Working Draft, Standard for Programming Language C++

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

1 General 1
   1.1 Scope ................................................................. 1
   1.2 Normative references .............................................. 1
   1.3 Definitions .......................................................... 2
   1.4 Implementation compliance ........................................ 4
   1.5 Structure of this International Standard ......................... 5
   1.6 Syntax notation ..................................................... 5
   1.7 The C++ memory model .............................................. 6
   1.8 The C++ object model .............................................. 6
   1.9 Program execution .................................................. 7
   1.10 Acknowledgments ................................................... 10

2 Lexical conventions 11
   2.1 Phases of translation ............................................... 11
   2.2 Character sets ....................................................... 12
   2.3 Trigraph sequences .................................................. 13
   2.4 Preprocessing tokens ................................................. 14
   2.5 Alternative tokens ................................................... 14
   2.6 Tokens ................................................................. 15
   2.7 Comments .............................................................. 15
   2.8 Header names .......................................................... 15
   2.9 Preprocessing numbers .............................................. 16
   2.10 Identifiers .............................................................. 16
   2.11 Keywords ............................................................. 17
   2.12 Operators and punctuators ......................................... 17
   2.13 Literals ............................................................... 18

3 Basic concepts 25
   3.1 Declarations and definitions ....................................... 25
   3.2 One definition rule .................................................. 27
   3.3 Declarative regions and scopes ..................................... 29
   3.4 Name lookup ........................................................... 35
   3.5 Program and linkage ................................................ 49
   3.6 Start and termination ................................................. 52
## Contents

3.7 Storage duration .................................................. 54
3.8 Object Lifetime .................................................. 57
3.9 Types ............................................................. 61
3.10 Lvalues and rvalues .............................................. 66
3.11 Alignment .......................................................... 68

4 Standard conversions ............................................. 69
  4.1 Lvalue-to-rvalue conversion ................................... 70
  4.2 Array-to-pointer conversion .................................... 70
  4.3 Function-to-pointer conversion ................................ 70
  4.4 Qualification conversions ....................................... 70
  4.5 Integral promotions ............................................ 72
  4.6 Floating point promotion ....................................... 72
  4.7 Integral conversions ........................................... 72
  4.8 Floating point conversions ..................................... 72
  4.9 Floating-integral conversions .................................. 73
  4.10 Pointer conversions ........................................... 73
  4.11 Pointer to member conversions ................................ 73
  4.12 Boolean conversions .......................................... 74
  4.13 Integer conversion rank ........................................ 74

5 Expressions ......................................................... 75
  5.1 Primary expressions ............................................ 77
  5.2 Postfix expressions ............................................. 79
  5.3 Unary expressions ............................................... 90
  5.4 Explicit type conversion (cast notation) ....................... 98
  5.5 Pointer-to-member operators .................................... 99
  5.6 Multiplicative operators ....................................... 100
  5.7 Additive operators ............................................. 100
  5.8 Shift operators ................................................ 102
  5.9 Relational operators ........................................... 102
  5.10 Equality operators ............................................ 103
  5.11 Bitwise AND operator ......................................... 104
  5.12 Bitwise exclusive OR operator ................................. 104
  5.13 Bitwise inclusive OR operator ................................ 104
  5.14 Logical AND operator .......................................... 105
  5.15 Logical OR operator ........................................... 105
  5.16 Conditional operator .......................................... 105
  5.17 Assignment and compound assignment operators ............ 107
  5.18 Comma operator ............................................... 107
  5.19 Constant expressions .......................................... 108

6 Statements .......................................................... 111
  6.1 Labeled statement ............................................... 111
  6.2 Expression statement ........................................... 111
  6.3 Compound statement or block .................................... 111

Draft
6.4 Selection statements .................................................. 112
6.5 Iteration statements ................................................... 114
6.6 Jump statements ....................................................... 116
6.7 Declaration statement ................................................ 117
6.8 Ambiguity resolution .................................................. 118

7 Declarations .................................................................. 121
7.1 Specifiers ................................................................. 122
7.2 Enumeration declarations ............................................. 137
7.3 Namespaces .............................................................. 141
7.4 The asm declaration ..................................................... 154
7.5 Linkage specifications ................................................... 154

8 Declarators .................................................................... 159
8.1 Type names ............................................................... 160
8.2 Ambiguity resolution ................................................... 161
8.3 Meaning of declarators ............................................... 162
8.4 Function definitions .................................................... 175
8.5 Initializers ................................................................. 177

9 Classes .......................................................................... 189
9.1 Class names .............................................................. 191
9.2 Class members .......................................................... 193
9.3 Member functions ..................................................... 195
9.4 Static members .......................................................... 199
9.5 Unions ...................................................................... 201
9.6 Bit-fields ................................................................. 202
9.7 Nested class declarations .............................................. 203
9.8 Local class declarations .............................................. 204
9.9 Nested type names ..................................................... 205

10 Derived classes ........................................................... 207
10.1 Multiple base classes .................................................. 208
10.2 Member name lookup ................................................ 210
10.3 Virtual functions ....................................................... 214
10.4 Abstract classes ........................................................ 218

11 Member access control ................................................... 221
11.1 Access specifiers ....................................................... 223
11.2 Accessibility of base classes and base class members .......... 224
11.3 Access declarations .................................................... 227
11.4 Friends ................................................................. 228
11.5 Protected member access .......................................... 231
11.6 Access to virtual functions ........................................... 232
11.7 Multiple access ........................................................ 233
11.8 Nested classes ........................................................ 233

Draft
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Special member functions</td>
<td></td>
</tr>
<tr>
<td>12.1</td>
<td>Constructors</td>
<td>235</td>
</tr>
<tr>
<td>12.2</td>
<td>Temporary objects</td>
<td>237</td>
</tr>
<tr>
<td>12.3</td>
<td>Conversions</td>
<td>239</td>
</tr>
<tr>
<td>12.4</td>
<td>Destructors</td>
<td>242</td>
</tr>
<tr>
<td>12.5</td>
<td>Free store</td>
<td>245</td>
</tr>
<tr>
<td>12.6</td>
<td>Initialization</td>
<td>247</td>
</tr>
<tr>
<td>12.7</td>
<td>Construction and destruction</td>
<td>253</td>
</tr>
<tr>
<td>12.8</td>
<td>Copying class objects</td>
<td>256</td>
</tr>
<tr>
<td>13</td>
<td>Overloading</td>
<td>263</td>
</tr>
<tr>
<td>13.1</td>
<td>Overloadable declarations</td>
<td>263</td>
</tr>
<tr>
<td>13.2</td>
<td>Declaration matching</td>
<td>266</td>
</tr>
<tr>
<td>13.3</td>
<td>Overload resolution</td>
<td>267</td>
</tr>
<tr>
<td>13.4</td>
<td>Address of overloaded function</td>
<td>284</td>
</tr>
<tr>
<td>13.5</td>
<td>Overloaded operators</td>
<td>285</td>
</tr>
<tr>
<td>13.6</td>
<td>Built-in operators</td>
<td>289</td>
</tr>
<tr>
<td>14</td>
<td>Templates</td>
<td>293</td>
</tr>
<tr>
<td>14.1</td>
<td>Template parameters</td>
<td>294</td>
</tr>
<tr>
<td>14.2</td>
<td>Names of template specializations</td>
<td>298</td>
</tr>
<tr>
<td>14.3</td>
<td>Template arguments</td>
<td>299</td>
</tr>
<tr>
<td>14.4</td>
<td>Type equivalence</td>
<td>306</td>
</tr>
<tr>
<td>14.5</td>
<td>Template declarations</td>
<td>307</td>
</tr>
<tr>
<td>14.7</td>
<td>Template instantiation and specialization</td>
<td>337</td>
</tr>
<tr>
<td>14.8</td>
<td>Function template specializations</td>
<td>350</td>
</tr>
<tr>
<td>15</td>
<td>Exception handling</td>
<td>371</td>
</tr>
<tr>
<td>15.1</td>
<td>Throwing an exception</td>
<td>372</td>
</tr>
<tr>
<td>15.2</td>
<td>Constructors and destructors</td>
<td>374</td>
</tr>
<tr>
<td>15.3</td>
<td>Handling an exception</td>
<td>375</td>
</tr>
<tr>
<td>15.4</td>
<td>Exception specifications</td>
<td>377</td>
</tr>
<tr>
<td>15.5</td>
<td>Special functions</td>
<td>379</td>
</tr>
<tr>
<td>15.6</td>
<td>Exceptions and access</td>
<td>381</td>
</tr>
<tr>
<td>16</td>
<td>Preprocessing directives</td>
<td>383</td>
</tr>
<tr>
<td>16.1</td>
<td>Conditional inclusion</td>
<td>385</td>
</tr>
<tr>
<td>16.2</td>
<td>Source file inclusion</td>
<td>386</td>
</tr>
<tr>
<td>16.3</td>
<td>Macro replacement</td>
<td>387</td>
</tr>
<tr>
<td>16.4</td>
<td>Line control</td>
<td>393</td>
</tr>
<tr>
<td>16.5</td>
<td>Error directive</td>
<td>393</td>
</tr>
<tr>
<td>16.6</td>
<td>Pragma directive</td>
<td>393</td>
</tr>
<tr>
<td>16.7</td>
<td>Null directive</td>
<td>394</td>
</tr>
<tr>
<td>16.8</td>
<td>Predefined macro names</td>
<td>394</td>
</tr>
<tr>
<td>16.9</td>
<td>Pragma operator</td>
<td>395</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
<td>Section</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>17</td>
<td>Library introduction</td>
<td>17.1 Definitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17.2 Additional definitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17.3 Method of description (Informative)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17.4 Library-wide requirements</td>
</tr>
<tr>
<td>18</td>
<td>Language support library</td>
<td>18.1 Types</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18.2 Implementation properties</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18.3 Integer types</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18.4 Start and termination</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18.5 Dynamic memory management</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18.6 Type identification</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18.7 Exception handling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18.8 Other runtime support</td>
</tr>
<tr>
<td>19</td>
<td>Diagnostics library</td>
<td>19.1 Exception classes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19.2 Assertions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19.3 Error numbers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19.4 System error support</td>
</tr>
<tr>
<td>20</td>
<td>General utilities library</td>
<td>20.1 Requirements</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20.2 Utility components</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20.3 Tuples</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20.4 Metaprogramming and type traits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20.5 Function objects</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20.6 Memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20.7 Date and time</td>
</tr>
<tr>
<td>21</td>
<td>Strings library</td>
<td>21.1 Character traits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>21.2 String classes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>21.3 Class template basic_string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>21.4 Null-terminated sequence utilities</td>
</tr>
<tr>
<td>22</td>
<td>Localization library</td>
<td>22.1 Locales</td>
</tr>
<tr>
<td></td>
<td></td>
<td>22.2 Standard locale categories</td>
</tr>
<tr>
<td></td>
<td></td>
<td>22.3 C Library Locales</td>
</tr>
<tr>
<td>23</td>
<td>Containers library</td>
<td>23.1 Container requirements</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23.2 Sequence containers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23.3 Associative containers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23.4 Unordered associative containers</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>24</td>
<td>Iterators library</td>
<td>743</td>
</tr>
<tr>
<td>24.1</td>
<td>Iterator requirements</td>
<td>743</td>
</tr>
<tr>
<td>24.2</td>
<td>Header <code>&lt;iterator&gt;</code> synopsis</td>
<td>748</td>
</tr>
<tr>
<td>24.3</td>
<td>Iterator primitives</td>
<td>751</td>
</tr>
<tr>
<td>24.4</td>
<td>Predefined iterators</td>
<td>755</td>
</tr>
<tr>
<td>24.5</td>
<td>Stream iterators</td>
<td>771</td>
</tr>
<tr>
<td>25</td>
<td>Algorithms library</td>
<td>781</td>
</tr>
<tr>
<td>25.1</td>
<td>Non-modifying sequence operations</td>
<td>791</td>
</tr>
<tr>
<td>25.2</td>
<td>Mutating sequence operations</td>
<td>795</td>
</tr>
<tr>
<td>25.3</td>
<td>Sorting and related operations</td>
<td>803</td>
</tr>
<tr>
<td>25.4</td>
<td>C library algorithms</td>
<td>817</td>
</tr>
<tr>
<td>26</td>
<td>Numerics library</td>
<td>819</td>
</tr>
<tr>
<td>26.1</td>
<td>Numeric type requirements</td>
<td>819</td>
</tr>
<tr>
<td>26.2</td>
<td>The floating-point environment</td>
<td>820</td>
</tr>
<tr>
<td>26.3</td>
<td>Complex numbers</td>
<td>821</td>
</tr>
<tr>
<td>26.4</td>
<td>Random number generation</td>
<td>831</td>
</tr>
<tr>
<td>26.5</td>
<td>Numeric arrays</td>
<td>876</td>
</tr>
<tr>
<td>26.6</td>
<td>Generalized numeric operations</td>
<td>898</td>
</tr>
<tr>
<td>26.7</td>
<td>C Library</td>
<td>901</td>
</tr>
<tr>
<td>27</td>
<td>Input/output library</td>
<td>907</td>
</tr>
<tr>
<td>27.1</td>
<td>Iostreams requirements</td>
<td>907</td>
</tr>
<tr>
<td>27.2</td>
<td>Forward declarations</td>
<td>908</td>
</tr>
<tr>
<td>27.3</td>
<td>Standard iostream objects</td>
<td>910</td>
</tr>
<tr>
<td>27.4</td>
<td>Iostreams base classes</td>
<td>912</td>
</tr>
<tr>
<td>27.5</td>
<td>Stream buffers</td>
<td>931</td>
</tr>
<tr>
<td>27.6</td>
<td>Formatting and manipulators</td>
<td>942</td>
</tr>
<tr>
<td>27.7</td>
<td>String-based streams</td>
<td>971</td>
</tr>
<tr>
<td>27.8</td>
<td>File-based streams</td>
<td>985</td>
</tr>
<tr>
<td>28</td>
<td>Regular expressions library</td>
<td>1003</td>
</tr>
<tr>
<td>28.1</td>
<td>Definitions</td>
<td>1003</td>
</tr>
<tr>
<td>28.2</td>
<td>Requirements</td>
<td>1004</td>
</tr>
<tr>
<td>28.3</td>
<td>Regular expressions summary</td>
<td>1006</td>
</tr>
<tr>
<td>28.4</td>
<td>Header <code>&lt;regex&gt;</code> synopsis</td>
<td>1006</td>
</tr>
<tr>
<td>28.5</td>
<td>Namespace <code>std::regex_constants</code></td>
<td>1012</td>
</tr>
<tr>
<td>28.6</td>
<td>Class <code>regex_error</code></td>
<td>1016</td>
</tr>
<tr>
<td>28.7</td>
<td>Class template <code>regex_traits</code></td>
<td>1017</td>
</tr>
<tr>
<td>28.8</td>
<td>Class template <code>basic_regex</code></td>
<td>1019</td>
</tr>
<tr>
<td>28.9</td>
<td>Class template <code>sub_match</code></td>
<td>1024</td>
</tr>
<tr>
<td>28.10</td>
<td>Class template <code>match_results</code></td>
<td>1030</td>
</tr>
<tr>
<td>28.11</td>
<td>Regular expression algorithms</td>
<td>1034</td>
</tr>
<tr>
<td>28.12</td>
<td>Regular expression Iterators</td>
<td>1039</td>
</tr>
<tr>
<td>28.13</td>
<td>Modified ECMAScript regular expression grammar</td>
<td>1045</td>
</tr>
</tbody>
</table>
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 Standard C library  

