expression, 113, 1216
unary-operator, 113, 1217
UnaryTypeTrait, 603
unary operator
  interpretation of, 326
unblock, 442
undefined, 441, 460, 461, 463, 985, 986, 989–992,
  996, 1000, 1024
undefined behavior, see behavior, undefined, 880
underlying type, 76
unevaluated operand, 89
unexpected(), 425
  called, 421
Unicode required set, 437
uniform distributions, 960–962
uniform random number generator
  requirements, 939
uniform_int_distribution
  discrete probability function, 960
uniform_real_distribution
  probability density function, 961
union
  standard-layout, 228
union, 77, 238
  class versus, 227
  anonymous, 239
  global anonymous, 239
unique pointer, 556
unit
  translation, 449, 450, 461
universal character name, 17
universal-character-name, 19, 1210
unnamed-namespace-definition, 169, 1222
unordered, 63
unordered associative containers, 764–842
  begin, 772
  bucket, 772
  bucket_count, 772
  bucket_size, 772
  cbegin, 772
cend, 772
clear, 771
complexity, 764
  const_local_iterator, 766
count, 772
def, 772
equal_range, 772
equality function, 764
equivalent keys, 764, 765, 831, 838
erase, 771
  exception safety, 774
  find, 771
  hash function, 764
  hash_function, 769
  hasher, 765
  insert, 770, 771
  iterator invalidation, 774
  iterators, 773
  key_eq, 769
  key_equal, 766
  key_type, 765
  lack of comparison operators, 764
  load_factor, 773
  local_iterator, 766
  max_bucket_count, 772
  max_load_factor, 773
  rehash, 773
  requirements, 764, 765, 774
  unique keys, 764, 765, 825, 835
unordered_map
  element access, 829
  unique keys, 825
unordered_multimap
  equivalent keys, 831
unordered_multiset
  equivalent keys, 838
unordered_set
  unique keys, 835
unqualified-id, 91, 1215
unsequenced, 10
unsigned
typedef and, 149
unsigned-suffix, 25, 1213
unsigned integer type, 75
unspecified, 482, 483, 488, 911, 1073, 1257, 1259
unspecified behavior, see behavior, unspecified, 990
unwinding
stack, 417
uppercase, 23, 446
user-defined literal, see literal, user defined
overloaded, 328
user-defined-character-literal, 31, 1214
user-defined-floating-literal, 31, 1214
user-defined-integer-literal, 31, 1214
user-defined-literal, 31, 1214
user-provided, 209
Uses-allocator construction, 548
using-declaration, 174–179
using-declaration, 174, 1222
using-directive, 179–182
using-directive, 179, 1222
usual arithmetic conversions, see conversion, usual
arithmetic
UTF-8 character literal, 26
valid, 39
valid but unspecified state, 442
value, 73
call by, 104
indeterminate, 212
null member pointer, 86
null pointer, 86
undefined unrepresentable integral, 86
value category, 79
value computation, 10–11, 14, 15, 106, 119, 128,
129, 131, 132, 271
value representation, 73
value-initialization, 211
variable, 34
indeterminate uninitialized, 211
variable template
definition of, 333
virt-specifier, 231, 1225
virt-specifier-seq, 231, 1225
virtual base class, 245
virtual function, 249–254
pure, 254
virtual function call, 253

constructor and, 287
destructor and, 287
undefined pure, 255
visibility, 45
visible, 45
void*
type, 77
void&, 196
volatile, 77
constructor and, 236, 268
destructor and, 236, 275
implementation-defined, 159
overloading and, 302

waiting function, 1196
wchar_t, 26, 30, 686
implementation-defined, 76
weak result type, 585
weibull_distribution
probability density function, 968
weights
discrete_distribution, 975
piecewise_constant_distribution, 977
weights at boundaries
piecewise_linear_distribution, 978
well-formed program, see program, well-formed
wide-character, 26

X(X&), see copy constructor
xvalue, 79

zero
division by undefined, 88
remainder undefined, 88
undefined division by, 124
zero-initialization, 211
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).

abstract-declarator, 192, 1224
abstract-pack-declarator, 192, 1224
access-specifier, 243, 1226
additive-expression, 125, 1217
alias-declaration, 147, 1217
alignment-specifier, 185, 1222
and-expression, 128, 1218
asm-definition, 182, 1222
assignment-expression, 131, 1218
assignment-operator, 131, 1218
attribute, 185, 1222
attribute-argument-clause, 186, 1223
attribute-declaration, 147, 1220
attribute-list, 185, 1222
attribute-namespacer, 186, 1223
attribute-scoped-token, 186, 1223
attribute-specifier, 185, 1222
attribute-specifier-seq, 185, 1222
attribute-token, 185, 1222
balanced-token, 186, 1223
balanced-token-seq, 186, 1223
base-clause, 243, 1225
base-specifier, 243, 1226
base-specifier-list, 243, 1225
base-type-specifier, 243, 1226
binary-digit, 24, 1212
binary-literal, 24, 1212
declaration, 147, 1219
declaration-seq, 147, 1219
declaration-statement, 143, 1219
declarator, 191, 1223
declarator-id, 192, 1223
decl-type-specifier, 159, 1221
delete-expression, 120, 1217
digit, 22, 1211
digit-sequence, 28, 1213
dynamic-exception-specification, 419, 1227
elaborated-type-specifier, 162, 1221
elif-group, 426, 1228
elif-groups, 426, 1228
else-group, 426, 1228
empty-declaration, 147, 1220
enclosing-namespace-specifier, 169, 1222
encoding-prefix, 26, 1213
endif-line, 426, 1228
enum-base, 166, 1221
enum-head, 166, 1221
enum-key, 166, 1221

Cross references 1313
nonzero-digit, 24, 1212
noptr-abstract-declarator, 192, 1224
noptr-abstract-pack-declarator, 192, 1224
noptr-declarator, 191, 1223
noptr-new-declarator, 116, 1217
octal-digit, 24, 1212
octal-escape-sequence, 26, 1213
octal-literal, 24, 1212
opaque-enum-declaration, 166, 1221
operator, 325, 1226
operator-function-id, 325, 1226
parameter-declaration, 201, 1224
parameter-declaration-clause, 200, 1224
parameters-and-qualifiers, 191, 1223
pm-expression, 123, 1217
pointer-literal, 31, 1214
postfix-expression, 102, 1216
pp-number, 22, 1211
pp-tokens, 427, 1229
preprocessing-file, 426, 1228
preprocessing-op-or-punc, 23, 1212
preprocessing-token, 19, 1210
primary-expression, 91, 1215
pseudo-destructor-name, 102, 1216
ptr-abstract-declarator, 192, 1224
ptr-declarator, 191, 1223
ptr-operator, 192, 1223
pure-specifier, 231, 1225
q-char, 21, 1211
q-char-sequence, 21, 1211
qualified-id, 93, 1215
qualified-namespace-specifier, 173, 1222
r-char, 28, 1214
r-char-sequence, 28, 1214
raw-string, 28, 1214
ref-qualifier, 192, 1223
relational-expression, 126, 1217
replacement-list, 427, 1229
s-char, 28, 1214
s-char-sequence, 28, 1214
selection-statement, 138, 1219
shift-expression, 126, 1217
sign, 28, 1213
simple-capture, 94, 1215
simple-declaration, 147, 1220
simple-escape-sequence, 26, 1213
simple-template-id, 337, 1227
simple-type-specifier, 159, 1221
statement, 137, 1218
statement-seq, 137, 1218
static_assert-declaration, 147, 1220
storage-class-specifier, 149, 1220
string-literal, 28, 1213
template-argument, 338, 1227
template-argument-list, 338, 1227
template-declaration, 333, 1226
template-id, 337, 1227
template-name, 338, 1227
template-parameter, 334, 1227
template-parameter-list, 333, 1226
text-line, 427, 1228
throw-expression, 130, 1218
token, 21, 1211
trailing-return-type, 192, 1223
trailing-type-specifier, 157, 1220
trailing-type-specifier-seq, 157, 1220
translation-unit, 59, 1215
try-block, 414, 1227
type-id, 192, 1224
type-id-list, 419, 1228
type-name, 159, 1221
type-parameter, 334, 1227
type-parameter-key, 334, 1227
type-specifier, 157, 1220
type-specifier-seq, 157, 1220
typedef-name, 152, 1220
typename-specifier, 365, 1227
ud-suffix, 31, 1214
unary-expression, 113, 1216
unary-operator, 113, 1217
universal-character-name, 19, 1210
unnamed-namespace-definition, 169, 1222
unqualified-id, 91, 1215
unsigned-suffix, 25, 1213
user-defined-character-literal, 31, 1214
user-defined-floating-literal, 31, 1214
user-defined-integer-literal, 31, 1214
user-defined-literal, 31, 1214
user-defined-string-literal, 31, 1214
using-declaration, 174, 1222
using-directive, 179, 1222
virt-specifier, 231, 1225
virt-specifier-seq, 231, 1225

Cross references 1315
Index of library names

_exit, 479
__alignas_is_defined, 496
__bool_true_false_are_defined, 495, 496
_1, 596

a
cauchy_distribution, 973
extreme_value_distribution, 970
uniform_int_distribution, 961
uniform_real_distribution, 962
weibull_distribution, 969
abort, 65, 143, 449, 479, 487, 491
abs, 993, 1005
complex, 933
accumulate, 1002
acos, 993, 1005
complex, 934
acosh, 1005
complex, 934
address
 allocator, 552
 addressof, 554
adjacent_difference, 1003
adjacent_find, 899
adopt_lock, 1176
adopt_lock_t, 1176
advance, 863
<algorithm>, 886
align, 547
all
bitset, 539
all_of, 897
allocate
 allocator, 552
 allocator_traits, 550
 scoped_allocator_adaptor, 644
allocate_shared, 573
allocator, 1127
allocator, 551
 address, 552
 allocate, 552
 constructor, 552
deallocate, 552
 destructor, 552
 max_size, 552
operator!=, 552
operator==, 552
allocator_arg, 547
allocator_arg_t, 547
allocator_traits, 548
allocate, 550
const_pointer, 549
const_void_pointer, 549
constructor, 550
deallocate, 550
destructor, 550
difference_type, 549
is_always_equal, 550
max_size, 550
pointer, 549
propagate_on_container_copy_assignment, 550
propagate_on_container_move_assignment, 550
propagate_on_container_swap, 550
rebind_alloc, 550
select_on_container_copy_construction, 551
size_type, 549
void_pointer, 549
alpha
gamma_distribution, 968
always_noconv
codecvt, 711
any
bitset, 539
any_of, 897
append
 basic_string, 668, 669
apply
valarray, 990
arg, 935
 complex, 933
<algorithm>, 774
array, 776, 777, 779
begin, 777
data, 778
end, 777
fill, 778
get, 779

Cross references 1316
max_size, 777
size, 777, 778
swap, 778
asin, 993, 1005
complex, 934
asinh, 1005
complex, 934
<assert.h>, 450
assign
basic_regex, 1115, 1116
basic_string, 669, 670
error_code, 508
error_condition, 510
function, 601
async, 1205
at
basic_string, 667
map, 812
unordered_map, 829
at_quick_exit, 479, 480
atan, 993, 1005
complex, 934
atan2, 993, 1005
atanh, 1005
complex, 934
atexit, 65, 449, 479
<atomic>, 1142
atomic type
atomic_compare_exchange_strong, 1154
atomic_compare_exchange_strong_explicit, 1154
atomic_compare_exchange_weak, 1154
atomic_compare_exchange_weak_explicit, 1154
atomic_exchange, 1154
atomic_exchange_explicit, 1154
atomic_fetch_, 1155
atomic_is_lock_free, 1153
atomic_load, 1153
atomic_load_explicit, 1153
atomic_store, 1153
atomic_store_explicit, 1153
compare_exchange_strong, 1154
compare_exchange_strong_explicit, 1154
compare_exchange_weak, 1154
compare_exchange_weak_explicit, 1154
constructor, 1152
exchange, 1154
fetch_, 1155
load, 1153
operator C, 1153
operator++, 1156
operator--, 1156
operator=, 1153
store, 1153
atomic_compare_exchange_strong
atomic type, 1154
shared_ptr, 581
atomic_compare_exchange_strong_explicit
atomic type, 1154
shared_ptr, 581
atomic_compare_exchange_weak
atomic type, 1154
shared_ptr, 581
atomic_compare_exchange_weak_explicit
atomic type, 1154
shared_ptr, 581
atomic_flag
atomic_type, 1154
atomic_flag_clear, 1157
atomic_flag_clear_explicit, 1157
atomic_flag_test_and_set, 1157
atomic_flag_test_and_set_explicit, 1157
atomic_is_lock_free
atomic_type, 1153
shared_ptr, 580
atomic_load
atomic_type, 1153
shared_ptr, 580
atomic_load_explicit
atomic_type, 1153
shared_ptr, 580
atomic_signal_fence, 1158
atomic_store
atomic_type, 1153
shared_ptr, 580
atomic_store_explicit
atomic_type, 1153
shared_ptr, 580
atomic_thread_fence, 1158
b
cauchy_distribution, 973
extreme_value_distribution, 970
uniform_int_distribution, 961
uniform_real_distribution, 962
weibull_distribution, 969

back
  basic_string, 668
back_insert_iterator, 869
  back_insert_iterator, 869
back_inserter, 870
bad
  basic_ios, 1029
bad_alloc, 119, 481, 486, 487
  bad_alloc, 486
  bad_alloc::what
    implementation-defined, 486
bad_array_new_length, 486
  bad_array_new_length, 486
bad_cast, 107, 487, 488
  bad_cast, 488
  bad_cast::what
    implementation-defined, 489
bad_exception, 491
  bad_exception, 491
  bad_exception::what
    implementation-defined, 491
bad_function_call, 597
  bad_function_call, 597
bad_typeid, 108, 487, 489
  bad_typeid, 489
bad_weak_ptr, 567
  bad_weak_ptr, 567
  what, 567
base
  move_iterator, 874
  reverse_iterator, 866
basic_filebuf, 1010, 1080
  basic_filebuf, 1081
    constructor, 1081
destructor, 1082
  operator=, 1082
  swap, 1082
basic_filebuf<char>, 1079
basic_filebuf<wchar_t>, 1079
basic_fstream, 1010, 1091
  basic_fstream, 1091
    constructor, 1092
  operator=, 1092
  swap, 1092
basic_ifstream, 1010, 1087
  basic_ifstream, 1087
    constructor, 1087, 1088
  operator=, 1088
  swap, 1088
basic_ifstream<char>, 1079
basic_ifstream<wchar_t>, 1079
basic_ios, 1010, 1024
  basic_ios, 1025
    constructor, 1026
destructor, 1026
exceptions, 1029
fill, 1027
init, 1026
move, 1027
rdbuf, 1026
set_rdbuf, 1028
  swap, 1028
tie, 1026
basic_ios<char>, 1015
basic_ios<wchar_t>, 1015
basic_iostream, 1054
  basic_iostream, 1055
    constructor, 1055
destructor, 1055
  operator=, 1055
  swap, 1055
basic_istream, 1010, 1043
  basic_istream, 1045
    constructor, 1045
destructor, 1045, 1046
get, 1050, 1051, 1054
  operator<<, 1055
  operator=, 1045
seekg, 1054
  swap, 1045
tellg, 1053
basic_istream<char>, 1042
basic_istream<wchar_t>, 1042
basic_istringstream, 1010, 1074
  basic_istringstream, 1075
    constructor, 1075
  operator=, 1075
  str, 1076
  swap, 1075
basic_istringstream<char>, 1069
basic_istringstream<wchar_t>, 1069
basic_ostringstream, 1010, 1089
  basic_ostringstream, 1089
    constructor, 1089, 1090
  operator=, 1090
  swap, 1090
basic_ostringstream<char>, 1079
basic_ofstream<wchar_t>, 1079
basic_ostream, 1010, 1122
  basic_ostream, 1058
  constructor, 1058
  destructor, 1058, 1059
  operator<<, 1061, 1062, 1064
  operator=, 1058
  seekp, 1059
  swap, 1058
basic_ostream<char>, 1042
basic_ostream<wchar_t>, 1042
basic_ostringstream_iterator, 1010
basic_ostringstream, 1010, 1076
  basic_ostringstream, 1077
    constructor, 1077
    operator=, 1077
    str, 1077
    swap, 1077
basic_ostringstream<char>, 1069
basic_ostringstream<wchar_t>, 1069
basic_regex, 1098, 1112, 1139
  assign, 1115, 1116
  basic_regex, 1114, 1115
  constants, 1113, 1114
  constructor, 1114, 1115
  flag_type, 1116
  getloc, 1116
  imbue, 1116
  mark_count, 1116
  operator=, 1115
  swap, 1117
basic_streambuf, 1010, 1033
  basic_streambuf, 1035
    constructor, 1035
    destructor, 1035
    operator=, 1037
    setbuf, 1074
    swap, 1037
basic_streambuf<char>, 1032
basic_streambuf<wchar_t>, 1032
basic_string, 658, 679, 1069
  append, 668, 669
  assign, 669, 670
  at, 667
  back, 668
  begin, 666
  capacity, 667
  cbegin, 666
  cend, 666
  clear, 667
  compare, 678, 679

operator<<, 1061, 1062, 1064
operator=, 1058
operator++, 668
operator+=, 668
operator<, 681
operator<=, 682
operator<<, 683
operator<<, 683
operator=, 665, 666
operator==, 681
operator>, 682
operator>>, 683
operator[] , 667
pop_back, 672
push_back, 669
rbegin, 666
rend, 666
replace, 673, 674
reserve, 667
resize, 666, 667
rfind, 676
shrink_to_fit, 667
size, 666
substr, 678
swap, 675, 683
basic_stringbuf, 1010, 1070
basic_stringbuf<char>, 1069
basic_stringbuf<wchar_t>, 1071
constructor, 1071
operator=, 1071
str, 1072
swap, 1071, 1072
basic_stringstream, 1010, 1078

Cross references 1319
basic_stringstream, 1078
constructor, 1079
operator=, 1079
str, 1079
swap, 1079
before
type_info, 488
before_begin
forward_list, 787
begin, 494
array, 777
basic_string, 666
initializer_list, 494
match_results, 1126
valarray, 1001
begin(C&), 883
begin(initializer_list<E>, 495
begin(T (&)[N]), 883
bernoulli_distribution, 962
constructor, 963
p, 963
beta
gamma_distribution, 968
bidirectional_iterator_tag, 862
binary_negate, 594
binary_search, 914
bind, 595–597
binomial_distribution, 963
constructor, 963
p, 964
t, 964
bit_and, 593
bit_and<>, 593
bit_not<>, 594
bit_or, 593
bit_or<>, 593
bit_xor, 593
bit_xor<>, 593
<bitset>, 534
bitset, 534
bitset, 536, 537
flip, 538
operator[], 540
reset, 538
set, 538
boolalpha, 1029
byte_string
wstring_convert, 700
c_str
basic_string, 675
cacos
complex, 934
cacosh
complex, 934
call_once, 1186
calloc, 555, 1252
capacity
basic_string, 667
vector, 801
casin
complex, 934
casinh
complex, 934
<cassert>, 450
catan
complex, 934
catanh
complex, 934
category
error_code, 509
error_condition, 511
locale, 693
ciauchy_distribution, 972
a, 973
b, 973
constructor, 973
cbefore_begin
forward_list, 787
cbegin
basic_string, 666
cbegin(const C&), 883
cbegin(const C&), 883
<ccomplex>, 936
cend
basic_string, 666
cend(const C&), 884
cerr, 1013
<cerrno>, 461
<cfenv>, 925
CHAR_BIT, 477
char_class_type
regex_traits, 1109
CHAR_MAX, 477
char_traits, 650–653
char_type, 650
int_type, 650
off_type, 650
pos_type, 650
state_type, 650
cchar_type
cchar_traits, 650