D  Compatibility features
   D.1 Increment operator with bool operand  
   D.2 static keyword  
   D.3 Access declarations  
   D.4 Implicit conversion from const strings  
   D.5 Standard C library headers  
   D.6 Old iostreams members  
   D.7 char* streams  
   D.8 Binders  
   D.9 auto_ptr  

E  Universal-character-names

F  Cross references
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>trigraph sequences</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>alternative tokens</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>keywords</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>alternative representations</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>Types of Integer Constants</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>escape sequences</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>string literal concatenations</td>
<td>23</td>
</tr>
<tr>
<td>8</td>
<td>relations on <code>const</code> and <code>volatile</code></td>
<td>66</td>
</tr>
<tr>
<td>9</td>
<td><code>simple-type-specifier</code> s and the types they specify</td>
<td>133</td>
</tr>
<tr>
<td>10</td>
<td>relationship between operator and function call notation</td>
<td>272</td>
</tr>
<tr>
<td>11</td>
<td>conversions</td>
<td>280</td>
</tr>
<tr>
<td>12</td>
<td>Library Categories</td>
<td>401</td>
</tr>
<tr>
<td>13</td>
<td>C++ Library Headers</td>
<td>408</td>
</tr>
<tr>
<td>14</td>
<td>C++ Headers for C Library Facilities</td>
<td>408</td>
</tr>
<tr>
<td>15</td>
<td>C++ Headers for Freestanding Implementations</td>
<td>409</td>
</tr>
<tr>
<td>16</td>
<td>Language support library summary</td>
<td>417</td>
</tr>
<tr>
<td>17</td>
<td>Header <code>&lt;stddef&gt;</code> synopsis</td>
<td>418</td>
</tr>
<tr>
<td>18</td>
<td>Header <code>&lt;limits&gt;</code> synopsis</td>
<td>427</td>
</tr>
<tr>
<td>19</td>
<td>Header <code>&lt;cfloat&gt;</code> synopsis</td>
<td>428</td>
</tr>
<tr>
<td>20</td>
<td>Header <code>&lt;stdlib&gt;</code> synopsis</td>
<td>429</td>
</tr>
<tr>
<td>21</td>
<td>Header <code>&lt;stdarg&gt;</code> synopsis</td>
<td>443</td>
</tr>
<tr>
<td>22</td>
<td>Header <code>&lt;csetjmp&gt;</code> synopsis</td>
<td>443</td>
</tr>
<tr>
<td>23</td>
<td>Header <code>&lt;time&gt;</code> synopsis</td>
<td>443</td>
</tr>
<tr>
<td>24</td>
<td>Header <code>&lt;signal&gt;</code> synopsis</td>
<td>444</td>
</tr>
<tr>
<td>25</td>
<td>Header <code>&lt;stdlib&gt;</code> synopsis</td>
<td>444</td>
</tr>
<tr>
<td>26</td>
<td>Header <code>&lt;stdbool&gt;</code> synopsis</td>
<td>444</td>
</tr>
<tr>
<td>27</td>
<td>Diagnostics library summary</td>
<td>445</td>
</tr>
<tr>
<td>28</td>
<td>Header <code>&lt;cassert&gt;</code> synopsis</td>
<td>450</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>29</td>
<td>Header <code>&lt;cerrno&gt;</code> synopsis</td>
<td>451</td>
</tr>
<tr>
<td>30</td>
<td>General utilities library summary</td>
<td>461</td>
</tr>
<tr>
<td>31</td>
<td>EqualityComparable requirements</td>
<td>462</td>
</tr>
<tr>
<td>32</td>
<td>LessThanComparable requirements</td>
<td>462</td>
</tr>
<tr>
<td>33</td>
<td>MoveConstructible requirements</td>
<td>462</td>
</tr>
<tr>
<td>34</td>
<td>CopyConstructible requirements</td>
<td>462</td>
</tr>
<tr>
<td>35</td>
<td>MoveAssignable requirements</td>
<td>462</td>
</tr>
<tr>
<td>36</td>
<td>CopyAssignable requirements</td>
<td>462</td>
</tr>
<tr>
<td>37</td>
<td>Swappable requirements</td>
<td>463</td>
</tr>
<tr>
<td>38</td>
<td>Destructible requirements</td>
<td>463</td>
</tr>
<tr>
<td>39</td>
<td>Descriptive variable definitions</td>
<td>463</td>
</tr>
<tr>
<td>40</td>
<td>Allocator requirements</td>
<td>465</td>
</tr>
<tr>
<td>41</td>
<td>Primary Type Category Predicates</td>
<td>482</td>
</tr>
<tr>
<td>42</td>
<td>Composite Type Category Predicates</td>
<td>483</td>
</tr>
<tr>
<td>43</td>
<td>Type Property Predicates</td>
<td>483</td>
</tr>
<tr>
<td>44</td>
<td>Type Property Queries</td>
<td>486</td>
</tr>
<tr>
<td>45</td>
<td>Type Relationship Predicates</td>
<td>487</td>
</tr>
<tr>
<td>46</td>
<td>Const-volatile modifications</td>
<td>488</td>
</tr>
<tr>
<td>47</td>
<td>Reference modifications</td>
<td>488</td>
</tr>
<tr>
<td>48</td>
<td>Sign modifications</td>
<td>489</td>
</tr>
<tr>
<td>49</td>
<td>Array modifications</td>
<td>489</td>
</tr>
<tr>
<td>50</td>
<td>Pointer modifications</td>
<td>490</td>
</tr>
<tr>
<td>51</td>
<td>Other transformations</td>
<td>490</td>
</tr>
<tr>
<td>52</td>
<td>Header <code>&lt;cstdlib&gt;</code> synopsis</td>
<td>545</td>
</tr>
<tr>
<td>53</td>
<td>Header <code>&lt;cstring&gt;</code> synopsis</td>
<td>545</td>
</tr>
<tr>
<td>54</td>
<td>Header <code>&lt;ctime&gt;</code> synopsis</td>
<td>546</td>
</tr>
<tr>
<td>55</td>
<td>Strings library summary</td>
<td>547</td>
</tr>
<tr>
<td>56</td>
<td>Traits requirements</td>
<td>548</td>
</tr>
<tr>
<td>57</td>
<td>basic_string(const Allocator&amp;) effects</td>
<td>562</td>
</tr>
<tr>
<td>58</td>
<td>basic_string(const basic_string&amp;) effects</td>
<td>562</td>
</tr>
<tr>
<td>59</td>
<td>basic_string(const basic_string&amp;, size_type, size_type, const Allocator&amp;) effects</td>
<td>563</td>
</tr>
<tr>
<td>60</td>
<td>basic_string(const charT*, size_type, const Allocator&amp;) effects</td>
<td>563</td>
</tr>
<tr>
<td>61</td>
<td>basic_string(const charT*, const Allocator&amp;) effects</td>
<td>563</td>
</tr>
<tr>
<td>62</td>
<td>basic_string(size_t, charT, const Allocator&amp;) effects</td>
<td>564</td>
</tr>
<tr>
<td>63</td>
<td>operator=(const basic_string&lt;charT, traits, Allocator&amp;) effects</td>
<td>564</td>
</tr>
<tr>
<td>64</td>
<td>operator=(const basic_string&lt;charT, traits, Allocator&gt;&amp;&amp;) effects</td>
<td>565</td>
</tr>
<tr>
<td>65</td>
<td>compare() results</td>
<td>578</td>
</tr>
<tr>
<td>66</td>
<td>Header <code>&lt;cctype&gt;</code> synopsis</td>
<td>586</td>
</tr>
<tr>
<td>67</td>
<td>Header <code>&lt;cwctype&gt;</code> synopsis</td>
<td>586</td>
</tr>
<tr>
<td>68</td>
<td>Header <code>&lt;cstring&gt;</code> synopsis</td>
<td>587</td>
</tr>
<tr>
<td>69</td>
<td>Header <code>&lt;cwchar&gt;</code> synopsis</td>
<td>587</td>
</tr>
<tr>
<td>70</td>
<td>Header <code>&lt;cstdlib&gt;</code> synopsis</td>
<td>587</td>
</tr>
<tr>
<td>71</td>
<td>Header <code>&lt;cuchar&gt;</code> synopsis</td>
<td>587</td>
</tr>
<tr>
<td>Table No.</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>72</td>
<td>Localization library summary</td>
<td>589</td>
</tr>
<tr>
<td>73</td>
<td>Locale Category Facets</td>
<td>593</td>
</tr>
<tr>
<td>74</td>
<td>Required Specializations</td>
<td>594</td>
</tr>
<tr>
<td>75</td>
<td>do_in/do_out result values</td>
<td>614</td>
</tr>
<tr>
<td>76</td>
<td>do_unshift result values</td>
<td>615</td>
</tr>
<tr>
<td>77</td>
<td>Integer Conversions</td>
<td>619</td>
</tr>
<tr>
<td>78</td>
<td>Length Modifier</td>
<td>620</td>
</tr>
<tr>
<td>79</td>
<td>Integer Conversions</td>
<td>624</td>
</tr>
<tr>
<td>80</td>
<td>Floating-point Conversions</td>
<td>624</td>
</tr>
<tr>
<td>81</td>
<td>Length Modifier</td>
<td>624</td>
</tr>
<tr>
<td>82</td>
<td>Numeric conversions</td>
<td>625</td>
</tr>
<tr>
<td>83</td>
<td>Fill padding</td>
<td>625</td>
</tr>
<tr>
<td>84</td>
<td>do_get_date effects</td>
<td>633</td>
</tr>
<tr>
<td>85</td>
<td>Header &lt;/locale&gt; synopsis</td>
<td>649</td>
</tr>
<tr>
<td>86</td>
<td>Containers library summary</td>
<td>651</td>
</tr>
<tr>
<td>87</td>
<td>Container requirements</td>
<td>652</td>
</tr>
<tr>
<td>88</td>
<td>Reversible container requirements</td>
<td>654</td>
</tr>
<tr>
<td>89</td>
<td>Sequence container requirements (in addition to container)</td>
<td>656</td>
</tr>
<tr>
<td>90</td>
<td>Optional sequence container operations</td>
<td>658</td>
</tr>
<tr>
<td>91</td>
<td>Associative container requirements (in addition to container)</td>
<td>659</td>
</tr>
<tr>
<td>92</td>
<td>Container requirements that are not required for unordered associative containers</td>
<td>663</td>
</tr>
<tr>
<td>93</td>
<td>Unordered associative container requirements (in addition to container)</td>
<td>664</td>
</tr>
<tr>
<td>94</td>
<td>Iterators library summary</td>
<td>743</td>
</tr>
<tr>
<td>95</td>
<td>Relations among iterator categories</td>
<td>743</td>
</tr>
<tr>
<td>96</td>
<td>Input iterator requirements</td>
<td>745</td>
</tr>
<tr>
<td>97</td>
<td>Output iterator requirements</td>
<td>745</td>
</tr>
<tr>
<td>98</td>
<td>Forward iterator requirements</td>
<td>746</td>
</tr>
<tr>
<td>99</td>
<td>Bidirectional iterator requirements (in addition to forward iterator)</td>
<td>747</td>
</tr>
<tr>
<td>100</td>
<td>Random access iterator requirements (in addition to bidirectional iterator)</td>
<td>748</td>
</tr>
<tr>
<td>101</td>
<td>Algorithms library summary</td>
<td>781</td>
</tr>
<tr>
<td>102</td>
<td>Header &lt;cstdlib&gt; synopsis</td>
<td>818</td>
</tr>
<tr>
<td>103</td>
<td>Numerics library summary</td>
<td>819</td>
</tr>
<tr>
<td>104</td>
<td>Uniform random number generator requirements</td>
<td>832</td>
</tr>
<tr>
<td>105</td>
<td>Random number engine requirements</td>
<td>833</td>
</tr>
<tr>
<td>106</td>
<td>Random number engine adaptor requirements</td>
<td>835</td>
</tr>
<tr>
<td>107</td>
<td>Random number distribution requirements</td>
<td>836</td>
</tr>
<tr>
<td>108</td>
<td>Header &lt;cmath&gt; synopsis</td>
<td>902</td>
</tr>
<tr>
<td>109</td>
<td>Header &lt;cstdio&gt; synopsis</td>
<td>903</td>
</tr>
<tr>
<td>110</td>
<td>Input/output library summary</td>
<td>907</td>
</tr>
<tr>
<td>111</td>
<td>fmtflags effects</td>
<td>917</td>
</tr>
<tr>
<td>112</td>
<td>fmtflags constants</td>
<td>917</td>
</tr>
</tbody>
</table>
Chapter 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 standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

— ISO/IEC 2382 (all parts), Information technology — Vocabulary
— ISO/IEC 9899:1990, Programming languages — C
— ISO/IEC 9899/Amd.1:1995, Programming languages — C, AMENDMENT 1: C Integrity
— ISO/IEC 9899:1999, Programming languages — C
— ISO/IEC TR 19769:2004, Information technology — Programming languages, their environments and system software interfaces — Extensions for the programming language C to support new character data types
The library described in clause 7 of ISO/IEC 9899:1990 and clause 7 of ISO/IEC 9899/Amd.1:1995 is hereinafter called the **Standard C Library**.1)


The library described in ISO/IEC TR 19769:2004 is hereinafter called the **C Unicode TR**.