Cross references
chi_squared_distribution, 972
constructor, 972
n, 972
chrono, 625
cin, 1013
<ciso646>, 1250
classic
locale, 698
classic_table
collapse, 709
clear
atomic_flag, 1157
basic_ios, 1028
basic_string, 667
error_code, 508
error_condition, 510
forward_list, 789
<climits>, 1258
<clocale>, 446, 1250
clock, 495, 496
clock_t, 496
CLOCKS_PER_SEC, 496
clog, 1013
close
basic_filebuf, 1083, 1093
basic_ifstream, 1088
basic_ofstream, 1090
messages, 738
code
future_error, 1196
system_error, 513
codecvt, 709, 742
always_noconv, 711
do_always_noconv, 713
do_encoding, 713
do_in, 711
do_length, 713
do_max_length, 713
do_out, 711
do_unshift, 712
encoding, 711
in, 711
length, 711
max_length, 711
out, 711
unshift, 711
codecvt_byname, 713
collate, 724
compare, 725
do_compare, 725
do_hash, 725
do_transform, 725
hash, 725
transform, 725
collate_byname, 726
combine
locale, 697
common_type, 630, 633
compare
basic_string, 678, 679
collate, 725
sub_match, 1118
compare_exchange_strong
atomic type, 1154
compare_exchange_strong_explicit
atomic type, 1154
compare_exchange_weak
atomic type, 1154
compare_exchange_weak_explicit
atomic type, 1154
complex
literals, 936
<complex>, 926
complex, 928
complex, 930
imag, 930
operator-, 932
operator/, 932
real, 930
<condition_variable>, 1187
condition_variable
constructor, 1188
destructor, 1188
notify_all, 1189
notify_one, 1189
wait, 1189
wait_for, 1190, 1191
wait_until, 1189, 1190
condition_variable_any
constructor, 1192
destructor, 1192
notify_all, 1192
notify_one, 1192
wait, 1192, 1193
wait_for, 1193, 1194
wait_until, 1193
conj, 935
complex, 933
const_pointer
allocator_traits, 549
const_pointer_cast
shared_ptr, 575
const_void_pointer
  allocator_traits, 549
construct
  scoped_allocator_adaptor, 644, 645
converted
  wstring_convert, 700
copy, 902
  basic_string, 674
copy_backward, 903
copy_n, 902
copyfmt
  basic_ios, 1027
copy_sign, 1005
cos, 993, 1005
  complex, 934
cosh, 993, 1005
  complex, 934
count, 899
  bitset, 539
duration, 632
count_if, 899
cout, 1013
crbegin
  basic_string, 666
crbegin(const C& c), 884
cref
  reference_wrapper, 588
crend
  basic_string, 666
crend(const C& c), 884
<csetjmp>, 461, 495, 496
cshift
  valarray, 990
<csignal>, 495, 496
<cstalign>, 495, 496
<cstdarg>, 461, 495, 496
<cstdbool>, 495, 496
<cstddf>, 1250, 1252
<cst dint>, 478
<cstdio>, 1013, 1014, 1080, 1083, 1084, 1250
<cstdlib>, 449, 495, 496, 1250, 1253
<cstring>, 447, 1250, 1258, 1262
<ctgmath>, 1004
<ctime>, 495, 496, 691, 1250
ctype, 704
do_is, 705
do_narrow, 706
do_scan_not, 706
do_tolower, 706
do_toupper, 706
do_widen, 706
is, 705
narrow, 705
scan_is, 705
scan_not, 705
tolower, 705
toupper, 705
widen, 705
ctype<char>, 707
classic_table, 709
destructor, 708
ctype<char>, 708
destructor, 708
do_narrow, 709
do_tolower, 709
do_toupper, 709
do_widen, 709
is, 708
narrow, 709
scan_is, 708
scan_not, 709
table, 709
tolower, 709
toupper, 709
widen, 709
ctype_base, 703
do_scan_is, 705
ctypebyname, 707
<cuchar>, 461
curr_symbol
  moneypunct, 736
current_exception, 492
<cwchar>, 461, 1250
data
  basic_string, 675
  array, 778
  vector, 802
date_order
  time_get, 727
DBL_DIG, 477
DBL_EPSILON, 477
DBL_MANT_DIG, 477
DBL_MAX, 477
DBL_MAX_10_EXP, 477
DBL_MAX_EXP, 477
DBL_MIN, 477
DBL_MIN_10_EXP, 477
DBL_MIN_EXP, 477
deallocate
  allocator, 552
  allocator_traits, 550

Cross references 1322
end(T (&)[N]), 883
endl, 1061, 1063
ends, 1064
entropy
    random_device, 957
eof
    basic_ios, 1029
epptr
    basic_streambuf, 1038
epsilon
    numeric_limits, 472
eq
    char_traits, 675–678
equal, 900
    istreambuf_iterator, 882
equal_range, 914
equal_to, 590
equal_to<>, 590
equivalent
    error_category, 506, 507
erase
deque, 783
    list, 795
    basic_string, 672
    vector, 802
erase_after
    forward_list, 788
erased
    forward_list, 788
erf, 1005
erfc, 1005
errc, 502
error_category, 502, 505
    constructor, 506
    default_error_condition, 506
destructor, 506
equivalent, 506, 507
message, 506
name, 506
operator!=, 506
operator<, 506
operator==, 506
error_code, 502, 507, 509
    assign, 508
category, 509
clear, 508
default_error_condition, 509
message, 508
operator bool, 509
operator!=, 509
operator<, 509
operator==, 508
operator==, 511
value, 509
error_condition, 502
    assign, 510
category, 511
clear, 510
    constructor, 510
message, 511
operator bool, 511
operator!, 511
operator<, 511
operator==, 510
operator==, 511
value, 511
error_type, 1107, 1108
exception
    bad_function_call, 597
    bad_weak_ptr, 567
<exception>, 489
exception, 490
    constructor, 490
destructor, 490
exception_ptr, 492
exceptions
    basic_ios, 1029
exchange
    atomic type, 1154
exit, 62, 64, 143, 449, 479, 480, 487
EXIT_FAILURE, 479
EXIT_SUCCESS, 479
exp, 993, 1005
    complex, 934
exp2, 1005
expired
    weak_ptr, 577
expm1, 1005
exponential_distribution, 966
    constructor, 967
    lambda, 967
extreme_value_distribution, 969
    a, 970
    b, 970
    constructor, 969
facet
    locale, 694
fail
    basic_ios, 1029

Cross references 1325
failed
    ostreambuf_iterator, 883
failure
    ios_base::failure, 1018
false
    numpunct, 723
fclose
    1084
fdim
    1005
FE_ALL_EXCEPT
    925
FE_DFL_ENV
    925
FE_DIVBYZERO
    925
FE_DOWNWARD
    925
FE_INEXACT
    925
FE_INVALID
    925
FE_OVERFLOW
    925
FE_TONEAREST
    925
FE_TOWARDZERO
    925
FE_UNDERFLOW
    925
femclemexcept
    925
fegetenv
    925
fegetexceptflag
    925
fegetround
    925
feholdexcept
    925
fenv_t
    925
feraiseexcept
    925
fesetenv
    925
fesetexceptflag
    925
fesetround
    925
fetch
    atomic type, 1155
fetestexcept
    925
fespecial
    925
fesetround
    925
filebuf
    1010, 1079
fill
    905
    array, 778
    basic_ios, 1027
gslice_array
    998
indirect_array
    1000
mask_array
    999
slice_array
    995
fill_n
    905
find
    898
    basic_string, 675
find_end
    898
find_first_not_of
    basic_string, 677
find_last_of
    basic_string, 677
fisher_f_distribution
    973
    constructor, 974
m
    974
n
    974
fixed
    1031
flag_type
    basic_regex, 1116
flags
    ios_base, 703, 1020
flip
    bitset, 538
    bitset, 538
    vector< bool >, 805
float_denorm_style
    469, 475
    numeric_limits, 473
float_round_style
    469, 475
floor
    1005
FLT_DIG
    477
FLT_EPSILON
    477
FLT_EVAL_METHOD
    477
FLT_MANT_DIG
    477
FLT_MAX
    477
FLT_MAX_10_EXP
    477
FLT_MAX_EXP
    477
FLT_MIN
    477
FLT_MIN_10_EXP
    477
FLT_MIN_EXP
    477
FLT_RADIX
    477
FLT_ROUNDS
    477
flush
    1020, 1046, 1059, 1064
    basic_ostream, 1063
fma
    1005
fmax
    1005
fmin
    1005
fmtflags
    ios, 1065
    ios_base, 1018, 1020
fopen
    1083
for_each
    897
format
    match_results, 1126, 1127
format_default
    1105
format_default
    1107
format_first_only
    1105, 1132
format_first_only
    1107
format_no_copy
    1105, 1132
Cross references 1326
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>format_no_copy, 1107</td>
<td></td>
</tr>
<tr>
<td>format_sed, 1105</td>
<td></td>
</tr>
<tr>
<td>format_sed, 1107</td>
<td></td>
</tr>
<tr>
<td>forward, 517</td>
<td></td>
</tr>
<tr>
<td>forward_as_tuple, 530</td>
<td></td>
</tr>
<tr>
<td>forward_iterator_tag, 862</td>
<td></td>
</tr>
<tr>
<td>&lt;forward_list&gt;, 775</td>
<td></td>
</tr>
<tr>
<td>forward_list</td>
<td></td>
</tr>
<tr>
<td>before_begin, 787</td>
<td></td>
</tr>
<tr>
<td>cbefore_begin, 787</td>
<td></td>
</tr>
<tr>
<td>clear, 789</td>
<td></td>
</tr>
<tr>
<td>emplace_after, 788</td>
<td></td>
</tr>
<tr>
<td>emplace_front, 787</td>
<td></td>
</tr>
<tr>
<td>erase_after, 788</td>
<td></td>
</tr>
<tr>
<td>erased, 788</td>
<td></td>
</tr>
<tr>
<td>forward_list, 786, 787</td>
<td></td>
</tr>
<tr>
<td>front, 787</td>
<td></td>
</tr>
<tr>
<td>insert_after, 787, 788</td>
<td></td>
</tr>
<tr>
<td>merge, 790</td>
<td></td>
</tr>
<tr>
<td>pop, 787</td>
<td></td>
</tr>
<tr>
<td>push_front, 787</td>
<td></td>
</tr>
<tr>
<td>remove, 790</td>
<td></td>
</tr>
<tr>
<td>remove_if, 790</td>
<td></td>
</tr>
<tr>
<td>resize, 788</td>
<td></td>
</tr>
<tr>
<td>reverse, 790</td>
<td></td>
</tr>
<tr>
<td>sort, 790</td>
<td></td>
</tr>
<tr>
<td>splice_after, 789</td>
<td></td>
</tr>
<tr>
<td>swap, 791</td>
<td></td>
</tr>
<tr>
<td>unique, 790</td>
<td></td>
</tr>
<tr>
<td>fpclassify, 1007</td>
<td></td>
</tr>
<tr>
<td>fpos, 1015, 1023, 1024</td>
<td></td>
</tr>
<tr>
<td>state, 1023</td>
<td></td>
</tr>
<tr>
<td>frac_digits</td>
<td></td>
</tr>
<tr>
<td>moneypunct, 736</td>
<td></td>
</tr>
<tr>
<td>free, 555</td>
<td></td>
</tr>
<tr>
<td>freeze</td>
<td></td>
</tr>
<tr>
<td>ostrstream, 1262</td>
<td></td>
</tr>
<tr>
<td>strstream, 1264</td>
<td></td>
</tr>
<tr>
<td>strstreambuf, 1258</td>
<td></td>
</tr>
<tr>
<td>frexp, 1005</td>
<td></td>
</tr>
<tr>
<td>from_bytes</td>
<td></td>
</tr>
<tr>
<td>wstring_convert, 700</td>
<td></td>
</tr>
<tr>
<td>from_time_t, 639</td>
<td></td>
</tr>
<tr>
<td>front</td>
<td></td>
</tr>
<tr>
<td>basic_string, 668</td>
<td></td>
</tr>
<tr>
<td>forward_list, 787</td>
<td></td>
</tr>
<tr>
<td>front_insert_iterator, 870</td>
<td></td>
</tr>
<tr>
<td>front_insert_iterator, 870</td>
<td></td>
</tr>
<tr>
<td>front_inserter, 870</td>
<td></td>
</tr>
<tr>
<td>fseek, 1083</td>
<td></td>
</tr>
<tr>
<td>&lt;fstream&gt;, 1079</td>
<td></td>
</tr>
<tr>
<td>fstream, 1010</td>
<td></td>
</tr>
<tr>
<td>function, 597</td>
<td></td>
</tr>
<tr>
<td>assign, 601</td>
<td></td>
</tr>
<tr>
<td>bool conversion, 601</td>
<td></td>
</tr>
<tr>
<td>destructor, 600</td>
<td></td>
</tr>
<tr>
<td>function, 599</td>
<td></td>
</tr>
<tr>
<td>invocation, 600</td>
<td></td>
</tr>
<tr>
<td>operator!=, 601</td>
<td></td>
</tr>
<tr>
<td>operator(), 601</td>
<td></td>
</tr>
<tr>
<td>operator=, 600</td>
<td></td>
</tr>
<tr>
<td>operator==, 601</td>
<td></td>
</tr>
<tr>
<td>swap, 601, 602</td>
<td></td>
</tr>
<tr>
<td>target, 601</td>
<td></td>
</tr>
<tr>
<td>target_type, 601</td>
<td></td>
</tr>
<tr>
<td>&lt;functional&gt;, 582</td>
<td></td>
</tr>
<tr>
<td>future</td>
<td></td>
</tr>
<tr>
<td>constructor, 1200, 1201</td>
<td></td>
</tr>
<tr>
<td>get, 1201</td>
<td></td>
</tr>
<tr>
<td>operator=, 1201</td>
<td></td>
</tr>
<tr>
<td>share, 1201</td>
<td></td>
</tr>
<tr>
<td>valid, 1201</td>
<td></td>
</tr>
<tr>
<td>wait, 1202</td>
<td></td>
</tr>
<tr>
<td>wait_for, 1202</td>
<td></td>
</tr>
<tr>
<td>wait_until, 1202</td>
<td></td>
</tr>
<tr>
<td>future_category, 1195</td>
<td></td>
</tr>
<tr>
<td>future_errc</td>
<td></td>
</tr>
<tr>
<td>make_error_code, 1195</td>
<td></td>
</tr>
<tr>
<td>make_error_condition, 1196</td>
<td></td>
</tr>
<tr>
<td>future_error</td>
<td></td>
</tr>
<tr>
<td>code, 1196</td>
<td></td>
</tr>
<tr>
<td>what, 1196</td>
<td></td>
</tr>
<tr>
<td>gamma_distribution, 967</td>
<td></td>
</tr>
<tr>
<td>alpha, 968</td>
<td></td>
</tr>
<tr>
<td>beta, 968</td>
<td></td>
</tr>
<tr>
<td>constructor, 968</td>
<td></td>
</tr>
<tr>
<td>gbump</td>
<td></td>
</tr>
<tr>
<td>basic_streambuf, 1037</td>
<td></td>
</tr>
<tr>
<td>gcount</td>
<td></td>
</tr>
<tr>
<td>basic_istream, 1050</td>
<td></td>
</tr>
<tr>
<td>generate, 905</td>
<td></td>
</tr>
<tr>
<td>seed_seq, 958</td>
<td></td>
</tr>
<tr>
<td>generate_canonical, 960</td>
<td></td>
</tr>
<tr>
<td>generate_n, 905</td>
<td></td>
</tr>
<tr>
<td>generic_category, 505, 507</td>
<td></td>
</tr>
<tr>
<td>geometric_distribution, 964</td>
<td></td>
</tr>
<tr>
<td>constructor, 964</td>
<td></td>
</tr>
<tr>
<td>p, 964</td>
<td></td>
</tr>
<tr>
<td>get</td>
<td></td>
</tr>
<tr>
<td>array, 779</td>
<td></td>
</tr>
<tr>
<td>basic_istream, 1050, 1051, 1054</td>
<td></td>
</tr>
</tbody>
</table>
future, 1201
messages, 738
money_get, 732
num_get, 715
pair, 522, 523
reference_wrapper, 587
shared_future, 1204
shared_ptr, 572
time_get, 728
tuple, 531, 532
unique_ptr, 562
get_allocator
   basic_string, 675
match_results, 1127
get_date
   time_get, 727
get_deleter
   shared_ptr, 575
   unique_ptr, 562
get_future
   packaged_task, 1208
   promise, 1198
get_id
   this_thread, 1167
   thread, 1166
get_money, 1066
get_monthname
   time_get, 727
get_new_handler, 462
get_pointer_safety, 547
get_temporary_buffer, 553
get_terminate, 462
get_time, 1067
   time_get, 727
get_unexpected, 462
get_weekday
   time_get, 727
get_year
   time_get, 727
gpctr
   basic_streambuf, 1037
greater, 590
greater<, 591
greater_equal, 590
greater_equal<, 591
grouping
   moneypunct, 736
   numpunct, 723
gslice, 995
   constructor, 996
gslice_array, 997
hardware_concurrency
   thread, 1166
has_denorm_loss
   numeric_limits, 473
has_facet
   locale, 698
has_infinity
   numeric_limits, 473
has_quiet_NaN
   numeric_limits, 473
has_signaling_NaN
   numeric_limits, 473
hash, 511, 581, 582, 602, 647, 686
collate, 725
hash_code, 540
   type_info, 488
   type_index, 647
hex, 1031
hexfloat, 1031
hypot, 1005
id
   locale, 695
idxl
   operator>, 635
ifstream, 1010, 1079
ignore, 530
   basic_istream, 1052
ilogb, 1005
imag, 935
   complex, 930, 933
imbue, 1111
   basic_filebuf, 1086
   basic_istream, 1035
ios_base, 1021
global
   locale, 698
good
   basic_ios, 1029
Cross references 1328
codecvt, 711
in_avail
  basic_streambuf, 1036
includes, 915
independent_bits_engine, 953
index_sequence, 516
index_sequence_for, 516
indirect_array, 999
  operator[], 1000
infinity
  numeric_limits, 473
Init
  ios_base::Init, 1020
init
  basic_ios, 1026, 1045, 1058
<initializer_list>, 494
initializer_list, 494
  begin, 494
  end, 494
  initializer_list, 494
  size, 495
inner_allocator
  scoped_allocator_adaptor, 644
inner_allocator_type
  scoped_allocator_adaptor, 642
inner_product, 1002
inplace_merge, 915
input_iterator_tag, 862
emplace
deque, 783
insert
deque, 783
  list, 795
  basic_string, 671, 672
map, 812
multimap, 816
unordered_map, 830
unordered_multimap, 834
vector, 802
push_back
deque, 783
push_front
deque, 783
insert_after
  forward_list, 787, 788
insert_iterator, 871
insert_iterator, 872
insert_or_assign
  map, 812
  unordered_map, 830
inserter, 872

int16_t, 478
int32_t, 478
int64_t, 478
int8_t, 478
int_fast16_t, 478
int_fast32_t, 478
int_fast64_t, 478
int_fast8_t, 478
int_least16_t, 478
int_least32_t, 478
int_least64_t, 478
int_least8_t, 478
INT_MAX, 477
INT_MIN, 477
int_type
  char_traits, 650
  wstring_convert, 701
integer_sequence, 534
internal, 1030
intervals
  piecewise_constant_distribution, 978
  piecewise_linear_distribution, 980
intmax_t, 478
intptr_t, 478
invalid_argument, 498, 499, 536
  invalid_argument, 499
INVOKE, 585, 586
invoke, 586
<iomanip>, 1042
<ios>, 1014
ios, 1010, 1015
ios_base, 1015
  destructor, 1023
  fmtflags, 1020
  ios_base, 1023
  iostate, 1018
  precision, 1021
  setf, 1021
  streamsize, 1021
ios_base::failure, 1018
ios_base::Init, 1020
  destructor, 1020
<iomanip>, 1010
iostate
  ios_base, 1018
<iostream>, 1012
iostream_category, 1031
iota, 1004
is
  ctype, 705
  ctype<char>, 708

Cross references 1329
is_always_equal
  allocator_traits, 550
    scoped_allocator_adaptor, 643
is_bind_expression, 595
is_bounded
  numeric_limits, 474
is_error_code_enum, 502
is_error_condition_enum, 502
is_exact
  numeric_limits, 472
is_heap, 919
is_heap_until, 919
is_iec559
  numeric_limits, 474
is_integer
  numeric_limits, 472
is_modulo
  numeric_limits, 474
is_open
  basic_filebuf, 1083, 1092
    basic_ifstream, 1088
    basic_ofstream, 1090
is_partitioned, 909
is_permutation, 901
is_placeholder, 595
is_signed
  numeric_limits, 472
is_sorted, 912
is_sorted_until, 912
isalnum, 698
isalpha, 698
isspace, 698
<istream>, 1041
istream, 1010, 1042
istream_iterator, 877
  constructor, 878
    destructor, 878
    operator!=, 879
    operator*, 878
    operator++, 878
    operator->, 878
    operator==, 878, 879
istreambuf_iterator, 880
  constructor, 881
    operator++, 882
istringstream, 1010, 1069
istrstream, 1261
  constructor, 1261
  istrstream, 1261
isunordered, 1007
isupper, 698
isdigit, 698
iter_swap, 904
<iterator>, 857
iword
  ios_base, 1022
jmp_buf, 496
join
  thread, 1166
joinable
  thread, 1166
kill_dependency, 1147
knuth_b, 956
lambda
  exponential_distribution, 967
LDBL_DIG, 477
LDBL_EPSILON, 477
LDBL_MANT_DIG, 477
LDBL_MAX, 477
LDBL_MAX_10_EXP, 477
LDBL_MAX_EXP, 477
LDBL_MIN, 477
LDBL_MIN_10_EXP, 477
LDBL_MIN_EXP, 477
left, 1030
length
  char_traits, 664, 666, 670, 681
    basic_string, 666
    codecvt, 711
Cross references
match_results, 1126
regex_traits, 1109
sub_match, 1117
valarray, 990
length_error, 498, 500, 658
length_error, 500
less, 590
less<>, 591
less_equal, 590
less_equal<>, 591
lexicographical_compare, 921
lgamma, 1005
<limits>, 469
linear_congruential_engine, 948
constructor, 949
<list>, 775
list, 791
list, 793, 794
splice, 796
swap, 797
literals
complex, 936
LLONG_MAX, 477
LLONG_MIN, 477
lrint, 1005
lround, 1005
load
atomic type, 1153
<locale>, 690, 691
locale, 1111, 1116, 1139
category, 693
classic, 698
combine, 697
constructor, 696
destructor, 697
facet, 694
global, 698
has_facet, 698
id, 695
name, 697
operator!=, 697
operator(), 697
operator=, 696
operators==, 697
use_facet, 698
lock, 1185
shared_lock, 1183
unique_lock, 1180
weak_ptr, 577
lock_guard
constructor, 1177
destructor, 1177
log, 993, 1005
complex, 934
log10, 993, 1005
complex, 935
log1p, 1005
log2, 1005
logb, 1005
logic_error, 498
logic_error, 499
logical_and, 592
logical_and<>, 592
logical_not, 592
logical_not<>, 592
logical_or, 592
logical_or<>, 592
lognormal_distribution, 971
constructor, 971
m, 971
s, 971
LONG_MAX, 477
longjmp, 495, 496
lookup_classname
regex_traits, 1110
regular expression traits, 1140
lookup_collatename
regex_traits, 1109
regular expression traits, 1140
lower_bound, 913
lowest
numeric_limits, 471
lrint, 1005
lround, 1005

m
fisher_f_distribution, 974
lognormal_distribution, 971
make_error_code, 502, 509, 1031
future_errc, 1195
make_error_condition, 502, 511, 1031
future_errc, 1196
make_exception_ptr, 493
make_heap, 918
make_index_sequence, 516
make_integer_sequence, 534
make_move_iterator, 876
make_pair, 522
make_ready_at_thread_exit
packaged_task, 1209
make_reverse_iterator, 869
make_shared, 573