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

The ECMAScript Language Specification described in Standard Ecma-262 is hereinafter called **ECMA-262**.

---

1.3 Definitions

For the purposes of this International Standard, the definitions given in ISO/IEC 2382 and the following definitions apply. 17.1 defines additional terms that are used only in clauses 17 through 27 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

An expression in the comma-separated list bounded by the parentheses in a function call expression; a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation; the operand of `throw`; or an expression, `type-id` or `template-name` in the comma-separated list bounded by the angle brackets in a template instantiation. Also known as an **actual argument** or **actual parameter**.

1.3.2 conditionally-supported

A 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]

1.3.3 diagnostic message

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

1.3.4 dynamic type

The type of the most derived object (1.8) to which the lvalue denoted by an lvalue 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] The dynamic type of an rvalue expression is its static type.

1.3.5 ill-formed

1) With the qualifications noted in clauses 17 through 27, and in C.2, the Standard C library is a subset of the Standard C++ library.

Draft
ill-formed program
input to a C++ implementation that is not a well-formed program ().

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

1.3.7 implementation limits
restrictions imposed upon programs by the implementation.

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

1.3.9 multibyte character
a sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment. The extended character set is a superset of the basic character set (2.2).

1.3.10 parameter
an 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; an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition; or a template-parameter. Parameters are also known as formal arguments or formal parameters.

1.3.11 signature
the name and the parameter-type-list (8.3.5) of a function, as well as the class or namespace of which it is a member. If a function or function template is a class member its signature additionally includes the cv-qualifiers (if any) on the function or function template itself. The signature of a function template additionally includes its return type and its template parameter list. The signature of a function template specialization includes the signature of the template of which it is a specialization and its template arguments (whether explicitly specified or deduced). [Note: Signatures are used as a basis for name mangling and linking. — end note ]

1.3.12 static type
the type of an expression (3.9), which type results from analysis of the program without considering execution semantics.
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.

1.3.13 undefined behavior
behavior, such as might arise upon use of an erroneous program construct or erroneous data, imposes no requirements. Undefined behavior may also be expected when this International Standard omits the description of any explicit definition of behavior. [Note: 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]

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

1.3.15 well-formed program
a C++ program constructed according to the syntax rules, diagnosable semantic rules, and the One Definition Rule (3.2).

1.4 Implementation compliance
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:

— 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\(^2\) that program.

— 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, except that

— 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.

\(^2\) “Correct execution” can include undefined behavior, depending on the data being processed; see 1.3 and 1.9.
For functions, function templates, objects, and values, the library clauses specify declarations. Implementations shall supply definitions consistent with the descriptions in the library clauses.

The names defined in the library have namespace scope (7.3). A C++ translation unit (2.1) obtains access to these names by including the appropriate standard library header (16.2).

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 (2.1).

Two kinds of implementations are defined: *hosted* and *freestanding*. 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 (17.4.1.3).

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

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

### 1.5 Structure of this International Standard

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.

Clauses 17 through 27 and Annex D (the *library clauses*) describe the Standard C++ library, which provides definitions for the following kinds of entities: macros (16.3), values (clause 3), types (8.1, 8.3), templates (clause 14), classes (clause 9), functions (8.3.5), and objects (clause 7).

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.

Finally, Annex E says what characters are valid in universal-character names in C++ identifiers (2.10).

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 presented on one line, marked by the phrase “one of.” An optional terminal or nonterminal symbol is indicated by the subscript “\(_\text{opt}\)”, so

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

\(^3\)This documentation also defines implementation-defined behavior; see 1.9.
1.7 The C++ memory model [intro.memory]

1.7 The C++ memory model is the byte. A byte is at least large enough to contain any member of the basic execution character set 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.

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 expression (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:

— If x is a complete object, then x is the complete object of x.
— 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 POD trivial or standard-layout type (3.9) shall occupy contiguous bytes of storage.

---

4) removed footnote
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.\(^5\)

Certain aspects and operations of the abstract machine are described in this International Standard as implementation-defined (for example, `sizeof(int)`). These constitute the parameters of the abstract machine. Each implementation shall include documentation describing its characteristics and behavior in these respects.\(^6\) 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, order of evaluation of arguments to a function). 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 sequence for a given program and a given input.

Certain other operations are described in this International Standard as undefined (for example, the effect of dereferencing the null pointer). \(^4\)[Note: this International Standard imposes no requirements on the behavior of programs that contain undefined behavior. — end note ]\(^4\)

A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible execution sequences of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution sequence 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).

The observable behavior of the abstract machine is its sequence of reads and writes to volatile data and calls to library I/O functions.\(^7\)

When the processing of the abstract machine is interrupted by receipt of a signal, the values of objects with type other than `volatile std::sig_atomic_t` are unspecified, and the value of any object not of type `volatile std::sig_atomic_t` that is modified by the handler becomes undefined.

An instance of each object with automatic storage duration \(^3\) 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:

---

\(^5\) 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.

\(^6\) This documentation also includes conditionally-supported constructs and locale-specific behavior. See 1.4.

\(^7\) An implementation can offer additional library I/O functions as an extension. Implementations that do so should treat calls to those functions as “observable behavior” as well.

Draft
— Access to volatile objects are evaluated strictly according to the rules of the abstract machine.
— 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.
— The input and output dynamics of interactive devices shall take place in such a fashion that prompting messages actually appear prior to a program waiting for input. What constitutes an interactive device is implementation-defined.

[Note: more stringent correspondences between abstract and actual semantics may be defined by each implementation. —end note]

10 [Note: operators can be regrouped according to the usual mathematical rules only where the operators really are associative or commutative.\(^8\)] 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]

11 A full-expression is an expression that is not a subexpression of another expression. If a language construct is defined to produce an implicit call of a function, a use of the language construct is considered to be an expression for the purposes of this definition. 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:

\(^8\) Overloaded operators are never assumed to be associative or commutative.
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() {
    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 argument expressions (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 lvalue (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 lvalue evaluation and fetching a value previously assigned to an object for rvalue 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, 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.9).

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 a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined.

[9] 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.\(^\text{10}\) 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 Acknowledgments


All rights in these originals are reserved.

\(^{10}\)In other words, function executions do not interleave with each other.
Chapter 2  Lexical conventions  [lex]

1  The text of the program is kept in units called source files in this International Standard. A source file together with all the headers (17.4.1.2) 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 ]

2  [ 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 ]

2.1 Phases of translation  [lex.phases]

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

1.1) 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. Trigraph sequences (2.3) are replaced by corresponding single-character internal representations. Any source file character not in the basic source character set (2.2) 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.)

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. If, as a result, a character sequence that matches the syntax of a universal-character-name is produced, the behavior is undefined. If a source file that is not empty does not end in a new-line character, or ends in a new-line character immediately preceded by a backslash character before any such splicing takes place, the behavior is undefined.

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.\(^{\text{(2)}}\) 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 implementation-defined.

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

\(^{\text{(2)}}\) 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 /* or */ comment.
The process of dividing a source file’s characters into preprocessing tokens is context-dependent. [Example: see the handling of < within a #include preprocessing directive. — end example]

4. Preprocessing directives are executed, macro invocations are expanded, and _Pragma unary operator expressions are executed. If a character sequence that matches the syntax of a universal-character-name is produced by token concatenation (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, escape sequence, or universal-character-name in character literals and string literals is converted to the corresponding member of the execution character set (2.13.2, 2.13.4); if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.¹³

6. Adjacent 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 object and function references are resolved. Library components are linked to satisfy external references to functions and objects 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.

2.2 Character sets

The basic source character set consists of 96 characters: the space character, the control characters representing horizontal tab, vertical tab, form feed, and new-line, plus the following 91 graphical characters:¹⁴

```
  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
```

¹³An implementation need not convert all non-corresponding source characters to the same execution character.
¹⁴The 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 universal-character-name construct provides a way to name other characters.

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 is less than 0x20 or in the range 0x7F-0x9F (inclusive), or if the universal-character-name designates a character in the basic source character set, then 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 representation has all zero bits. 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 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 are implementation-defined, and any additional members are locale-specific.

### 2.3 Trigraph sequences

Before any other processing takes place, each occurrence of one of the following sequences of three characters (“trigraph sequences”) is replaced by the single character indicated in Table 1.

<table>
<thead>
<tr>
<th>trigraph</th>
<th>replacement</th>
<th>trigraph</th>
<th>replacement</th>
<th>trigraph</th>
<th>replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>??=</td>
<td>#</td>
<td>??&lt;</td>
<td>(</td>
<td>??&lt;</td>
<td>(</td>
</tr>
<tr>
<td>??/</td>
<td>\</td>
<td>??)</td>
<td>)</td>
<td>??&gt;</td>
<td>}</td>
</tr>
<tr>
<td>??\</td>
<td>^</td>
<td>??!</td>
<td>!</td>
<td>??-</td>
<td>~</td>
</tr>
</tbody>
</table>

Example:

```c
?=define arraycheck(a,b) a?!(b??) ??!??! b??(a??)
```

becomes

```c
#define arraycheck(a,b) a[b] || b[a]
```

—end example ]

No other trigraph sequence exists. Each that does not begin one of the trigraphs listed above is not changed.
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, string literals, preprocessing-op-or-punc, 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, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token, even if that would cause further lexical analysis to fail.

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

[Example: The program fragment x+++++y is parsed as x ++ ++ + y, which, if x and y are of built-in types, violates a constraint on increment operators, even though the parse x ++ ++ y might yield a correct expression. —end example]

2.5 Alternative tokens

Alternative token representations are provided for some operators and punctuators.¹⁵)

In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its
The set of alternative tokens is defined in Table 2.

### 2.6 Tokens

<table>
<thead>
<tr>
<th>alternative</th>
<th>primary</th>
<th>alternative</th>
<th>primary</th>
<th>alternative</th>
<th>primary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;%</code></td>
<td>{</td>
<td><code>and</code></td>
<td><code>&amp;&amp;</code></td>
<td><code>and_eq</code></td>
<td><code>&amp;=</code></td>
</tr>
<tr>
<td><code>%&gt;</code></td>
<td><code>}</code></td>
<td><code>bitor</code></td>
<td>`</td>
<td></td>
<td>`</td>
</tr>
<tr>
<td><code>&lt;:</code></td>
<td><code>[]</code></td>
<td><code>or</code></td>
<td>`</td>
<td></td>
<td>`</td>
</tr>
<tr>
<td><code>:</code></td>
<td><code>[]</code></td>
<td><code>xor</code></td>
<td><code>^</code></td>
<td><code>not</code></td>
<td><code>!</code></td>
</tr>
<tr>
<td><code>%:</code></td>
<td><code>#</code></td>
<td><code>compl</code></td>
<td><code>∼</code></td>
<td><code>not_eq</code></td>
<td><code>!=</code></td>
</tr>
<tr>
<td><code>%;%:</code></td>
<td><code>##</code></td>
<td><code>bitand</code></td>
<td><code>&amp;</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

### 2.7 Comments

The characters `/*` start a comment, which terminates with the characters `*/`. These comments do not nest. The characters `//` start a comment, which terminates with the next new-line character. If there is a form-feed or a vertical-tab character in such a comment, only white-space characters shall appear between it and the new-line that terminates the comment; no diagnostic is required. [Note: The comment characters `//`, `/*`, and `*/` have no special meaning within a `//` comment and are treated just like other characters. Similarly, the comment characters `//` and `/*` have no special meaning within a `/*` comment. — end note]

### 2.8 Header names

<table>
<thead>
<tr>
<th>header-name:</th>
<th>&lt; h-char-sequence &gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>q-char-sequence:</td>
<td>*</td>
</tr>
</tbody>
</table>

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

16) Thus the “stringized” values (16.3.2) of `[]` and `<:` will be different, maintaining the source spelling, but the tokens can otherwise be freely interchanged.

17) Literals include strings and character and numeric literals.

Draft
2.9 Preprocessing numbers

Precisely, with the number of preprocessing tokens shall only appear within a #include preprocessing directive (16.2). The sequences in both forms of header-name are mapped in an implementation-defined manner to headers or to external source file names as specified in 16.2.

If either of the characters ’ or \, or either of the character sequences /* or // appears in a q-char-sequence or a h-char-sequence, or the character ” appears in a h-char-sequence, the behavior is undefined.18)

2.9 Preprocessing numbers

pp-number:
  . digit
   . digit
   pp-number digit
   pp-number nondigit
   pp-number e sign
   pp-number E sign
   pp-number .

Preprocessing number tokens lexically include all integral literal tokens (2.13.1) and all floating literal tokens (2.13.3).

A preprocessing number does not have a type or a value; it acquires both after a successful conversion (as part of translation phase 7, 2.1) to an integral literal token or a floating literal token.

2.10 Identifiers

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

18) Thus, sequences of characters that resemble escape sequences cause undefined behavior.
17 Lexical conventions

2.11 Keywords

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 Annex E. Upper- and lower-case letters are different. All characters are significant.\(^1\)

In addition, some identifiers are reserved for use by C++ implementations and standard libraries (17.4.3.1.2) and shall not be used otherwise; no diagnostic is required.

2.11 Keywords [lex.key]

The identifiers shown in Table 3 are reserved for use as keywords (that is, they are unconditionally treated as keywords in phase 7):

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

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>not_eq</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 [lex.operators]

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:

\(^{1}\)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 \texttt{\textbackslash a} 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.
2.13 Literals

Each `preprocessing-op-or-punc` is converted to a single token in translation phase 7 (2.1).