Cross references
make_tuple, 529
make_unique, 565
malloc, 555, 1252
<map>, 805
map, 807
c onstructor, 811
insert, 812
insert_or_assign, 812
map, 811
operator<, 811
operator==, 811
swap, 812
t ry_emplace, 812
mark_count
basic_regex, 1116
mask_array, 998
operator[], 998
match_any, 1105
match_any, 1107
match_continuous, 1105, 1135
match_continuous, 1107
match_default, 1105
match_flag_type, 1105, 1106, 1141
match_not_bol, 1105
match_not_bol, 1106
match_not_bow, 1105
match_not_bow, 1107
match_not_eol, 1105
match_not_eol, 1106
match_not_eow, 1105
match_not_eow, 1107
match_not_null, 1105, 1135
match_not_null, 1107
match_prev_avail, 1105, 1135
match_prev_avail, 1107
match_results, 1123, 1133, 1136
begin, 1126
empty, 1125
end, 1126
format, 1126, 1127
g et_allocator, 1127
length, 1126
match_results, 1124, 1125
matched, 1123
max_size, 1125
operator!=, 1128
operator==, 1128
operator[], 1126
position, 1126
prefix, 1126
size, 1125
state, 1125
str, 1126
suffix, 1126
swap, 1127
max, 920
duration, 633
duration_values, 630
numeric_limits, 471
time_point, 638
valarray, 990
max_align_t, 468, 469
max_digits10
numeric_limits, 472
max_element, 921
max_exponent
numeric_limits, 473
max_exponent10
numeric_limits, 473
max_length
codecv t, 711
max_size
allocator, 552
allocator_traits, 550
array, 777
basic_string, 666
match_results, 1125
scoped_allocator_adaptor, 644
MB_LEN_MAX, 477
mean
normal_distribution, 971
poisson_distribution, 966
student_t_distribution, 975
mem_fn, 597
memchr, 687
<memory>, 541
merge, 914
list, 796
forward_list, 790
mersenne_twister_engine, 949
constructor, 950
message
do_close, 739
error_category, 506
error_code, 509
error_condition, 511
messages, 738
close, 738
do_get, 739
do_open, 738
g et, 738

Cross references 1332
narrow
  basic_ios, 1027
cctype, 705
cctype<char>, 709
NDEBUG, 450
nearbyint, 1005
negate, 588
negate<>, 589
negative_binomial_distribution, 965
  constructor, 965
  p, 965
t, 965
negative_sign
moneypunct, 736
nested_exception, 493
  nested_exception, 493
  nested_ptr, 493
rethrow_if_nested, 494
rethrow_nested, 493
throw_with_nested, 493
nested_ptr
  nested_exception, 493
<new>, 481
new
  operator, 481, 485
  operator, 461, 462, 482–485, 555
new_handler, 487
next, 864
next_permutation, 922
nextafter, 1005
nexttoward, 1005
noboolalpha, 1029
none
  bitset, 539
  none_of, 897
norm, 935
  complex, 933
normal_distribution, 970
  constructor, 970
  mean, 971
  stddev, 971
noshownbase, 1029
noshowpoint, 1029
noshowpos, 1030
noskipws, 1030
not1, 594
not2, 594
not_equal_to, 590
not_equal_to<>, 591
notify_all
  condition_variable, 1189
  condition_variable_any, 1192
notify_all_at_thread_exit, 1187
notify_one
  condition_variable, 1189
  condition_variable_any, 1192
nounitbuf, 1030
nounuppercase, 1030
nth_element, 912
NULL, 468, 469
nullptr_t, 468, 469
num_get, 714
  do_get, 716, 718
  get, 715
num_put, 718
  do_put, 719, 722
  put, 719
<numeric>, 1001
numeric_limits, 470
numeric_limits, 469
  denorm_min, 474
  digits, 471
digits10, 472
epsilon, 472
float_denorm_style, 473
has_denorm_loss, 473
has_infinity, 473
has_quiet_NaN, 473
has_signaling_NaN, 473
infinity, 473
is_bounded, 474
is_exact, 472
is_iec559, 474
is_integer, 472
is_modulo, 474
is_signed, 472
lowest, 471
max, 471
max_digits10, 472
max_exponent, 473
max_exponent10, 473
min, 471
min_exponent, 472
min_exponent10, 472
quiet_NaN, 474
radix, 472
round_error, 472
round_style, 475
signaling_NaN, 474
tinyness_before, 475
traps, 474
numeric_limits<bool>, 477

Cross references 1334
valarray, 991
operator**=
  complex, 931
duration, 633
gslice_array, 998
indirect_array, 1000
mask_array, 999
slice_array, 995
valarray, 989
operator+
  basic_string, 679, 680
complex, 932
duration, 632, 638
move_iterator, 875, 876
reverse_iterator, 867, 868
time_point, 638
valarray, 988, 991
operator++
  atomic type, 1156
back_insert_iterator, 870
duration, 632
front_insert_iterator, 871
insert_iterator, 872
istream_iterator, 878
istreambuf_iterator, 882
move_iterator, 875
ostream_iterator, 880
ostreambuf_iterator, 883
raw_storage_iterator, 553
regex_iterator, 1135, 1136
regex_token_iterator, 1139
reverse_iterator, 866
operator++=
  basic_string, 668
complex, 931
duration, 633
gslice_array, 998
indirect_array, 1000
mask_array, 999
move_iterator, 875
reverse_iterator, 867
slice_array, 995
time_point, 637
valarray, 989
operator-=
  complex, 931
duration, 633
gslice_array, 998
indirect_array, 1000
mask_array, 999
move_iterator, 875
reverse_iterator, 867
slice_array, 995
time_point, 637
valarray, 989
operator->
  istream_iterator, 878
move_iterator, 875
regex_iterator, 1135
regex_token_iterator, 1139
reverse_iterator, 866
shared_ptr, 572
unique_ptr, 562
operator--
  atomic type, 1156
duration, 632
move_iterator, 875
reverse_iterator, 867
operator/
  complex, 932
duration, 634
valarray, 991
operator/=  
  complex, 931
duration, 633
gslice_array, 998
indirect_array, 1000
mask_array, 999
move_iterator, 875
reverse_iterator, 867
slice_array, 995
valarray, 989
operator<<
  basic_string, 681
complex, 932
duration, 635
error_category, 506
error_code, 509
error_condition, 511
move_iterator, 876
queue, 845
reverse_iterator, 868
shared_ptr, 573, 574
stack, 850
sub_match, 1118–1122
thread::id, 1164
time_point, 638

Cross references 1336
tuple, 532
  type_index, 647
  unique_ptr, 565, 566
valarray, 992
operator<
  shared_ptr, 575
  sub_match, 1122
operator<<
  bitset, 539, 540
  complex, 933
operator<=
  bitset, 537
operator<, 516
  pair, 522
  basic_string, 682
  duration, 635
  move_iterator, 876
  queue, 845
  reverse_iterator, 868
  shared_ptr, 566, 574
  stack, 851
  sub_match, 1118–1122
  thread::id, 1164
  time_point, 638
  tuple, 533
  type_index, 647
  unique_ptr, 566
valarray, 992
operator<<
  basic_istream, 1055
  basic_ostream, 1060–1062, 1064
  basic_string, 683
  error_code, 509
  thread::id, 1164
  valarray, 991
operator<<=
  gslice_array, 998
  indirect_array, 1000
  mask_array, 999
  slice_array, 995
valarray, 989
operator=
  bad_alloc, 486
  bad_cast, 488
  bad_exception, 491
  bad_typeid, 489
  reverse_iterator, 866
  atomic type, 1153
  back_insert_iterator, 870
  basic_filebuf, 1082
  basic_fstream, 1092
error_code, 511
error_condition, 511
function, 601
istream_iterator, 878
istreambuf_iterator, 882
match_results, 1128
move_iterator, 876
queue, 845
regex_iterator, 1135
regex_token_iterator, 1136, 1139
reverse_iterator, 867
scoped_allocator_adaptor, 646
shared_ptr, 573
stack, 850
sub_match, 1118–1122
thread::id, 1164
time_point, 638
tuple, 532
type_index, 647
unique_ptr, 565, 566
valarray, 992
operator>>, 516
pair, 521
basic_string, 682
idxl, 635
move_iterator, 876
queue, 845
reverse_iterator, 868
shared_ptr, 574
stack, 851
sub_match, 1118–1122
thread::id, 1164
time_point, 639
tuple, 533
type_index, 647
unique_ptr, 566
valarray, 992
operator>>=, 516
pair, 522
basic_string, 682
duration, 635
move_iterator, 876
queue, 845
reverse_iterator, 868
shared_ptr, 574
stack, 851
sub_match, 1118–1122
thread::id, 1164
time_point, 639
tuple, 533
type_index, 647
unique_ptr, 566
valarray, 992
operator%>, 516
pair, 521
basic_string, 682
duration, 635
move_iterator, 876
queue, 845
reverse_iterator, 868
shared_ptr, 574
stack, 851
sub_match, 1118–1122
thread::id, 1164
time_point, 639
tuple, 533
type_index, 647
unique_ptr, 566
valarray, 992
operator%>>, 516
pair, 522
basic_string, 682
duration, 635
move_iterator, 876
queue, 845
reverse_iterator, 868
shared_ptr, 574
stack, 851
sub_match, 1118–1122
thread::id, 1164
time_point, 639
tuple, 533
type_index, 647
unique_ptr, 566
valarray, 992

Cross references

unique_ptr, 566
valarray, 992
operator>>
bitset, 539, 540
complex, 932
operator>>>=
bitset, 537
operator>>>
basic_istream, 1048
basic_string, 683
istream, 1047–1049
valarray, 991
operator>>>=
gslice_array, 998
indirect_array, 1000
mask_array, 999
slice_array, 995
valarray, 989
operator[]
basic_string, 667
bitset, 540
indirect_array, 1000
map, 811
mask_array, 998
match_results, 1126
move_iterator, 876
reverse_iterator, 867
unique_ptr, 564
unordered_map, 829
valarray, 986–988
operator%=
duration, 634
valarray, 991
operator%=
duration, 633
gslice_array, 998
indirect_array, 1000
mask_array, 999
slice_array, 995
valarray, 989
operator&
bitset, 540
valarray, 991
operator&=
bitset, 537
gslice_array, 998
indirect_array, 1000
mask_array, 999
slice_array, 995
valarray, 989
operator&&
valarray, 991, 992
operator~
  bitset, 540
  valarray, 991
operator~=
  bitset, 537
  gslice_array, 998
  indirect_array, 1000
  mask_array, 999
  slice_array, 995
  valarray, 989
operator~
  bitset, 538
  valarray, 988
operators==
  locale, 697
operator|
  bitset, 540
  valarray, 991
operator||
  valarray, 991, 992
<ostream>, 1042
ostream, 1010, 1042
ostream_iterator, 879
  constructor, 879
  destructor, 879
  operator*, 880
  operator++, 880
  operator=, 880
ostreambuf_iterator, 882
  constructor, 883
ostringstream, 1010, 1069
ostrstream, 1262
  constructor, 1262
  ostrstream, 1262
out
codecvt, 711
out_of_range, 500, 536, 538, 539, 658
  out_of_range, 500
out_of_range_error, 498
outer Allocator
  scoped_allocator_adaptor, 644
output_iterator_tag, 862
overflow

basic_filebuf, 1085
basic_streambuf, 1041
basic_stringbuf, 1073
strstreambuf, 1258
overflow_error, 498, 501, 502, 536, 538, 539
  overflow_error, 501
owner_before
  shared_ptr, 573, 577
owns_lock
  shared_lock, 1185
  unique_lock, 1181

P
  bernoulli_distribution, 963
  binomial_distribution, 964
  geometric_distribution, 964
  negative_binomial_distribution, 965
packaged_task
  constructor, 1207, 1208
  destructor, 1208
  get_future, 1208
  make_ready_at_thread_exit, 1209
  operator(), 1208
  operator=, 1208
  reset, 1209
  swap, 1208, 1209
  valid, 1208
pair, 519, 527, 529
  get, 522, 523
  operator=, 520, 521
  pair, 519, 520
  swap, 521
param
  seed_seq, 959
partial_sort, 911
partial_sort_copy, 912
partial_sum, 1002
partition, 909
partition_copy, 909
partition_point, 910
pbackfail
  basic_filebuf, 1084
  basic_streambuf, 1040
  basic_stringbuf, 1072
  strstreambuf, 1259
pbase
  basic_streambuf, 1038
pbump
  basic_streambuf, 1038
pcount
  ostrstream, 1262

Cross references 1339
央　ISO/IEC

Cross references

strstrstream, 1264
strstrstreambuf, 1258
peek
  basic_istream, 1052
piecewise_constant_distribution, 976
  constructor, 977, 978
densities, 978
intervals, 978
piecewise_construct, 523
piecewise_construct_t, 523
piecewise_linear_distribution, 978
  constructor, 979, 980
densities, 980
intervals, 980
placeholders, 596
plus, 588
plus<>, 589
pointer
  allocator_traits, 549
pointer_to
  pointer_traits, 546
difference_traits, 545
difference_type, 546
element_type, 545
pointer_to, 546
rebind, 546
poisson_distribution, 965
  constructor, 966
mean, 966
polar
  complex, 934
pop
  priority_queue, 848
  forward_list, 787
pop_back
  basic_string, 672
pop_heap, 918
pos_type
  char_traits, 650
position
  match_results, 1126
positive_sign
  moneypunct, 736
pow, 936, 993, 1005
  complex, 935
pptr
  basic_stringbuf, 1038
precision
  ios_base, 703, 1021
prefix
  match_results, 1126
prev, 864
prev_permutation, 922
priority_queue, 845
  emplace, 848
  priority_queue, 846, 847
  swap, 848
probabilities
  discrete_distribution, 976
proj
  complex, 933
promise
  constructor, 1198
destructor, 1198
get_future, 1198
operator*, 1198
set_exception, 1199
set_exception_at_thread_exit, 1199
set_value, 1199
set_value_at_thread_exit, 1199
swap, 1198, 1200
propagate_on_container_copy_assignment
  allocator_traits, 550
  scoped_allocator_adaptor, 643
propagate_on_container_move_assignment
  allocator_traits, 550
  scoped_allocator_adaptor, 643
propagate_on_container_swap
  allocator_traits, 550
  scoped_allocator_adaptor, 643
proxy
  istreambuf_iterator, 881
ptrdiff_t, 468
pubimbue
  basic_stringbuf, 1035
pubseekoff
  basic_stringbuf, 1036
pubseekpos
  basic_stringbuf, 1036
pubsetbuf
  basic_stringbuf, 1036
pubsync
  basic_stringbuf, 1036
push
  priority_queue, 848
push_back
  basic_string, 669
push_front
  forward_list, 787
push_heap, 918
pub
  basic_ostream, 1063
money_put, 734  
num_put, 719  
time_put, 731  
put_money, 1066  
put_time, 1067  
putback  
  basic_istream, 1053  
putenv, 495  
pword  
  ios_base, 1022  
<queue>, 842  
queue, 843  
  swap, 845  
quick_exit, 479, 480  
quiet_NaN  
  numeric_limits, 474  
quoted, 1068  
radix  
  numeric_limits, 472  
raise, 496  
<random>, 945–947  
random_access_iterator_tag, 862  
random_device, 957  
  constructor, 957  
  entropy, 957  
  operator(), 957  
ranged_error, 498, 501  
  ranged_error, 501  
rng24, 956  
rng24_base, 956  
rng48, 956  
rng48_base, 956  
ratio, 622  
ratio_equal, 624  
ratio_greater, 625  
ratio_greater_equal, 625  
ratio_less, 625  
ratio_less_equal, 625  
ratio_not_equal, 624  
raw_storage_iterator  
  constructor, 553  
  operator*, 553  
  operator++, 553  
  operator=, 553  
rbegin  
  basic_string, 666  
rbegin(C&), 884  
rbegin(initializer_list<E>), 884  
rbegin(T (&array)[N]), 884  
rdbuf  
  basic_filebuf, 1092  
  basic_ifstream, 1088  
  basic_ios, 1026  
  basic_iostream, 1076  
  basic_ofstream, 1090  
  basic_ostream, 1077  
  basic_stringstream, 1079  
  istringstream, 1261  
  ostrstream, 1262  
  ostream, 1264  
  wbuffer_convert, 702, 703  
rdstate  
  basic_istream, 1028  
read  
  basic_istream, 1052  
readsome  
  basic_istream, 1053  
real, 935  
  complex, 930, 933  
reallocate, 555, 1252  
rebind  
  pointer_traits, 546  
rebind_allocator  
  allocator_traits, 550  
ref  
  reference_wrapper, 587  
reference_wrap, 586  
  cref, 588  
  get, 587  
  operator T&, 587  
  operator(), 587  
  operator=, 587  
  ref, 587  
  reference_wrapper, 587  
<regex>, 1098  
regex, 1098  
regex_constants, 1105  
  error_type, 1107, 1108  
  match_flag_type, 1105  
  syntax_option_type, 1105  
regex_error, 1108, 1112, 1141  
  constructor, 1108  
regex_iterator, 1133  
  increment, 1135  
  operator!=, 1135  
  operator*, 1135  
  operator++, 1135, 1136  
  operator->, 1135  
  operator==, 1135  
regex_iterator, 1134

Cross references
regex_match, 1128–1130
regex_replace, 1132, 1133
regex_search, 1130, 1131
regex_token_iterator, 1136
end-of-sequence, 1136
operator!=, 1139
operator*, 1139
operator++, 1139
operator->, 1139
operator==, 1136, 1139
regex_token_iterator, 1138
regex_traits, 1108
char_class_type, 1109
isctype, 1110
length, 1109
lookup_classname, 1110
lookup_collatename, 1109
transform, 1109
transform_primary, 1109
translate, 1109
translate_nocase, 1109
value, 1111
register_callback
ios_base, 1023
regular expression traits
isctype, 1140
lookup_classname, 1140
lookup_collatename, 1140
transform_primary, 1141
rel_ops, 514
release
shared_lock, 1184
unique_lock, 1181
unique_ptr, 562
remainder, 1005
remove, 906
list, 796
forward_list, 790
remove_copy, 906
remove_copy_if, 906
remove_if, 906
forward_list, 790
remquo, 1005
rend
basic_string, 666
rend(const C&), 884
rend(initializer_list<E>), 884
rend(T (&array)[N]), 884
rep
system_clock, 639
replace, 904
basic_string, 673, 674
replace_copy, 905
replace_copy_if, 905
replace_if, 904
reserve
basic_string, 667
vector, 801
reset
bitset, 538
packaged_task, 1209
shared_ptr, 572
unique_ptr, 562, 565
weak_ptr, 577
resetiosflags, 1064
resize
deque, 782
list, 794
basic_string, 666, 667
forward_list, 788
valarray, 990
vector, 801
rethrow_exception, 492
rethrow_if_nested
nested_exception, 494
rethrow_nested
nested_exception, 493
returnTemporaryBuffer, 553
reverse, 907
list, 797
forward_list, 790
reverse_copy, 907
reverse_iterator, 864
reverse_iterator, 866
base, 866
constructor, 866
make_reverse_iterator non-member function, 869
operator++, 866
operator--, 867
rfind
basic_string, 676
right, 1030
rint, 1005
rotate, 908
rotate_copy, 908
round, 1005
round_error
numeric_limits, 472
round_indeterminate, 475
round_style
numeric_limits, 475
Cross references
round_to_nearest, 475
round_toward_infinity, 475
round_toward_neg_infinity, 475
round_toward_zero, 475
runtime_error, 498, 500
    runtime_error, 500, 501
s
lognormal_distribution, 971
sbumpc
    basic_streambuf, 1036
scalbln, 1005
scalbn, 1005
scan_is
    ctype, 705
    ctype<char>, 708
scan_not
    ctype, 705
    ctype<char>, 709
SCHAR_MAX, 477
SCHAR_MIN, 477
scientific, 1031
<scoped_allocator>, 640
scoped_allocator_adaptor
    allocate, 644
    construct, 644, 645
    constructor, 643
deallocate, 644
destructor, 646
inner_allocator, 644
inner_allocator_type, 642
is_always_equal, 643
max_size, 644
operator!=, 646
operator==, 646
outer_allocator, 644
propagate_on_container_copy_assignment, 643
propagate_on_container_move_assignment, 643
propagate_on_container_swap, 643
select_on_container_copy_construction, 646
      ios_base, 1020
seekg
    basic_istream, 1054
seekoff
    basic_filebuf, 1085
    basic_stringbuf, 1073
    streambuf, 1259
seekp
    basic_ostream, 1059
seekpos
    basic_filebuf, 1086
    basic_streambuf, 1038
    basic_stringbuf, 1074
    streambuf, 1260
set_difference, 917
set_exception
    promise, 1199
set_exception_at_thread_exit
    promise, 1199
set_intersection, 916
set_new_handler, 462, 487
set_rdbuf
    basic_ios, 1028
set_symmetric_difference, 917
set_terminate, 462, 491
set_unexpected, 462, 1264
set_union, 916
set_value
    promise, 1199
set_value_at_thread_exit
    promise, 1199
setbase, 1065
setbuf
    basic_filebuf, 1085
    basic_streambuf, 1038, 1074
    streambuf, 1261

search, 901
search_n, 902
seed_seq
    constructor, 958
generate, 958
param, 959
size, 959
seekdir