2.13 Literals

There are several kinds of literals.\(^{20}\)

- literal:
  - integer-literal
  - character-literal
  - floating-literal
  - string-literal
  - boolean-literal

2.13.1 Integer literals

- integer-literal:
  - decimal-literal `integer-suffix_{opt}`
  - octal-literal `integer-suffix_{opt}`
  - hexadecimal-literal `integer-suffix_{opt}`

- decimal-literal:
  - `nonzero-digit`
  - `decimal-literal digit`

- octal-literal:
  - 0
  - `octal-literal octal-digit`

- hexadecimal-literal:
  - `0x hexadecimal-digit`
  - `0X hexadecimal-digit`
  - `hexadecimal-literal hexadecimal-digit`

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

- `octal-digit`: one of
  - 0 1 2 3 4 5 6 7

- `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

\(^{20}\) 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. 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 decimal integer literal (base ten) begins with a digit other than 0 and consists of a sequence of decimal digits. An octal integer literal (base eight) begins with the digit 0 and consists of a sequence of octal 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, or 0XC. —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 Constant</th>
<th>Octal or Hexadecimal Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>long</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>long long</td>
<td>long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>u or U</td>
<td>unsigned int</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>unsigned long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td>unsigned long long int</td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>u or U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l or L</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Both u or U and l or L</td>
<td>unsigned long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>ll or LL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Both u or U and ll or LL</td>
<td>unsigned long long int</td>
<td>unsigned long long int</td>
</tr>
</tbody>
</table>

If an integer literal cannot be represented by any type in its list and an extended integer type can represent its value, it

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

2.13.2 Character literals

character-literal:
   ' c-char-sequence '
   u' c-char-sequence '
   U' c-char-sequence '
   L' c-char-sequence '

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

c-char:
   any member of the source character set except
   the single-quote ', backslash \, or new-line character

escape-sequence
universal-character-name

escape-sequence:
   simple-escape-sequence
   octal-escape-sequence
   hexadecimal-escape-sequence

simple-escape-sequence:
   one of
   \ , \a \b \f \n \r \t \v

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

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

1 A character literal is one or more characters enclosed in single quotes, as in ‘x’, optionally preceded by one of the letters u, U, or L, as in u’y’, U’z’, or L’x’, respectively. A character literal that does not begin with u, U, or L is an ordinary character literal, also referred to as a narrow-character literal. An ordinary character literal that contains a single c-char 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 has type int and implementation-defined value.

2 A character literal that begins with the letter u, such as u’y’, 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-char s is ill-formed. A character literal that begins with the letter U, such as U’z’, 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-char s is ill-formed. A character literal that begins with the letter L, such as L'x', is a wide-character literal. A wide-character literal has type wchar_t. The value of a wide-character literal containing a single c-char has value equal to the numerical value of the encoding of the c-char in the execution wide-character set. The value of a wide-character literal containing multiple c-char s is implementation-defined.

Certain non-graphic characters, the single quote ‘, the double quote " , the question mark ?, 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.

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-line</td>
<td>\n</td>
</tr>
<tr>
<td>horizontal tab</td>
<td>\t</td>
</tr>
<tr>
<td>vertical tab</td>
<td>\v</td>
</tr>
<tr>
<td>backspace</td>
<td>\b</td>
</tr>
<tr>
<td>carriage return</td>
<td>\r</td>
</tr>
<tr>
<td>form feed</td>
<td>\f</td>
</tr>
<tr>
<td>alert</td>
<td>\a</td>
</tr>
<tr>
<td>backslash</td>
<td>\</td>
</tr>
<tr>
<td>question mark</td>
<td>?</td>
</tr>
<tr>
<td>single quote</td>
<td>‘</td>
</tr>
<tr>
<td>double quote</td>
<td>&quot;</td>
</tr>
<tr>
<td>octal number</td>
<td>\ooo</td>
</tr>
<tr>
<td>hex number</td>
<td>\xhhh</td>
</tr>
</tbody>
</table>

The escape \ooo consists of the backslash followed by one, two, or three octal digits that are taken to specify the value of the desired character. The escape \xhhh consists of the backslash followed by x followed by one or more hexadecimal digits that are taken to specify the value of the desired character. There is no limit to the number of digits in a hexadecimal sequence. A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. The value of a character literal is implementation-defined if it falls outside of the implementation-defined range defined for char (for literals with no prefix), char16_t (for literals prefixed by 'u'), char32_t (for literals prefixed by 'U'), or wchar_t (for literals prefixed by 'L').

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

22) They are intended for character sets where a character does not fit into a single byte.
2.13 Literals

2.13.3 Floating literals

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. 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.

2.13.4 String literals

A string literal is a sequence of characters (as defined in 2.13.2) surrounded by double quotes, optionally beginning with one of the letters u, U, or L, as in "...", u"...", U"...", or L"...", respectively.
A string literal that does not begin with u, U, or L is an ordinary string literal, also referred to as a narrow string literal. An ordinary string literal has type “array of n const char”, where n is the size of the string as defined below; it has static storage duration (3.7) and is initialized with the given characters.

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 has static storage duration and is initialized with the given characters. A single c-char may produce more than one char16_t character in the form of surrogate pairs.

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 has static storage duration and 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 has static storage duration and is initialized with the given characters.

Whether all string literals are distinct (that is, are stored in nonoverlapping objects) is implementation-defined. The effect of attempting to modify a string literal is undefined.

In translation phase 6 (2.1), adjacent string literals are concatenated. If both string literals have the same prefix, the resulting concatenated string literal has that prefix. If one string literal has no prefix, it is treated as a string literal of the same prefix as the other operand. Any other concatenations are conditionally supported with implementation-defined behavior. [Note: This concatenation is an interpretation, not a conversion. —end note] [Example: Here are some examples of valid concatenations:

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

—end example]

Characters in concatenated strings are kept distinct.

[Example:

"\xA" "B"

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

After any necessary concatenation, in translation phase 7 (2.1), '0' is appended to every string literal so that programs that scan a string can find its end.

Escape sequences and universal-character-names in string literals have the same meaning as in character literals (2.13.2), except that the single quote ' is representable either by itself or by the escape sequence \\', and the double quote " shall be preceded by a \\. 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,
2.13 Literals

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 must 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’.

2.13.5 Boolean literals

boolean-literal:
false
true

The Boolean literals are the keywords false and true. Such literals have type bool. They are not lvalues.
Chapter 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 notion of an lvalue. 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, subobject, base class subobject, array element, variable, function, instance of a function, enumerator, type, class member, template, namespace, or parameter pack.

4 A name is a use of an identifier (2.10) that denotes an entity or label (6.6.4, 6.1). A variable is introduced by the declaration of an object. The variable’s name denotes the object.

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 Some names denote types, classes, enumerations, or templates. In general, it is necessary to determine whether or not a name denotes one of these entities before parsing the program that contains it. The process that determines this is called name lookup (3.4).

7 Two names are the same if
   — they are identifiers composed of the same character sequence; or
   — they are the names of overloaded operator functions formed with the same operator; or
   — they are the names of user-defined conversion functions formed with the same type.

8 An identifier 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 identifier specified in each translation unit.

3.1 Declarations and definitions  [basic.def]

1 A declaration (clause 7) introduces names into a translation unit or redeclares names introduced by previous declarations. A declaration specifies the interpretation and attributes of these names.

2 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-specification\(^{23}\) (7.5) and neither an initializer nor a function-body, it declares a static data member in a class definition (9.4), it is a class name declaration (9.1), or it is a typedef declaration (7.1.3), a using-directive using-declaration (7.3.3), or a (7.3.4).

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

```c
int a; // defines a
extern const int c = 1; // defines c
int f(int x) { return x+a; } // defines f and defines x
struct S { int a; int b; }; // defines S, S::a, and S::b
struct X {
    int x; // defines non-static data member x
    static int y; // declares static data member y
    X(): x(0) { } // defines a constructor of X
};
int X::y = 1; // defines X::y
enum { up, down; } // defines up and down
namespace N { int d; } // defines N and N::d
namespace M1 = N; // defines M1
X anX; // defines anX
```

whereas these are just declarations:

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

—end example]

[Note: in some circumstances, C++ implementations implicitly define the default constructor (12.1), copy constructor (12.8), assignment operator (12.8), or destructor (12.4) member functions. [Example: given

```c
#include <string>

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

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

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

\(^{23}\) Appearing inside the braced-enclosed declaration-seq in a linkage-specification does not affect whether a declaration is a definition.

Draft
struct C {
    std::string s;
    C() : s() { }
    C(const C& x) : s(x.s) { }
    C& operator=(const C& x) { s = x.s; return *this; }
    ~C() { }
};

— end example — end note

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. An object or non-overloaded function whose name appears as a potentially-evaluated expression is used unless it is an object that satisfies the requirements for appearing in an integral constant expression (5.19) and the lvalue-to-rvalue conversion (4.1) is immediately applied. A virtual member function is used if it is not pure. An overloaded function is used if it is selected by overload resolution when referred to from a potentially-evaluated expression. [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 copy constructor is used even if the call is actually elided by the implementation. — end note] An allocation or deallocation function for a class is 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 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 used by the definition of a constructor of that class. A non-placement deallocation function for a class is 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).2) A copy-assignment function for a class is used by an implicitly-defined copy-assignment function for another class as specified in 12.8. A default constructor for a class is used by default initialization or value initialization as specified in 8.5. A constructor for a class is used as specified in 8.5. A destructor for a class is used as specified in 12.4.

3 Every program shall contain exactly one definition of every non-inline function or object that is 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 used.

4 Exactly one definition of a class is required in a translation unit if the class is used in a way that requires the class type to be complete. [Example: the following complete translation unit is well-formed, even though it never defines X:

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

2) An implementation is not required to call allocation and deallocation functions from constructors or destructors; however, this is a permissible implementation technique.
3.2 One definition rule

— *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 an lvalue 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 *void* $\ast$, is converted to the type pointer to $T$ or reference to $T$ using an implicit 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

— 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 (10), or

— an lvalue of type $T$ is assigned to (5.17), or

— the type $T$ is the subject of an *alignof* expression (5.3.6) or an *alignas* specifier (7.1.7).

— *end note* ]

5 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 *const* object with internal or no linkage if the object has the same *integral* or *enumeration* *literal* type in all definitions of $D$, and the object is initialized with a constant expression (5.19), and the value (but not the address) of the object is used, and the object has the same value in all definitions of $D$; 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).  

— if D is a class with an implicitly-declared constructor (12.1), it is as if the constructor was implicitly defined in every translation unit where it is used, and the implicit definition in every translation unit shall call the same constructor for a base class or a class member of D. [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()

// translation unit 2:
struct X {
    X(int);
    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 last four requirements from the list above shall apply 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 program shall behave 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 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.

[Example: in

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

25) 8.3.6 describes how default argument names are looked up.
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 (11.4), certain uses of the elaborated-type-specifier (7.1.6.3), and using-directive s (7.3.4) alter this general behavior.

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

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

— exactly one declaration shall declare a class name or enumeration name that is not a typedef name and the other declarations shall all refer to the same object or enumerator, or all refer to functions and function templates; in this case the class name or enumeration name is hidden (3.3.8). [Note: a namespace name or a class template name must be unique in its declarative region (7.3.2, clause 14). —end note]

[Note: these restrictions apply to the declarative region into which a name is introduced, which is not necessarily the same as the region in which the declaration occurs. In particular, elaborated-type-specifier s (7.1.6.3) and friend declarations (11.4) 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]

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

3.3.1 Point of declaration [basic.scope.pdecl]

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

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

2 [Note: a nonlocal name remains visible up to the point of declaration of the local name that hides it.][Example:

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

declares a local array of two integers. —end example] —end note]

3 The point of declaration for a class 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 its enum-specifier (7.2). The point of declaration of a template alias immediately follows the identifier for the alias being declared.

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

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

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

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

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

— end note ]

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

— for a declaration of the form `class-key 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 non-class, non-function-prototype 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 (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.

[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 object 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 ]

### 3.3.2 Local scope

A name declared in a block (6.3) is local to that block. Its potential scope begins at its point of declaration (3.3.1) and ends at the end of its declarative region.

The potential scope of a function parameter name 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, else 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.

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

4 Names declared in the for-init-statement, 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.

3.3.3 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.

3.3.4 Function scope [basic.funscope]

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

3.3.5 Namespace scope [basic.scope.namespace]

The declarative region of a namespace-definition is its namespace-body. The potential scope denoted by an original-name-space-name is the concatenation of the declarative regions established by each of the namespace-declarations in the same declarative region with that original-name-space-name. 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 (3.3.1) 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:

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

### 3.3 Declarative regions and scopes

```cpp
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 (3.3.1) and ends at the end of the translation unit that is its declarative region. Names with global namespace scope are said to be global.

#### 3.3.6 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, and default arguments, and constructor ctor-initializers, 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, member function definitions (including the member function body and, for constructor functions (12.1), the ctor-initializer (12.6.2)) and any portion of the declarator part of such definitions which follows the identifier, including a parameter-declaration-clause and any default arguments (8.3.6).[Example:

typedef int c;
enum { i = 1 };

class X {
char v[i]; // error: i refers to ::i
    // but when reevaluated is X::i
    int f() { return sizeof(c); } // OK: X::c
    char c;
enum { i = 2 };
3.3 Declarative regions and scopes

};
typedef char* T;
struct Y {
    T a;                                  // error: T refers to ::T
    typedef long T;                     // but when reevaluated is Y::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:

— in the scope of its class (as described above) or a class derived (clause 10) from its class,
— after the . operator applied to an expression of the type of its class (5.2.5) or a class derived from its class,
— after the -> operator applied to a pointer to an object of its class (5.2.5) or a class derived from its class,
— after the :: scope resolution operator (5.1) applied to the name of its class or a class derived from its class.

3.3.7 Enumeration scope [basic.scope.enum]

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.

3.3.8 Name hiding [basic.scope.hiding]

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

A class name (9.1) or enumeration name (7.2) can be hidden by the name of an object, function, or enumerator declared in the same scope. If a class or enumeration name and an object, 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 object, function, or enumerator name is visible.

In a member function definition, the declaration of a local name hides the declaration of a member of the class with the same name; see 3.3.6. 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.