Cross references 1343
strstreambuf, 1261
setenv, 495
setf
  ios_base, 1020, 1021
setfill, 1065
setg
  basic_streambuf, 1037
  strstreambuf, 1258
setiosflags, 1064
setjmp, 461, 496
<setjmp.h>, 495
setlocale, 446
setp
  basic_streambuf, 1038
setprecision, 1065
setstate
  basic_ios, 1028
setw, 1066
sgetc
  basic_streambuf, 1036
sgetn
  basic_streambuf, 1036
share
  future, 1201
shared_from_this
  enable_shared_from_this, 579
shared_future
  constructor, 1203
  destructor, 1203
  get, 1204
  operator=, 1203, 1204
valid, 1204
wait, 1204
wait_for, 1204
wait_until, 1205
shared_lock
  constructor, 1182, 1183
  destructor, 1183
lock, 1183
mutex, 1185
operator bool, 1185
operator=, 1183
owns_lock, 1185
release, 1184
swap, 1184, 1185
try_lock, 1184
try_lock_for, 1184
try_lock_until, 1184
unlock, 1184
<shared_mutex>, 1168
shared_ptr, 567, 579

atomic_compare_exchange_strong, 581
atomic_compare_exchange_strong_explicit, 581
atomic_compare_exchange_weak, 581
atomic_compare_exchange_weak_explicit, 581
atomic_exchange, 580
atomic_exchange_explicit, 580
atomic_is_lock_free, 580
atomic_load, 580
atomic_load_explicit, 580
atomic_store, 580
atomic_store_explicit, 580
const_pointer_cast, 575
destructor, 571
dynamic_pointer_cast, 574
get, 572
generate_deleter, 575
operator bool, 573
operator!=, 574
operator*, 572
operator->, 572
operator<, 573, 574
operator<=, 575
operator<=, 566, 574
operator>=, 571, 572
operator==, 573
operator>, 574
operator>=, 574
owner_before, 573, 577
reset, 572
shared_ptr, 569
static_pointer_cast, 574
swap, 572, 574
unique, 572
use_count, 572

shift
  valarray, 990
showbase, 1029
showmanyc
  basic_filebuf, 1084
  basic_streambuf, 1039, 1084
showpoint, 1029
showpos, 1030
shrink_to_fit
  basic_string, 667
deque, 782
vector, 801
SHRT_MAX, 477
SHRT_MIN, 477

Cross references 1344
shuffle, 908
shuffle_order_engine, 954
  constructor, 955
sig_atomic_t, 496
SIG_DFL, 496
SIG_ERR, 496
SIG_IGN, 496
SIGABRT, 496
SIGFPE, 496
SIGILL, 496
SIGINT, 496
signal, 496
<signal.h>, 495
signaling_NaN
  numeric_limits, 474
signbit, 1007
SIGSEGV, 496
SIGTERM, 496
sin, 993, 1005
  complex, 935
sinh, 993, 1005
  complex, 935
size
  array, 777, 778
  basic_string, 666
bitset, 539
gslice, 997
initializer_list, 495
match_results, 1125
seed_seq, 959
slice, 994
size_t, 115, 468
size_type
  allocator_traits, 549
skipws, 1030
sleep_for
  this_thread, 1167
sleep_until
  this_thread, 1167
slice, 993
  slice, 993
slice_array, 994
snexec
  basic_streambuf, 1036
sort, 911
  list, 797
    forward_list, 790
sort_heap, 919
splice
  list, 795
  list, 796
splice_after
  forward_list, 789
sputch
  basic_streambuf, 1036
sputc
  basic_streambuf, 1037
sputn
  basic_streambuf, 1037
sqrt, 993, 1005
  complex, 935
<string.h>, 1069
<staarg.h>, 495
stable_partition, 909
stable_sort, 911
<stack>, 848
stack, 848
  swap, 851
start
  gslice, 997
  slice, 994
state
  fpos, 1023
  match_results, 1125
  wbuffer_convert, 702
  wstring_convert, 701
state_type
  char_traits, 650
  wbuffer_convert, 703
  wstring_convert, 701
static_pointer_cast
  shared_ptr, 574
<stalign.h>, 495
<stdarg.h>, 495
<stdbool.h>, 495
<cstdlib.h>, 498
<stdexcept.h>, 498
<stdlib.h>, 495, 1253
stod, 685
stof, 684, 685
stoi, 684, 685
stol, 684, 685
stold, 684, 685
stoll, 684, 685
store
  atomic type, 1153
stoul, 684, 685
stoull, 684, 685
str
  basic_istringstream, 1076
  basic_ostringstream, 1077
basic_stringbuf, 1072
basic_stringstream, 1079
istream, 1261
match_results, 1126
ostream, 1262
stringstream, 1264
strstreambuf, 1258
sub_match, 1118
strchr, 686
<string>, 1032
streambuf, 1010, 1032
streamoff, 1015, 1024, 1254
streamsize, 1015
ios_base, 1021
strftime, 731
stride
gelice, 997
slice, 994
<string>, 654
stringbuf, 1010, 1069
stringstream, 1010
strlen, 1258, 1262
strpbrk, 686
strrchr, 687
strstr, 687
stringstream, 1263
destructor, 1263
stringstream, 1263
stringstreambuf, 1255, 1257
stringstreambuf, 1257
destructor, 1258
setg, 1258
student_t_distribution, 974
constructor, 975
mean, 975
sub_match, 1117
compare, 1118
constructor, 1117
length, 1117
operator basic_string, 1117
operator!=, 1118–1122
operator<, 1118–1122
operator<, 1118–1122
operator<, 1118–1122
operator>, 1118–1122
operator>>, 1118–1122
str, 1118
substr
basic_string, 678
subtract_with_carry_engine, 951
constructor, 951, 952
suffix
match_results, 1126
sum
valarray, 990
sungetc
basic_stringbuf, 1036
swap, 517, 533
pair, 522
array, 778
basic_filebuf, 1082
basic_fstream, 1092
basic_ofstream, 1088
basic_ios, 1028
basic_iostream, 1055
basic_istream, 1045
basic_istringstream, 1075
basic_ofstream, 1090
basic_ostream, 1087
basic_ostringstream, 1077
basic_regex, 1117
basic_stringbuf, 1037
basic_string, 675, 683
basic_stringbuf, 1071, 1072
basic_stringstream, 1079
deque, 783
forward_list, 791
function, 601, 602
list, 797
map, 812
match_results, 1127
multimap, 816
multiset, 823
packaged_task, 1208, 1209
pair, 521
priority_queue, 848
promise, 1198, 1200
queue, 845
set, 820
shared_lock, 1184, 1185
shared_ptr, 572, 574
stack, 851
thread, 1165, 1166
tuple, 529
unique_lock, 1181
unique_ptr, 562
unordered_map, 830
unordered_multimap, 834
unordered_multiset, 842
unordered_set, 838
valarray, 989, 993

Cross references 1346
vector, 801, 802  
vector<bool>, 805  
weak_ptr, 577, 578  
swap(unique_ptr&, unique_ptr&), 565  
swap_ranges, 903  
sync  
  basic_filebuf, 1086  
  basic_istream, 1053  
  basic_streambuf, 1038  
sync_with_stdio  
ios_base, 1021  
syntax_option_type, 1105  
awk, 1105  
basic, 1105  
collate, 1105, 1141  
ECMAScript, 1105  
egrep, 1105  
extended, 1105  
grep, 1105  
icase, 1105  
nosubs, 1105  
optimize, 1105  
syntax_option_type  
awk, 1106  
basic, 1106  
collate, 1106  
ECMAScript, 1106  
egrep, 1106  
extended, 1106  
grep, 1106  
icase, 1106  
nosubs, 1106  
optimize, 1106  
system, 495, 496  
system_category, 505, 507  
system_clock  
rep, 639  
system_error, 502, 512  
  code, 513  
  system_error, 512, 513  
  what, 513  
t  
binomial_distribution, 964  
negative_binomial_distribution, 965  
table  
  ctype<char>, 709  
tan, 993, 1005  
  complex, 935  
tanh, 993, 1005  
  complex, 935  

Cross references
do_get_date, 729
do_get_monthname, 729
do_get_time, 728
do_get_weekday, 729
do_get_year, 729
get, 728
get_date, 727
get_monthname, 727
get_time, 727
get_weekday, 727
get_year, 727
time_get_byname, 730
time_point
  constructor, 637
  max, 638
  min, 638
  operator!=, 638
  operator+, 638
  operator==, 637
  operator-, 638
  operator=, 637
  operator<, 638
  operator<=, 638
  operator==, 638
  operator>, 639
  operator>=, 639
  time_since_epoch, 637
time_point_cast, 639
time_put, 730
  do_put, 731
  put, 731
time_put_byname, 731
time_since_epoch
tinyess_before
  numeric_limits, 475
to_bytes
  wstring_convert, 701
to_string, 685
  bitset, 539
to_time_t, 639
to_ullong
  bitset, 538
to_ulong
  bitset, 538
to_wstring, 685
tolower, 699
cctype<char>, 705
toupper, 699
cctype, 705
cctype<char>, 709
transform, 904
collate, 725
regex_traits, 1109
transform_primary
  regex_traits, 1109
translate
  regex_traits, 1109
translate_nocase
  regex_traits, 1109
traps
  numeric_limits, 474
treat_as_floating_point, 629
true
  numpunct, 723
trunc, 1005
try_emplace
  map, 812
unordered_map, 830
try_lock, 1185
  shared_lock, 1184
  unique_lock, 1180
try_lock_for
  shared_lock, 1184
  unique_lock, 1180
try_lock_until
  shared_lock, 1184
  unique_lock, 1180
try_to_lock, 1176
try_to_lock_t, 1176
<tuple>, 523
tuple, 523, 525, 779
  constructor, 526, 527
  forward_as_tuple, 530
  get, 531, 532
  make_tuple, 529
  operator!=, 532
  operator<, 532
  operator<=, 533
  operator==, 528, 529
  operator>, 533
  operator>=, 533
  swap, 529
tie, 530
tuple, 526
tuple_cat, 530
tuple_element, 522, 531, 779
tuple_size, 522, 530, 779
  in general, 530
type_index
constructor, 647
hash_code, 647
name, 647
operator!=, 647
operator<, 647
operator<=, 647
operator==, 647
operator>, 647
operator>=, 647
type_info, 108, 487
type_info::name
implementation-defined, 488
<typeinfo>, 487, 646

UCHAR_MAX, 477
uflow
  basic_filebuf, 1084
  basic_streambuf, 1040
uint16_t, 478
uint32_t, 478
uint64_t, 478
uint8_t, 478
uint_fast16_t, 478
uint_fast32_t, 478
uint_fast64_t, 478
uint_fast8_t, 478
uint_least16_t, 478
uint_least32_t, 478
uint_least64_t, 478
uint_least8_t, 478
UINT_MAX, 477
uintptr_t, 478
ULLONG_MAX, 477
unary_negate, 594
uncaught_exception, 1264
uncaught_exceptions, 425, 492
undeclare_no_pointers, 547
undeclarereachable, 546
underflow
  basic_filebuf, 1084
  basic_streambuf, 1039
  basic_stringbuf, 1072
  strstreambuf, 1259
underflow_error, 498
  underflow_error, 502
unexpected, 1264
unexpected_handler, 462, 1264
unget
  basic_istream, 1053
uniform_int_distribution, 960
  a, 961
  b, 961
  constructor, 961
uniform_real_distribution, 961
  a, 962
  b, 962
  constructor, 962
uninitialized_copy, 554
uninitialized_copy_n, 554
uninitialized_fill, 554
uninitialized_fill_n, 555
unique, 906
  list, 796
  forward_list, 790
  shared_ptr, 572
unique_copy, 907
unique_lock
  constructor, 1178, 1179
  destructor, 1179
  lock, 1180
  mutex, 1181
  operator bool, 1181
  operator=, 1179
  owns_lock, 1181
  release, 1181
  swap, 1181
  try_lock, 1180
  try_lock_for, 1180
  try_lock_until, 1180
  unlock, 1180
unique_ptr, 571
  constructor, 560, 561, 564
  destructor, 561
  get, 562
  get_deleter, 562
  operator bool, 562
  operator=, 565, 566
  operator*, 562
  operator->, 562
  operator<, 565, 566
  operator==, 566
  operator>=, 566
  operator[], 564
  release, 562
  reset, 562, 565
  swap, 562
  unique_ptr, 559
unitbuf, 1030