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

### 3.4 Name lookup

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).

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

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

[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

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.

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.

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,

```cpp
typedef int f;
namespace N {
    struct A {
        friend void f(A &);
        operator int();
        void g(A a) {
            int i = f(a);    // i 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]
3.4 Name lookup  

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

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

A name used in the definition of a function following the function’s declarator-id 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:

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

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

—end example ]

A name used in the definition of a class X outside of a member function body or nested class definition shall be declared in one of the following ways:

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

— 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), or

— 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

— if X is a member of namespace N, or is a nested class of a class that is a member of N, or is a local class or a nested class within a local class of a function that is a member of N, before the definition of class X in namespace N or in one of N ’s enclosing namespaces.

Example:

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

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

27) 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.

28) 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).
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: 3.3.6 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 ]

A name used in the definition of a member function (9.3) of class X following the function’s declarator-id 29) shall be declared in one of the following ways:

— before its use in the block in which it is used or in an enclosing block (6.3), or
— shall be a member of class X or be a member of a base class of X (10.2), or
— 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) 30) or
— 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
— 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 member function definition, in namespace N or in one of N’s enclosing namespaces.

[ Example:
    
    class B {
    
    name space M {
        namespace N {
            class X : public B {
                void f();
            };
        }
    }

    29) That is, an unqualified name that occurs, for instance, in a type or default argument expression in the parameter-declaration-clause or in the function body, or in an expression of a non-initializer in a constructor definition.

    30) 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.

Draft
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 ]

9 Name lookup for a name used in the definition of a friend function (11.4) defined inline in the class granting friendship shall proceed as described for lookup in member function definitions. If the friend function is not defined in the class granting friendship, name lookup in the friend function definition shall proceed as described for lookup in namespace member function definitions.

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

```c
struct A {
    typedef int AT;
    void f1(AT);
    void f2(float);
};
struct B {
    typedef float BT;
    friend void A::f1(AT);  // parameter type is A::AT
    friend void A::f2(BT);  // parameter type is B::BT
};

— end example ]

11 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 ]

12 During the lookup of a name used in the constant-expression of an enumerator-definition, previously declared enumerators of the enumeration are visible and hide the names of entities declared in the block, class, or namespace scopes containing the enum-specifier.
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 used in the definition of the variable member (after the declarator-id) is looked up as if the definition of the variable 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

When an unqualified name is used as the postfix-expression in a function call (5.2.2), other namespaces not considered during the usual unqualified lookup (3.4.1) may be searched, and in those namespaces, namespace-scope friend function declarations (11.4) not otherwise visible may be found. These modifications to the search depend on the types of the arguments (and for template template arguments, the namespace of the template argument).

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-declaration used to specify the types do not contribute to this set. The sets of namespaces and classes are determined in the following way:

— 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 namespaces of which its associated classes are members. 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]
3.4 Name lookup

— If $T$ is an enumeration type, its associated namespace is the namespace in which it is defined. If it is 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 $X$.

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: the namespace in which the function or function template is defined and the classes and namespaces associated with its (non-dependent) parameter types and return type.

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 *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:

— Any *using-directive* s in the associated namespace are ignored.
— 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 (11.4).
— All names except those of (possibly overloaded) functions and function templates are ignored.

### 3.4.3 Qualified name lookup

The name of a class or namespace member or enumeration can be referred to after the :: scope resolution operator (5.1) applied to a nested-name-specifier that nominates its class, or namespace, or enumeration. During the lookup for a name preceding the :: scope resolution operator, object, function, and enumerator names are ignored. If the name found does not designate a namespace or a class, enumeration, or dependent type, the program is ill-formed.[Example:

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

— end example ]

[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 ]

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

```cpp
class X { }
class C {
    class X { }
    static const int number = 50;
    static X arr[number];
};
X C::arr[number];    // ill-formed:
// equivalent to: ::X C::arr[C::number];
// not to: ::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 (3.3.8).

A name prefixed by a nested-name-specifier that nominates an enumeration type shall represent an enumeration of that enumeration.
If a pseudo-destructor-name (5.2.4) contains a nested-name-specifier, the type-name s are looked up as types in the scope designated by the nested-name-specifier. Similarly, in a qualified-id of the form:

```cpp
class-name :opt nested-name-specifier_opt class-name : : ~ class-name
```

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

```cpp
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
```

```cpp
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

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 (3.3.6). — end note] The exceptions to the name lookup rule above are the following:

- a destructor name is looked up as specified in 3.4.3;
- a conversion-type-id of an operator-function-id is looked up both in the scope of the class and in the context in which the entire postfix-expression occurs and shall refer to the same type in both contexts;
- the names in a template-argument of a template-id are looked up in the context in which the entire postfix-expression occurs.
- the lookup for a name specified in a using-declaration (7.3.3) also finds class or enumeration names hidden within the same scope (3.3.8).

2 In a lookup in which the constructor is an acceptable lookup result, if the nested-name-specifier nominates a class C, and the name specified after the nested-name-specifier, when looked up in C, is the injected-class-name of C (clause 9), 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.

Draft
3.4 Name lookup

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

Example:

```c
struct A { A(); }
struct B: public A { B(); }
A::A() {
} // object of type 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
```

Example

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

1. If the nested-name-specifier of a qualified-id nominates a namespace, the name specified after the nested-name-specifier is looked up in the scope of the namespace, except that the names in a template-argument of a template-id are looked up in the context in which the entire postfix-expression occurs.

2. Given X::m (where X is a user-declared namespace), or given ::m (where X is the global namespace), let S be the set of all declarations of m in X and in the transitive closure of all namespaces nominated by using-directive s in X and its used namespaces, except that using-directive s are ignored in any namespace, including X, directly containing one or more declarations of m. No namespace is searched more than once in the lookup of a name. If S is the empty set, the program is ill-formed. Otherwise, if S has exactly one member, or if the context of the reference is a using-declaration (7.3.3), S 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, 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);
    void g(int);
    int i;
}
namespace B {
```

Draft
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);      // i 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)
}

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;

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

```cpp
namespace B {
    int b;
}

namespace A {
    using namespace B;
    int a;
}

namespace B {
    using namespace A;
}

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

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

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

namespace B {
    struct y { }; 
}

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

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

```
nested-name-specifier unqualified-id
```
the unqualified-id shall name a member of the namespace designated by the nested-name-specifier. [Example:

```cpp
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:

```cpp
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 ---

### 3.4.4 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 (3.3.8).

1. 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 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 identifier;
   ```

   the elaborated-type-specifier is a declaration that introduces the class-name as described in 3.3.1.

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:

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

struct Data {
    struct Node* Node;     // OK: Refers to Node at global scope
    friend struct ::Glob;  // error: Glob is not declared
                             // cannot introduce a qualified type (7.1.6.3)
    friend struct Glob;    // OK: Refers to (as yet) undeclared Glob
                             // at global scope.
                          /**< ... */
};

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

---

Draft
3.4 Name lookup

struct Data; // OK: Redefines 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::Datum; // error: Datum undefined
struct Base::Data* pBase; // OK: refers to nested Data

— end example ]

3.4.5 Class member access [basic.lookup.classref]

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 or function template. If the lookup in the class of the object expression finds a template, the name is also looked up in the context of the entire postfix-expression and
   — if the name is not found, the name found in the class of the object expression is used, otherwise
   — if the name is found in the context of the entire postfix-expression and does not name a class template, the name found in the class of the object expression is used, otherwise
   — if the name found is a class template, it must refer to the same entity as the one found in the class of the object expression, otherwise the program is ill-formed.

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. If the type of the object expression is of pointer to scalar type, 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.

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 looked up both in the context of the entire postfix-expression and in the scope of the class of the object expression. If the name is found only in the scope of the class of the object expression, the name shall refer to a class-name. If the name is found only in the context of the entire postfix-expression, the name shall refer to a class-name or namespace-name. If the name is found in both contexts, the class-name-or-namespace-name shall refer to the same entity.

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-argument s 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* shall denote the same type in both the context in which the entire *postfix-expression* occurs and in the context of the class of the object expression (or the class pointed to by the pointer expression).

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

1 When looking up a *namespace-name* in a *using-directive* or *namespace-alias-definition*, only namespace names are considered.

3.5 Program and linkage [basic.link]

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

   translation-unit:
   
   declaration-seqopt

2 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:

   — 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.

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

   — When a name has *no linkage*, the entity it denotes cannot be referred to by names from other scopes.

3 A name having namespace scope (3.3.5) has internal linkage if it is the name of

   — an object, reference, function or function template that is explicitly declared static or,

   — an object or reference that is explicitly declared const and neither explicitly declared extern nor previously declared to have external linkage; or

   — a data member of an anonymous union.

4 A name having namespace scope has external linkage if it is the name of

   — an object or reference, unless it has internal linkage; or

   — a function, unless it has internal linkage; or

   — 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

   — 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

   — an enumerator belonging to an enumeration with external linkage; or

   — a template, unless it is a function template that has internal linkage (clause 14); or

   — a namespace (7.3), unless it is declared within an unnamed namespace.
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 external linkage if the name of the class has external linkage.

The name of a function declared in block scope, and the name of an object 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]

Draft
Names not covered by these rules have no linkage. Moreover, except as noted, a name declared in a local scope (3.3.2) has no linkage. A type is said to have linkage if and only if:

- it is a class or enumeration type that is named (or has a name for linkage purposes (7.1.3)) and the name has linkage; or
- it is a specialization of a class template (14)\(^{31}\); or
- it is a fundamental type (3.9.1); or
- it is a compound type (3.9.2) other than a class or enumeration, compounded exclusively from types that have linkage; or
- 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 linkage, unless the variable or function has extern "C" linkage (7.5). [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 is not permitted. 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:

```c
void f()
{
    struct A { int x; }; // no linkage
    extern A a; // ill-formed
    typedef A B;
    extern B b; // ill-formed
}
```

— end example] This implies that names with no linkage cannot be used as template arguments (14.3).

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

- both names have external linkage or else both names have internal linkage and are declared in the same translation unit; and
- both names refer to members of the same namespace or to members, not by inheritance, of the same class; and
- when both names denote functions, the parameter-type-lists of the functions (8.3.5) are identical; and
- when both names denote function templates, the signatures (14.5.6.1) are the same.

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 object 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]

\(^{31}\) A class template always has external linkage, and the requirements of 14.3.1 and 14.3.2 ensure that the template arguments will also have appropriate linkage.
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 return type of type int, but otherwise its type is implementation-defined. All implementations shall allow both of the following definitions of main:

```c
int main() { /* ... */ }
```

and

```c
int main(int argc, char* argv[]) { /* ... */ }
```

In the latter form 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 s) (17.3.2.1.3.2) 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 nonnegative. 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 (3.2) within a program. The linkage (3.5) of main is implementation-defined. A program that declares main to be inline or static 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]

Calling the function std::exit(int) declared in <cstdlib> (18.4) terminates the program without leaving the current block and hence without destroying 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 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

```c
return 0;
```

3.6.2 Initialization of non-local objects

Objects with static storage duration (3.7.1) shall be zero-initialized (8.5) before any other initialization takes place. A reference with static storage duration and an object of POD trivial or literal type with static storage duration can be initialized with a constant expression (5.19); this is called constant initialization. Together, zero-initialization and constant initialization are called static initialization; all other initialization is dynamic initialization. Static initialization shall be performed before any dynamic initialization takes place. Dynamic initialization of an object is either ordered or unordered. Definitions of explicitly specialized class template static data members have ordered initialization. Other class template static data members (i.e., implicitly or explicitly instantiated specializations) have unordered initialization. Other objects defined in namespace scope have ordered initialization. Objects defined within a single translation unit
and with ordered initialization shall be initialized in the order of their definitions in the translation unit. The order of initialization is unspecified for objects with unordered initialization and for objects defined in different translation units. An unordered initialization is indeterminately sequenced with respect to every other dynamic initialization. [Note: 8.5.1 describes the order in which aggregate members are initialized. The initialization of local static objects is described in 6.7. — end note]

An implementation is permitted to perform the initialization of an object of namespace scope 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 with static storage duration prior to its initialization, and

— the static version of the initialization produces the same value in the initialized object as would be produced by the dynamic initialization if all objects not required to be initialized statically were initialized dynamically.

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

```cpp
inline double fd() { return 1.0; }
extern double d1;
double d2 = d1;                      // unspecified:
                               // may be statically initialized to 0.0 or
                               // dynamically initialized to 1.0

double d1 = fd();                  // may be initialized statically to 1.0

— end note]

It is implementation-defined whether or not the dynamic initialization (8.5, 9.4, 12.1, 12.6.1) of an object of namespace scope 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 use of any function or object defined in the same translation unit as the object to be initialized.32) [Example:

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

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

// - File 3 -
#include "a.h"
#include "b.h"
extern A a;
```

32) An object defined in namespace scope having initialization with side-effects must be initialized even if it is not used (3.7.1).
extern B b;

int main() {
    a.Use();
    b.Use();
}

It is implementation-defined whether either a or b is initialized before main is entered or whether the initializations are delayed until a is first 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 used by the initialization of a, that is, before A::A is called. If, however, a is initialized at some point after the first statement of main, b will be initialized prior to its use in A::A. — end example

4 If construction or destruction of a non-local static object ends in throwing an uncaught exception, the result is to call std::terminate (18.7.3.3).

3.6.3 Termination [basic.start.term]

1 Destructors (12.4) for initialized objects of static storage duration (declared at block scope or at namespace scope) are called as a result of returning from main and as a result of calling std::exit (18.4). These objects are destroyed in the reverse order of the completion of their constructor or of the completion of their dynamic initialization. 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 local object with static storage duration initialized during the construction of the subobjects is destroyed.

2 If a function contains a local object of static storage duration that has been destroyed and the function is called during the destruction of an object with static storage duration, the program has undefined behavior if the flow of control passes through the definition of the previously destroyed local object.

3 If a function is registered with std::atexit (see <cstdlib>, 18.4) then following the call to std::exit, any objects with static storage duration initialized prior to the registration of that function shall not be destroyed until the registered function is called from the termination process and has completed. For an object with static storage duration constructed after a function is registered with std::atexit, then following the call to std::exit, the registered function is not called until the execution of the object's destructor has completed. If std::atexit is called during the construction of an object, the complete object to which it belongs shall be destroyed before the registered function is called.

4 Calling the function std::abort() declared in <cstdlib> terminates the program without executing destructors for objects of automatic or static storage duration and without calling the functions passed to std::atexit().

3.7 Storage duration [basic.stc]

1 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:

   — static storage duration
   — automatic storage duration
   — dynamic storage duration

Draft
Static 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 class specifiers \texttt{static} and \texttt{auto} are related to storage duration as described below.

The storage duration categories apply to references as well. The lifetime of a reference is its storage duration.

### 3.7.1 Static storage duration \([\text{basic.stc.static}]\)

1. All objects which neither have dynamic storage duration nor are local have \textit{static storage duration}. The storage for these objects shall last for the duration of the program (3.6.2, 3.6.3).

2. If an object of 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 may be eliminated as specified in 12.8.

3. The keyword \texttt{static} can be used to declare a local variable with static storage duration. \[\text{Note: 6.7 describes the initialization of local \texttt{static} variables; 3.6.3 describes the destruction of local \texttt{static} variables. — end note}\]

4. The keyword \texttt{static} applied to a class data member in a class definition gives the data member static storage duration.

### 3.7.2 Automatic storage duration \([\text{basic.stc.auto}]\)

1. Local objects explicitly declared \texttt{auto} or \texttt{register} or not explicitly declared \texttt{static} or \texttt{extern} have \textit{automatic storage duration}. The storage for these objects lasts until the block in which they are created exits.

2. \[\text{Note: these objects are initialized and destroyed as described in 6.7. — end note}\]

3. If a named automatic object 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 may be eliminated as specified in 12.8.

### 3.7.3 Dynamic storage duration \([\text{basic.stc.dynamic}]\)

1. Objects can be created dynamically during program execution (1.9), using \textit{new-expression}s (5.3.4), and destroyed using \textit{delete-expression}s (5.3.5). A C++ implementation provides access to, and management of, dynamic storage via the global \textit{allocation functions} \texttt{operator new} and \texttt{operator new[]} and the global \textit{deallocation functions} \texttt{operator delete} and \texttt{operator delete[]}.

2. The library provides default definitions for the global allocation and deallocation functions. Some global allocation and deallocation functions are replaceable (18.5.1). 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 (17.4.3.4). The following allocation and deallocation functions (18.5) are implicitly declared in global scope in each translation unit of a program.

```cpp
void* operator new(std::size_t) throw(std::bad_alloc);
void* operator new[](std::size_t) throw(std::bad_alloc);
void operator delete(void*) throw();
void operator delete[](void*) throw();
```
3.7 Storage duration

These implicit declarations introduce only the function names operator new, operator new[], operator delete, operator delete[]. [Note: the implicit declarations do not introduce the names std, std::bad_alloc, and 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, std::bad_alloc, and 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).

3 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 3.7.3.1 and 3.7.3.2.

3.7.3.1 Allocation functions

An allocation function shall be a class member function or a global function; a program is ill-formed if an allocation function is declared in a namespace scope other than global scope or declared static in global scope. The return type shall be void*. The first parameter shall have type std::size_t (18.1). 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.

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 different from any previously returned value unless that value was subsequently passed to an operator delete. The effect of dereferencing a pointer returned as a request for zero size is undefined.

3 An allocation function that fails to allocate storage can invoke the currently installed new-handler function (18.5.2.2), if any. [Note: A program-supplied allocation function can obtain the address of the currently installed new_handler using the std::set_new_handler function (18.5.2.3). —end note] If an allocation function declared with an empty exception-specification (15.4), throw(), 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 of a type that would match a handler (15.3) of type std::bad_alloc (18.5.2.1).

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 of type std::type_info (5.2.8), for the copy of an object thrown by a throw expression (15.1). —end note]

3.7.3.2 Deallocation functions

Deallocation functions shall be class member functions or global functions; a program is ill-formed if deallocation

---

33) 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. 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, 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.

The value of the first argument supplied to a deallocation functions 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 value supplied to `operator delete(void*)` in the standard library shall be 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 value supplied to `operator delete[](void*)` in the standard library shall be 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. The effect of using an invalid pointer value (including passing it to a deallocation function) is undefined.\(^{34}\)

3.7.4 **Duration of subobjects** \[basic.stc.inherit\]

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

3.8 **Object Lifetime** \[basic.life\]

The lifetime of an object is a runtime property of the object. The lifetime of an object of type `T` begins when:

- storage with the proper alignment and size for type `T` is obtained, and

- if `T` is a class type and the constructor invoked to create the object is non-trivial (12.1), the principal constructor call (12.6.2) has completed. \[Note: the initialization can be performed by a constructor call or, in the case of an aggregate with an implicitly-declared non-trivial default constructor, an aggregate initialization 8.5.1. \[— end note]\

The lifetime of an object of type `T` ends when:

- 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.

\(^{34}\) On some implementations, it causes a system-generated runtime fault.
3.8 Object Lifetime

[ Note: the lifetime of an array object or of an object of **PODtrivial** type (3.9) starts as soon as storage with proper size and alignment is obtained, and its lifetime ends when the storage which the array or object occupies is reused or released. 12.6.2 describes the lifetime of base and member subobjects.  — end note ]

2 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 ]

3 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.

4 Before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that refers to the storage location where the object will be or was located may be used but only in limited ways. Such a pointer refers to allocated storage (3.7.3.2), and using the pointer as if the pointer were of type void*, is well-defined. Such a pointer may be dereferenced but the resulting lvalue may only be used in limited ways, as described below. 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 program has undefined behavior. If the object will be or was of a non-PODtrivial class type, the program has undefined behavior if:

— 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 base class type, or
— the pointer is used as the operand of a static_cast (5.2.9) (except when the conversion is to void*, or to void* and subsequently to char*, or unsigned char*).

— 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();
};

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
}
```

35) For example, before the construction of a global object of non-POD class type (12.7).
\[
\ldots = \text{this}; \\
\text{\hspace{1cm}} \text{// OK, this points to valid memory}
\]

```cpp
void g() {
    void* p = std::malloc(sizeof(D1) + sizeof(D2));
    B* pb = new (p) D1;
    pb->mutate();
    \&pb; \\
    \text{\hspace{1cm}} \text{// OK: pb points to valid memory}
    void* q = pb;
    \text{\hspace{1cm}} \text{// OK: pb points to valid memory}
    pb->f();
    \text{\hspace{1cm}} \text{// undefined behavior, lifetime of } \ast pb \text{ 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 lvalue which refers to the original object may be used but only in limited ways. Such an lvalue refers to allocated storage (3.7.3.2), and using the properties of the lvalue which do not depend on its value is well-defined. If an lvalue-to-rvalue conversion (4.1) is applied to such an lvalue, the program has undefined behavior; if the original object will be or was of a non-\textit{POD trivial} class type, the program has undefined behavior if:

- the lvalue is used to access a non-static data member or call a non-static member function of the object, or
- the lvalue is implicitly converted (4.10) to a reference to a base class type, or
- the lvalue is used as the operand of a \texttt{static\_cast} (5.2.9) except when the conversion is ultimately to \texttt{cv char\&} or \texttt{cv unsigned char\&}, or
- the lvalue is used as the operand of a \texttt{dynamic\_cast} (5.2.7) or as the operand of \texttt{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, a reference that referred to the original object, or the name of the original object will automatically refer to the new object and, once the lifetime of the new object has started, can be used to manipulate the new object, if:

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

    ```cpp
    struct C {
        int i;
        void f();
        const C\& operator=( const C\& );
    };
    
    const C\& C::operator=( const C\& other)
    {
    
    Draft
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
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) or automatic (3.7.2) 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:

```cpp
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 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:

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

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

— end example ]

36) that is, an object for which a destructor will be called implicitly—either either upon exit from the block for an object with automatic storage duration or upon exit from the program for an object with static storage duration.
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 `POD` trivial 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 `POD` trivial type `T`, if two pointers to `T` point to distinct `T` objects `obj1` and `obj2`, where neither `obj1` nor `obj2` is a base-class subobject, if the value of `obj1` is copied into `obj2`, using the `std::memcpy` library function, `obj2` shall subsequently hold the same value as `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 trivial 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 `POD` trivial 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.38)

Object types have alignment requirements (3.9.1, 3.9.2). The alignment of a complete object type is an implementation-defined integer value representing a number of bytes, an object is allocated at an address that meets the alignment requirements of its object type.

A class that has been declared but not defined, 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

---

37) By using, for example, the library functions (17.4.1.2) `std::memcpy` or `std::memmove`.

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

39) The size and layout of an instance of an incompletely-defined object type is unknown.
type and therefore incomplete; if the class type is completed later on in the translation unit, the array type becomes complete; the array type at those two points is the same type. The declared type of an array object might be an array of unknown size and therefore be incomplete at one point in a translation unit and complete later on; the array types at those two points (“array of unknown bound of T” and “array of N T”) are different types. The type of a pointer to array of unknown size, or of a type defined by a `typedef` declaration to be an array of unknown size, cannot be completed.

[Example:

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 ]

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

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

10 Arithmetic types (3.9.1), enumeration types, pointer types, and pointer to member types (3.9.2), 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. Scalar types, trivial class types (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.

11 A type is a literal type if it is:

— a scalar type; or
— a class type (clause 9) with
  — a trivial copy constructor,
  — a trivial destructor,
  — at least one constexpr constructor other than the copy constructor,
  — no virtual base classes, and
  — all non-static data members and base classes of literal types; or
— an array of literal type.

If two types T1 and T2 are the same type, then T1 and T2 are layout-compatible types. [Note: Layout-compatible enumerations are described in 7.2. Layout-compatible POD-standard-layout structs and POD-standard-layout unions are described in 9.2. — end note]

3.9.1 Fundamental types [basic.fundamental]

1 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. 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 character types, all bits of the object representation participate in the value representation. For unsigned character types, all possible bit patterns of the value representation represent numbers. 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.

2 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.

3 For each of the standard signed integer types, there exists a corresponding (but different) standard unsigned integer type: “unsigned char”, “unsigned short int”, “unsigned int”, “unsigned long int”, and “unsigned long long int”, each of which occupies the same amount of storage and has the same alignment requirements (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 nonnegative 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

\[40\] that is, large enough to contain any value in the range of INT_MIN and INT_MAX, as defined in the header <climits>.

\[41\] See 7.1.6.2 regarding the correspondence between types and the sequences of type-specifier s that designate them.
collectively called the standard integer types, and the extended signed integer types and extended unsigned integer types are collectively called the extended integer types.

Unsigned integers, declared unsigned, 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.\(^{42}\)

Type \(\text{wchar}\_t\) is a distinct type whose values can represent distinct codes for all members of the largest extended character set specified among the supported locales (22.1.1). Type \(\text{wchar}\_t\) shall have the same size, signedness, and alignment requirements (3.11) as one of the other integral types, called its underlying type. Types \(\text{char16}\_t\) and \(\text{char32}\_t\) denote distinct types with the same size, signedness, and alignment as \(\text{uint}\_\text{least16}\_t\) and \(\text{uint}\_\text{least32}\_t\), respectively, in <\text{stdint.h}>, called the underlying types.

Values of type \(\text{bool}\) are either \text{true} or \text{false}.\(^{43}\) [Note: there are no signed, unsigned, short, or long \(\text{bool}\) types or values. — end note] As described below, \(\text{bool}\) values behave as integral types. Values of type \(\text{bool}\) participate in integral promotions (4.5).

Types \(\text{bool}\), \(\text{char}\), \(\text{char16}\_t\), \(\text{char32}\_t\), \(\text{wchar}\_t\), and the signed and unsigned integer types are collectively called integral types.\(^{44}\) A synonym for integral type is integer type. The representations of integral types shall define values by use of a pure binary numeration system.\(^{45}\) [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: \(\text{float}\), \(\text{double}\), and \(\text{long double}\). The type \(\text{double}\) provides at least as much precision as \(\text{float}\), and the type \(\text{long double}\) provides at least as much precision as \(\text{double}\). The set of values of the type \(\text{float}\) is a subset of the set of values of the type \(\text{double}\); the set of values of the type \(\text{double}\) is a subset of the set of values of the type \(\text{long double}\). The value representation of floating-point types is implementation-defined. Integral and floating types are collectively called arithmetic types. Specializations of the standard template \texttt{std::numeric_limits} (18.2) shall specify the maximum and minimum values of each arithmetic type for an implementation.

The \(\text{void}\) type has an empty set of values. The \(\text{void}\) 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 \(\text{cv}\) \(\text{void}\) (5.4). An expression of type \(\text{void}\) shall be used only as an expression statement (6.2), as an operand of a comma expression (5.18), as a second or third operand of \(?\) (5.16), as the operand of type \(\text{void}\), or as the expression in a return statement (6.6.3) for a function with the return type \(\text{void}\).

[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:

- arrays of objects of a given type, 8.3.4;
- functions, which have parameters of given types and return \(\text{void}\) or references or objects of a given type, 8.3.5;

\(^{42}\)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.

\(^{43}\)Using a \(\text{bool}\) value in ways described by this International Standard as “undefined,” such as by examining the value of an uninitialized automatic variable, might cause it to behave as if it is neither \text{true} nor \text{false}.

\(^{44}\)Therefore, enumerations (7.2) are not integral; however, enumerations can be promoted to integral types as specified in 4.5.

\(^{45}\)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.)
— *pointers* to `void` or objects or functions (including static members of classes) of a given type, 8.3.1;
— *references* to objects or functions of a given type, 8.3.2. There are two types of references:
  — *lvalue reference*
  — *rvalue reference*
— *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* 46) *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.