Cross references 1349
unlock
  shared_lock, 1184
unique_lock, 1180
 unordered_map, 824, 825
   at, 829
   insert, 830
   insert_or_assign, 830
   operator[], 829
   swap, 830
   try_emplace, 830
 unordered_map, 829
 unordered_multimap, 824, 831
   insert, 834
   swap, 834
   unordered_multimap, 834
 unordered_multiset, 824, 838, 839
   swap, 842
   unordered_multiset, 841
 unordered_set, 824, 835
   swap, 838
   unordered_set, 838
 unsetf
   ios_base, 1021
 unshift
   codecvt, 711
 upper_bound, 913
 uppERCASE, 1030
 use_count
   shared_ptr, 572
   weak_ptr, 577
 use_facet
   locale, 698
 uses_allocator, 548, 1198, 1209
 uses_allocator<tuple>, 533
 USHRT_MAX, 477
<utility>, 514
 va_arg, 496
 va_copy, 496
 va_end, 461, 496
 va_list, 461, 496
 va_start, 495, 496
 <valarray>, 980
 valarray, 983, 997
   begin, 1001
   constructor, 985
   destructor, 985
   end, 1001
   operator!=, 992

operator*, 991
operator**, 989
operator+, 991
operator+=, 989
operator-=, 989
operator/, 991
operator/=, 989
operator<, 992
operator<=, 992
operator<, 991
operator<<, 989
operator<<=, 989
operator==, 986, 991
operator==, 992
operator>, 992
operator>=, 992
operator>>, 991
operator>>, 991
operator%?, 991
operator%?, 991
operator%?, 991
operator&?, 991
operator&?, 991
operator&&, 992
operator&&, 992
operator^?, 991
operator^?, 991
operator^?, 991
operator|?, 991
operator|?, 991
operator||, 992
operator||, 992
operator&, 989
operator&?, 989
operator&&, 992
operator&&, 992
swap, 989, 993
swap, 989, 993
valarray, 985

valid
 future, 1201
 packaged_task, 1208
 shared_future, 1204

value
   error_code, 509
   error_condition, 511
   regex_traits, 1111
<vector>, 776
 vector, 797
   operator<, 800
   operator<=, 800
   vector, 800
   swap, 802
 vector<bool>, 803
   flip, 805
   swap, 805

void_pointer
   allocator_traits, 549

wait
   condition_variable, 1189
condition_variable_any, 1192, 1193
future, 1202
shared_future, 1204
wait_for
condition_variable, 1190, 1191
collection_variable_any, 1193, 1194
future, 1202
shared_future, 1204
wait_until
condition_variable, 1189, 1190
collection_variable_any, 1193
future, 1202
shared_future, 1205
wbuffer_convert, 702
collection, 703
destructor, 703
rdbuf, 702, 703
state, 702
state_type, 703
wcerr, 1014
wcin, 1014
wcout, 1014
wcschr, 687
wcschr, 687
wcschr, 687
wcschr, 687
weak_ptr, 570, 575
constructor, 576
destructor, 577
expired, 577
lock, 577
operator=, 577
reset, 577
swap, 577, 578
use_count, 577
weibull_distribution, 968
a, 969
b, 969
constructor, 969
wfilebuf, 1010, 1079
wfstream, 1010
what
bad_alloc, 486
bad_cast, 489
bad_exception, 491
bad_typeid, 489
exception, 490
bad_weak_ptr, 567
future_error, 1196
system_error, 513
wide_string
wstring_convert, 701
widen
basic_ios, 1027
ctype, 705
ctype<char>, 709
width
ios_base, 703, 1021
wifstream, 1010, 1079
wios, 1015
wistream, 1010, 1042
wstringstream, 1010, 1069
wmemchr, 687
wifstream, 1010, 1079
wostream, 1010, 1042
wstringstream, 1010, 1069
wregex, 1098
write
basic_ostream, 1063
ws, 1048, 1054
wstringstream, 1010, 1032
wstring_convert, 699
byte_string, 700
constructor, 701
converted, 700
destructor, 701
from_bytes, 700
int_type, 701
state, 701
state_type, 701
to_bytes, 701
wide_string, 701
wstringstream, 1010
xalloc
ios_base, 1022
xsgetn
basic_streambuf, 1039
xsputn
basic_streambuf, 1041
yield
this_thread, 1167
zero
duration, 633
duration_values, 629
Index of implementation-defined behavior

The entries in this section are rough descriptions; exact specifications are at the indicated page in the general text.

#pragma, 436

additional formats for time_get::do_get_date, 729
alignment, 80
alignment additional values, 80
alignment of bit-fields within a class object, 240
allocation of bit-fields within a class object, 240
argument values to construct basic_ios::failure, 1028
assignability of placeholder objects, 597
behavior of attribute scoped token, 186
behavior of iostream classes when traits::pos_type is not streampos or when traits::off_type is not streamoff, 1010
behavior of non-standard attributes, 186
bits in a byte, 6
choice of larger or smaller value of floating literal, 28
concatenation of some types of string literals, 30
conversions between pointers and integers, 111
converting characters from source character set to execution character set, 18
converting function pointer to object pointer and vice versa, 111
default number of buckets in unordered_map, 829
default number of buckets in unordered_multimap, 834
default number of buckets in unordered_multiset, 841, 842
default number of buckets in unordered_set, 838
defining main in freestanding environment, 62
definition and meaning of __STDC__, 436
definition and meaning of __STDC_VERSION__, 436
derived type for typeid, 108
diagnostic message, 2
dynamic initialization of static objects before main, 64
dynamic initialization of thread-local objects before entry, 64
effect of calling basic_filebuf::setbuf with non-zero arguments, 1085
effect of calling basic_filebuf::sync when a get area exists, 1086
effect of calling basic_streambuf::setbuf with non-zero arguments, 1074
effect of calling ios_base::sync_with_stdio after any input or output operation on standard streams, 1021
effect on C locale of calling locale::global, 698
encoding of universal character name not in execution character set, 27
error_category for errors originating outside the operating system, 467
exception type when shared_ptr constructor fails, 570
exceptions thrown by standard library functions that do not have an exception specification, 467
execution character-set and execution wide-character set, 19
exit status, 480
extended signed integer types, 75
formatted character sequence generated by time_put::do_put in C locale, 731
headers for freestanding implementation, 449
interactive device, 8
linkage of main, 62
linkage of names from Standard C library, 450
locale names, 696
manner of search for included source file, 429
mapping from name to catalog when calling messages::do_open, 738
mapping header name to header or external source file, 21
mapping physical source file characters to basic source character set, 17
mapping to message when calling messages::do_get, 739

Cross references 1352
meaning of `asm` declaration, 182
meaning of attribute declaration, 148
negative value of character literal in preprocessor, 428
nesting limit for `#include` directives, 429
number of threads in a program under a freestanding implementation, 11
numeric values of character literals in `#if` directives, 428
parameters to `main`, 62
passing argument of class type through ellipsis, 104
physical source file characters, 17
presence and meaning of `native_handle_type` and `native_handle`, 1159
rank of extended signed integer type, 87
required libraries for freestanding implementation, 5
result of `exception::what`, 490
result of inexact floating-point conversion, 85
result of right shift of negative value, 126
return value of `bad_alloc::what`, 486
return value of `bad_cast::what`, 489
return value of `bad_exception::what`, 491
return value of `bad_typeid::what`, 489
return value of `char_traits<char16_t>::eof`, 652
return value of `char_traits<char32_t>::eof`, 653
return value of `type_info::name()`, 488
search locations for "" header, 429
search locations for <> header, 429
semantics of linkage specification on templates, 334
semantics of linkage specifiers, 182
semantics of non-standard escape sequences, 27
sequence of places searched for a header, 429
signedness of `char`, 75
`sizeof` applied to fundamental types other than `char`, `signed char`, and `unsigned char`, 115
stack unwinding before call to `std::terminate()`, 418, 425
start-up and termination in freestanding environment, 62
string resulting from `__func__`, 208
string returned by `what()` for `bad_weak_ptr`, 567, 597

Cross references 1353

support for extended alignment, 622
support for over-aligned types, 115, 552, 553
supported multibyte character encoding rules, 651, 654
text of `_DATE_` when date of translation is not available, 436
text of `_TIME_` when time of translation is not available, 436
type of `ios_base::streamoff`, 1254
type of `ios_base::streampos`, 1254
type of `std::ptrdiff_t`, 125, 469
type of `regex_constants::error_type`, 1107
type of `size_t`, 469
type of `streamoff`, 651
type of `streampos`, 651
type of `std::u16streampos`, 652
type of `std::u32streampos`, 653
type of `std::wstreampos`, 653
type of `array::const_iterator`, 777
type of `array::iterator`, 777

underlying type for enumeration, 168
use of non-POF function as signal handler, 496

value of `ctype<char>::table_size`, 708
value of bit-field that cannot represent
assigned value, 131
incremented value, 106
initializer, 214
value of character literal outside range of corresponding type, 27
value of multicharacter literal, 26
value of result of inexact integer to floating-point conversion, 86
value of result of unsigned to signed conversion, 85
value of wide-character literal containing multiple characters, 27
value of wide-character literal with single c-char that is not in execution wide-character set, 27
value representation of floating-point types, 76
value representation of pointer types, 77
values of a trivially copyable type, 73
values of various `ATOMIC_..._LOCK_FREE` macros, 1147

whether `get_pointer_safety` returns `pointer_safety::relaxed` or `pointer_safety::preferred` if the implementation has relaxed pointer safety, 547
whether `time_get::do_get_year` accepts two-digit year numbers, 729
whether an implementation has relaxed or strict pointer safety, 69
whether locale object is global or per-thread, 693
whether sequence pointers are copied by `basic_filebuf` move constructor, 1082
whether sequence pointers are copied by `basic_stringbuf` move constructor, 1071
whether source of translation units must be available to locate template definitions, 18
whether stack is unwound before calling `std::terminate()` when a `noexcept` specification is violated, 425
whether values are rounded or truncated to the required precision when converting between `time_t` values and `time_point` objects, 639
which functions in Standard C++ library may be recursively reentered, 465