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. — end note] The value representation of pointer types is implementation-defined. Pointers to `cv`-qualified and `cv`-unqualified versions (3.9.3) of layout-compatible types shall have the same value representation and alignment requirements (3.11). [Note: pointers to over-aligned types 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]

Objects of `cv`-qualified (3.9.3) or `cv`-unqualified type `void*` (pointer to void), can be used to point to objects of unknown type. A `void*` shall be able to hold any object pointer. A `cv`-qualified or `cv`-unqualified (3.9.3) `void*` shall have the same representation and alignment requirements as a `cv`-qualified or `cv`-unqualified `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 when the object is created. The presence of a `const` specifier in a *decl-specifier-seq* declares an object of `const`-qualified *object type*; such object is called a `const object`. The presence of a `volatile` specifier in a *decl-specifier-seq* declares an object of `volatile`-qualified *object type*; such object is called a `volatile object`. The presence of both `cv`-qualifiers in a

46) Static class members are objects or functions, and pointers to them are ordinary pointers to objects or functions.
3.10 Lvalues and rvalues

Decl-specifier-seq declares an object of const-volatile-qualified object type; such object is called a const volatile object.
The cv-qualified or cv-unqualified versions of a type are distinct types; however, they shall have the same representation and alignment requirements (3.9).\(^{47}\)

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 Each non-static, non-mutable, non-reference data member of a const-qualified class object is const-qualified, each non-static, non-reference data member of a volatile-qualified class object is volatile-qualified and similarly for members of a const-volatile class. See 8.3.5 and 9.3.2 regarding cv-qualified function types.

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 8: relations on const and volatile

<table>
<thead>
<tr>
<th>Type</th>
<th>&lt;</th>
<th>&lt;</th>
<th>&lt;</th>
<th>&lt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>no cv-qualifer</td>
<td>const</td>
<td>volatile</td>
<td>const volatile</td>
<td>const volatile</td>
</tr>
</tbody>
</table>

5 In this International Standard, the notation cv (or cv1, cv2, etc.), used in the description of types, represents an arbitrary set of cv-qualifiers, i.e., one of \{const\}, \{volatile\}, \{const, volatile\}, or the empty set. Cv-qualifiers applied to an array type attach to the underlying element type, so the notation “cv T,” where T is an array type, refers to an array whose elements are so-qualified. Such array types can be said to be more (or less) cv-qualified than other types based on the cv-qualification of the underlying element types.

3.10 Lvalues and rvalues

[basic.lval]

1 Every expression is either an lvalue or an rvalue.

2 An lvalue refers to an object or function. Some rvalue expressions—those of (possibly cv-qualified) class or array type—also refer to objects.\(^{48}\)

3 [Note: some built-in operators and function calls yield lvalues. [Example: if E is an expression of pointer type, then \*E is an lvalue expression referring to the object to which E points. As another example, the function

```c
int& f();
```

yields an lvalue, so the call f() is an lvalue expression. — end example ] — end note ]

4 [Note: some built-in operators expect lvalue operands. [Example: built-in assignment operators all expect their left-hand operands to be lvalues. — end example ] Other built-in operators yield rvalues, and some expect them. [Example: the unary and binary + operators expect rvalue arguments and yield rvalue results. — end example ] The discussion of

\(^{47}\)The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

\(^{48}\)Expressions such as invocations of constructors and of functions that return a class type refer to objects, and the implementation can invoke a member function upon such objects, but the expressions are not lvalues.
each built-in operator in clause 5 indicates whether it expects lvalue operands and whether it yields an lvalue. —end note —

The result of calling a function that does not return an lvalue reference is an rvalue. User defined operators are functions, and whether such operators expect or yield lvalues is determined by their parameter and return types.

An expression which holds a temporary object resulting from a cast to a type other than an lvalue reference type is an rvalue (this includes the explicit creation of an object using functional notation (5.2.3)).

Whenever an lvalue appears in a context where an rvalue is expected, the lvalue is converted to an rvalue; see 4.1, 4.2, and 4.3.

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.

Class rvalues can have cv-qualified types; non-class rvalues always have cv-unqualified types. Rvalues shall always have complete types or the void type; in addition to these types, lvalues can also have incomplete types.

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 an lvalue of other than one of the following types the behavior is undefined.49)

— 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 members (including, recursively, a 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.

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

Draft
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 alignas (7.1.7).

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) (18.1).

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 (7.1.7). A type having an extended alignment requirement is an over-aligned type. [Note: every over-aligned type is or contains a class type with a non-static data member to which an extended alignment has been applied. —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.\(^{50}\)

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 types char, signed char, and unsigned char shall have the weakest alignment requirement. [Note: this enables the character types to be used as the underlying type for an aligned memory area (7.1.7). —end note]

Comparing alignments is meaningful and provides the obvious results:

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

[Note: the runtime pointer alignment function (20.6.7) can be used to obtain an aligned pointer within a buffer; the aligned-storage templates in the library (20.4.7) can be used to obtain aligned storage. —end note]

If a request for a specific extended alignment in a specific context is not supported by an implementation, the implementation may reject the request as ill-formed. The implementation may also silently ignore the requested alignment. [Note: additionally, a request for runtime allocation of dynamic memory for which the requested alignment cannot be honored may be treated as an allocation failure. —end note]

\(^{50}\)It is intended that every valid alignment value be an integral power of two.
Chapter 4  Standard conversions

1 Standard conversions are implicit conversions defined for built-in types. Clause 4 enumerates the full set of such conversions. A standard conversion sequence is a sequence of standard conversions in the following order:

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

   — 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.

   — 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:

   — When used as operands of operators. The operator’s requirements for its operands dictate the destination type (clause 5).

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

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

   — 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). The effect of the implicit conversion is the same as performing the 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 (8.3.2), and an rvalue otherwise. The expression e is used as an lvalue if and only if the initialization uses it as an lvalue.

4 [Note: For user-defined 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]

5 [Note: There are some contexts where certain conversions are suppressed. For example, the lvalue-to-rvalue conversion is not done on the operand of the unary & operator. Specific exceptions are given in the descriptions of those operators]
4.1 Lvalue-to-rvalue conversion

1 An lvalue (3.10) of a non-function, non-array type \( T \) can be converted to an rvalue. If \( T \) is an incomplete type, a program that necessitates this conversion is ill-formed. If the object to which the lvalue refers is not an object of type \( T \) and is not an object of a type derived from \( T \), or if the object is uninitialized, a program that necessitates this conversion has undefined behavior. If \( T \) is a non-class type, the type of the rvalue is the cv-unqualified version of \( T \). Otherwise, the type of the rvalue is \( T \).

2 When an lvalue-to-rvalue conversion occurs in an unevaluated operand or a subexpression thereof (clause 5) the value contained in the referenced object is not accessed. Otherwise, if the lvalue has a class type, the conversion copy-initializes a temporary of type \( T \) from the lvalue and the result of the conversion is an rvalue for the temporary. Otherwise, the value contained in the object indicated by the lvalue is the rvalue result.

[Note: See also 3.10. — end note]

4.2 Array-to-pointer conversion

1 An lvalue or rvalue of type “array of \( N \ T \)” or “array of unknown bound of \( T \)” can be converted to an rvalue of type “pointer to \( T \).” The result is a pointer to the first element of the array.

2 A string literal (2.13.4) with no prefix, with a u prefix, with a U prefix, or with an L prefix can be converted to an rvalue of type “pointer to char”, “pointer to char16_t”, “pointer to char32_t”, or “pointer to wchar_t”, respectively. In any case, the result is a pointer to the first element of the array. This conversion is considered only when there is an explicit appropriate pointer target type, and not when there is a general need to convert from an lvalue to an rvalue. [Note: this conversion is deprecated. See Annex D. — end note] For the purpose of ranking in overload resolution (13.3.3.1.1), this conversion is considered an array-to-pointer conversion followed by a qualification conversion (4.4). [Example: "abc" is converted to “pointer to const char” as an array-to-pointer conversion, and then to “pointer to char” as a qualification conversion. — end example]

4.3 Function-to-pointer conversion

1 An lvalue of function type \( T \) can be converted to an rvalue of type “pointer to \( T \).” The result is a pointer to the function.

[Note: See 13.4 for additional rules for the case where the function is overloaded. — end note]

4.4 Qualification conversions

1 An rvalue of type “pointer to \( cv1 \ T \)” can be converted to an rvalue of type “pointer to \( cv2 \ T \)” if “\( cv2 \ T \)” is more cv-qualified than “\( cv1 \ T \)”.

2 An rvalue of type “pointer to member of \( X \) of type \( cv1 \ T \)” can be converted to an rvalue of type “pointer to member of \( X \) of type \( cv2 \ T \)” if “\( cv2 \ T \)” is more cv-qualified than “\( cv1 \ T \)”.

[Note: Function types (including those used in pointer to member function types) are never cv-qualified (8.3.5). — end note]

51) In C++ class rvalues can have cv-qualified types (because they are objects). This differs from ISO C, in which non-lvalues never have cv-qualified types.

52) This conversion never applies to non-static member functions because an lvalue that refers to a non-static member function cannot be obtained.
A conversion can add cv-qualifiers at levels other than the first in multi-level pointers, subject to the following rules:

Two pointer types $T_1$ and $T_2$ are similar if there exists a type $T$ and integer $n > 0$ such that:

$T_1$ is $cv_{1,0}$ pointer to $cv_{1,1}$ pointer to ... $cv_{1,n-1}$ pointer to $cv_{1,n} T$

and

$T_2$ is $cv_{2,0}$ pointer to $cv_{2,1}$ pointer to ... $cv_{2,n-1}$ pointer to $cv_{2,n} T$

where each $cv_{1,j}$ is const, volatile, const volatile, or nothing. The n-tuple of cv-qualifiers after the first in a pointer type, e.g., $cv_{1,1}$, $cv_{1,2}$, ... $cv_{1,n}$ in the pointer type $T_1$, is called the cv-qualification signature of the pointer type. An expression of type $T_1$ can be converted to type $T_2$ if and only if the following conditions are satisfied:

- the pointer types are similar.
- for every $j > 0$, if const is in $cv_{1,j}$ then const is in $cv_{2,j}$, and similarly for volatile.
- if the $cv_{1,j}$ and $cv_{2,j}$ are different, then const is in every $cv_{2,k}$ for $0 < k < j$.

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

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

—is end note ]

A multi-level pointer to member type, or a multi-level mixed pointer and pointer to member type has the form:

$cv_{0}P_0$ to $cv_{1}P_1$ to ... $cv_{n-1}P_{n-1}$ to $cv_{n} T$

where $P_i$ is either a pointer or pointer to member and where $T$ is not a pointer type or pointer to member type.

Two multi-level pointer to member types or two multi-level mixed pointer and pointer to member types $T_1$ and $T_2$ are similar if there exists a type $T$ and integer $n > 0$ such that:

$T_1$ is $cv_{1,0}P_0$ to $cv_{1,1}P_1$ to ... $cv_{1,n-1}P_{n-1}$ to $cv_{1,n} T$

and

$T_2$ is $cv_{2,0}P_0$ to $cv_{2,1}P_1$ to ... $cv_{2,n-1}P_{n-1}$ to $cv_{2,n} T$

For similar multi-level pointer to member types and similar multi-level mixed pointer and pointer to member types, the rules for adding cv-qualifiers are the same as those used for similar pointer types.

53) These rules ensure that const-safety is preserved by the conversion.
4.5 Integral promotions

4.5 Integral promotions

1 An rvalue of an integer type other than bool, char16_t, char32_t, or wchar_t whose integer conversion rank (4.13) is less than the rank of int can be converted to an rvalue of type int if int can represent all the values of the source type; otherwise, the source rvalue can be converted to an rvalue of type unsigned int.

2 An rvalue of type char16_t, char32_t, or wchar_t (3.9.1) can be converted to an rvalue of the first of the following types that can represent all the values of its underlying type: int, unsigned int, long int, unsigned long int, long long int, or unsigned long long int. If none of the types in that list can represent all the values of its underlying type, an rvalue of type char16_t, char32_t, or wchar_t can be converted to an rvalue of its underlying type. An rvalue of an unscoped enumeration type (7.2) can be converted to an rvalue of the first of the following types that can represent all the values of the enumeration (i.e. the values in the range $b_{min}$ to $b_{max}$ 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, an rvalue of an unscoped enumeration type can be converted to an rvalue 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.

3 An rvalue for an integral bit-field (9.6) can be converted to an rvalue 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.

4 An rvalue of type bool can be converted to an rvalue of type int, with false becoming zero and true becoming one.

5 These conversions are called integral promotions.

4.6 Floating point promotion

4.6 Floating point promotion

1 An rvalue of type float can be converted to an rvalue of type double. The value is unchanged.

2 This conversion is called floating point promotion.

4.7 Integral conversions

4.7 Integral conversions

1 An rvalue of an integer type can be converted to an rvalue of another integer type. An rvalue of an unscoped enumeration type can be converted to an rvalue 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 (and bit-field width); 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

4.8 Floating point conversions

1 An rvalue of floating point type can be converted to an rvalue of another floating point type. If the source value can be
4.9 Floating-integral conversions

An rvalue of a floating point type can be converted to an rvalue 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]

An rvalue of an integer type or of an unscoped enumeration type can be converted to an rvalue of a floating point type. The result is exact if possible. Otherwise, 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 source type is bool, the value false is converted to zero and the value true is converted to one.

4.10 Pointer conversions

A null pointer constant is an integral constant expression (5.19) rvalue of integer type that evaluates to zero. 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 pointer to object or pointer to function type. 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).

An rvalue of type “pointer to cv T,” where T is an object type, can be converted to an rvalue of type “pointer to cv void”. The result of converting a “pointer to cv T” to a “pointer to cv void” points to the start of the storage location where the object of type T resides, as if the object is a most derived object (1.8) of type T (that is, not a base class subobject). The null pointer value is converted to the null pointer value of the destination type.

An rvalue of type “pointer to cv D”, where D is a class type, can be converted to an rvalue 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

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. 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).

An rvalue of type “pointer to member of B of type cv T”, where B is a class type, can be converted to an rvalue 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”, it can be dereferenced.
4.12 Boolean conversions

An rvalue of arithmetic, unscoped enumeration, pointer, or pointer to member type can be converted to an rvalue 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.

4.13 Integer conversion rank

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

- No two signed integer types shall have the same rank, even if they have the same representation.
- The rank of a signed integer type shall be greater than the rank of any signed integer type with a smaller size.
- 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.
- The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type.
- The rank of any standard integer type shall be greater than the rank of any extended integer type with the same size.
- The rank of char shall equal the rank of signed char and unsigned char.
- The rank of bool shall be less than the rank of all other standard integer types.
- The ranks of char16_t, char32_t, and wchar_t shall equal the ranks of their underlying types (3.9.1).
- 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.
- 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 (5). — end note]
Chapter 5  Expressions

[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]

[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, lvalue, 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]

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.

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, unless such an expression appears where an integral constant expression is required (5.19), in which case the program is ill-formed. [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]

If an expression initially has the type “lvalue 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 lvalue reference, and the expression is an lvalue.

If an expression initially has the type “rvalue reference to T” (8.3.2, 8.5.3), the type is adjusted to “T” prior to any further analysis, and the expression designates the object or function denoted by the rvalue reference. If the expression is the result of calling a function, whether implicitly or explicitly, it is an rvalue; otherwise, it is an lvalue. [Note: In general, the effect of this rule is that named rvalue references are treated as lvalues and unnamed rvalue references are treated as rvalues. — end note]

[Example:

```cpp
struct A {
    A&& operator+(A, A);
    A&& f();
};
```]

55) The precedence of operators is not directly specified, but it can be derived from the syntax.
\begin{verbatim}
A a;
A&& ar = a;
\end{verbatim}

The expressions \( f() \) and \( a + a \) are rvalues of type \( A \). The expression \( ar \) is an lvalue of type \( A \). — end example ]

7 An expression designating an object is called an \textit{object-expression}.

8 \textbf{Clause 5 specifies for some operators that some of their operands are unevaluated operands.} In some contexts, unevaluated operands appear (5.2.8, 5.3.3, 7.1.6.2). An unevaluated operand is not evaluated. \[ Note: \] 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 an lvalue expression appears as an operand of an operator that expects an rvalue 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 an rvalue. \[ Note: \] because cv-qualifiers are removed from the type of an expression of non-class type when the expression is converted to an rvalue, an lvalue expression of type \texttt{const int} can, for example, be used where an rvalue expression of type \texttt{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 \textit{usual arithmetic conversions}, which are defined as follows:

---

11 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.\[57]\)

\[56]\) As a consequence, operands of type \texttt{bool}, \texttt{char16_t}, \texttt{char32_t}, \texttt{wchar_t}, or an enumerated type are converted to some integral type.

\[57]\) The cast and assignment operators must still perform their specific conversions as described in \texttt{5.4, 5.2.9 and 5.17.}

Draft
5.1 Primary expressions

Primary expressions are literals, names, and names qualified by the scope resolution operator ::.

- **primary-expression:**
  - literal
  - this
  - ( expression )
  - id-expression

- **id-expression:**
  - unqualified-id
  - qualified-id

- **unqualified-id:**
  - identifier
  - operator-function-id
  - conversion-function-id
  - ∼ class-name
  - template-id

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

The keyword **this** names a pointer to the object for which a non-static member function (9.3.2) is invoked. The keyword **this** shall be used only inside a non-static class member function body (9.3) or in a constructor or non-initializer (12.6.2). The type of the expression is a pointer to the function’s class (9.3.2), possibly with cv-qualifiers on the class type. The expression is an rvalue.

The operator :: followed by an identifier, a qualified-id, or an operator-function-id is a primary-expression. Its type is specified by the declaration of the identifier, qualified-id, or operator-function-id. The result is the entity denoted by the identifier, qualified-id, or operator-function-id. The result is an lvalue if the entity is a function or variable. The identifier, qualified-id, or operator-function-id shall have global namespace scope or be visible in global scope because of a using-directive (7.3.4). [Note: the use of :: allows a type, an object, a function, an enumerator, or a namespace declared in the global namespace to be referred to even if its identifier has been hidden (3.4.3). — end note]

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-id s, see 13.5; for conversion-function-id s, see 12.3.2; for template-id s, see 14.2. A class-name 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.
5.1 Primary expressions

Expressions 78

qualified-id:
  ::opt nested-name-specifier templateopt unqualified-id
  :: identifier
  :: operator-function-id
  :: template-id

nested-name-specifier:
  type-name ::
  namespace-name ::
  nested-name-specifier identifier ::
  nested-name-specifier templateopt simple-template-id ::

A nested-name-specifier that names 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. [Note: a class member can be referred to using a qualified-id at any point in its potential scope (3.3.6). —end note] Where class-name :: class-name is used, and the two class-name s refer to the same class, this notation names the constructor (12.1). Where class-name :: ~ class-name is used, the two class-name s shall refer to the same class; this notation names the destructor (12.4). [Note: a typedef-name that names a class is a class-name (9.1). —end note]

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.

A nested-name-specifier that names 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 an rvalue.

In a qualified-id, if the id-expression 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.

An id-expression that denotes a non-static data member or non-static member function of a class can only be used:
  — as part of a class member access (5.2.5) in which the object-expression refers to the member’s class or a class derived from that class, or
  — to form a pointer to member (5.3.1), or
  — in the body of a non-static member function of that class or of a class derived from that class (9.3.1), or
  — in a mem-initializer for a constructor for that class or for a class derived from that class (12.6.2), or
  — if that id-expression denotes a non-static data member and it is the sole constituent of an unevaluated operand, except for optional enclosing parentheses. [Example:
    struct S {
        int m;
    };
    int i = sizeof(S::m); // OK
    int j = sizeof(S::m + 42); // error: reference to non-static member in subexpression

—end example]
5.2 Postfix expressions

Postfix expressions group left-to-right.

postfix-expression:

primary-expression
postfix-expression [ expression ]
postfix-expression ( expression-list_opt )
simple-type-specifier ( expression-list_opt )
typename-specifier ( expression-list_opt )
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 )
typeid ( expression )
typeid ( type-id )

expression-list:

assignment-expression . . . opt
expression-list , assignment-expression . . . opt

pseudo-destructor-name:

:: opt nested-name-specifier_opt type-name :: ~ type-name
:: opt nested-name-specifier template simple-template-id :: ~ type-name
:: opt nested-name-specifier_opt ~ type-name

[ 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 ]

An assignment-expression followed by an ellipsis is a pack expansion (14.5.3).

5.2.1 Subscripting

A postfix expression followed by an expression in square brackets is a postfix expression. One of the expressions shall have the type "pointer to T" and the other shall have enumeration or integral type. The result is an lvalue of type "T." The type "T" shall be a completely-defined object type.58) The expression E1[E2] is identical (by definition) to *(E1)+(E2)) [ Note: see 5.3 and 5.7 for details of * and + and 8.3.4 for details of arrays. — end note ]

5.2.2 Function call

There are two kinds of function call: ordinary function call and member function59) (9.3) call. A function call is a postfix expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the arguments to the function. For an ordinary function call, 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

58) This is true even if the subscript operator is used in the following common idiom: &x[0].
59) A static member function (9.4) is an ordinary function.
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 member function call, the postfix expression shall be an implicit (9.3.1, 9.4) or explicit class member access (5.2.5) whose id-expression is a function member name, or a pointer-to-member expression (5.5) selecting a function member. The first expression in the postfix expression is then called the object expression, and the call is as a member of the object pointed to or referred to. In the case of an implicit class member access, the implied object is the one pointed to by this.

[Note: a member function call of the form `f()` is interpreted as `(*this).f()` (see 9.3.1). — end note] If a function or member function name is used, the name can be overloaded (clause 13), in which case the appropriate function shall be selected according to the rules in 13.3. The function called in a member function call is normally selected according to the static type of the object expression (clause 10), but if that function is virtual and is not specified using a qualified-id then the function actually called will be the final overrider (10.3) of the selected function in the dynamic type of the object expression. [Note: the dynamic type is the type of the object pointed or 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: 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 The type of the function call expression is the return type of the statically chosen function (i.e., ignoring the virtual keyword), even if the type of the function actually called is different. This type shall be a complete object type, a reference type or the type `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. If the function is a non-static member function, the 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: 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: 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: the access of the constructor, conversion functions or destructor is checked at the point of call in the calling function. If a constructor or destructor for a function parameter throws an exception, the search for a handler starts in the scope of the calling function; in particular, if the function called has a function-try-block (clause 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.

5 [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, `const_cast` is required to be used to cast away the constness in order to modify the argument’s value. Where a parameter is of const reference type a temporary object is introduced if needed (7.1.6, 2.13, 2.13.4, 8.3.4, 12.2). In addition, it is possible to modify the values of nonconstant objects through pointer parameters. — end note]

6 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]

7 When there is no parameter for a given argument, the argument is passed in such a way that the receiving function can obtain the value of the argument by invoking va_arg (18.8). 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. After these conversions, if the argument does not have arithmetic, enumeration, pointer, pointer to member, or class type, the program is ill-formed. Passing an argument of non-POD trivial class type (clause 9) 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 argument expressions are all unsequenced relative to one another. All side effects of argument expression evaluations are sequenced before the function is entered (see 1.9). — end note]

9 Recursive calls are permitted, except to the function named main (3.6.1).

10 A function call is an lvalue if and only if the result type is an lvalue reference.

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

1 A simple-type-specifier (7.1.6) 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 simple-type-specifier specifies 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 \( T(x_1, x_2, ...) \) is equivalent in effect to the declaration \( T \ t(x_1, x_2, ...) \); for some invented temporary variable \( t \), with the result being the value of \( t \) as an rvalue.

2 The expression \( T() \), where \( T \) is a simple-type-specifier (7.1.6.2) for a non-array complete object type or the (possibly cv-qualified) void type, creates an rvalue of the specified type, which is value-initialized (8.5; no initialization is done for the void() case). [Note: if \( T \) is a non-class type that is cv-qualified, the cv-qualifiers are ignored when determining the type of the resulting rvalue (3.10). — end note]

5.2.4 Pseudo destructor call [expr.pseudo]

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

2 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-name s in a pseudo-destructor-name of the form

\[ : : opt \ \text{nested-type-specifier} \ \text{opt} \ \text{type-name} : : \sim \ \text{type-name} \]
shall designate the same scalar type.

5.2.5 Class member access

A postfix expression followed by a dot . or an arrow ->, optionally followed by the keyword template (14.8.1), 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, determine the result of the entire postfix expression.

For the first option (dot) the type of the first expression (the object expression) shall be “class object” (of a complete type). For the second option (arrow) the type of the first expression (the pointer expression) shall be “pointer to class object” (of a complete type). In these cases, 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 -> operators. — end note]

If \(E1\) has the type “pointer to class \(X\),” then the expression \(E1->E2\) is converted to the equivalent form \((*(E1)).E2\); the remainder of 5.2.5 will address only the first option (dot). Abbreviating object-expression.id-expression as \(E1.E2\), then the type and lvalue properties of this expression are determined as follows. In the remainder of 5.2.5, \(cq\) represents an arbitrary set of cv-qualifiers, as defined in 3.9.3.

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

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

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

— If \(E2\) is a (possibly overloaded) member function, function overload resolution (13.3) is used to determine whether \(E1.E2\) refers to a static or a non-static member function.

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

— Otherwise, if \(E1.E2\) refers to a non-static member function, and the type of \(E2\) is “function of parameter-type-list \(cv\) returning \(T\)”, then \(E1.E2\) is not an lvalue. 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 \(E1.E2\) is “function of parameter-type-list \(cv\) returning \(T\)”.

---

60) This 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.

61) Note that if \(E1\) has the type “pointer to class \(X\),” then \((\ast(E1))\) is an lvalue.
— If E2 is a nested type, the expression E1.E2 is ill-formed.

— If E2 is a member enumerator, and the type of E2 is T, the expression E1.E2 is not an lvalue. The type of E1.E2 is T.

If E2 is a non-static data member or a non-static member function, the program is ill-formed if the class of which E2 is directly a member is an ambiguous base (10.2) of the naming class (11.2) of E2.

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 bool, in which case it is set to 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 an rvalue. The type of the result is the cv-unqualified version of the type of the operand. See also 5.7 and 5.17.

The operand of postfix -- is decremented analogously to the postfix ++ operator, except that the operand shall not be of type 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.” Types shall not be defined in a dynamic_cast. The dynamic_cast operator shall not cast away constness (5.2.11).

If T is a pointer type, v shall be an rvalue of a pointer to complete class type, and the result is an rvalue 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 rvalue of the type referred to by T.

If the type of v is the same as the required result type (which, for convenience, will be called R in this description), or it is the same as R except that the class object type in R 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 R.

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. 62) The result is an lvalue if T is an lvalue reference, or an rvalue if T is an rvalue reference. In both the pointer and reference cases, cv1 shall be the same cv-qualification as, or greater cv-qualification than, cv2, and B shall be an accessible unambiguous base class of D.

[Example:

```c
struct B {};  
struct D : B {};  
```

62) The most derived object (1.8) pointed or referred to by v can contain other B objects as base classes, but these are ignored.

Draft
5.2 Postfix expressions

Expressions 84

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

— end example —

6 Otherwise, v shall be a pointer to or an lvalue of a polymorphic type (10.3).

7 If T is “pointer to cv void,” then the result is a pointer to the most derived object pointed to by v. Otherwise, a 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.

8 The run-time check logically executes as follows:

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

— 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 T, that is unambiguous and public, the result is a pointer (an lvalue referring) to the T subobject of the most derived object.

— Otherwise, the run-time check fails.

9 The value of a failed cast to pointer type is the null pointer value of the required result type. A failed cast to reference type throws std::bad_cast (18.6.2).

[Example:

class A { virtual void f(); };  
class B { virtual void g(); };  
class D : public virtual A, private B {};  
void g()
{
    D d;  
    B* bp = (B*)&d; // cast needed to break protection
    A* ap = &d;  // public derivation, no cast needed
    D& dr = dynamic_cast<D&>(*bp); // fails
    ap = dynamic_cast<A*>(bp); // fails
    bp = dynamic_cast<B*>(ap); // fails
    ap = dynamic_cast<A*>(&d); // succeeds
    bp = dynamic_cast<B*>(&d); // fails
}

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
}
5.2 Postfix expressions

```cpp
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` (18.6.1) and dynamic type `const std::type_info` or `const name` where `name` is an implementation-defined class derived from `std::type_info` which preserves the behavior described in 18.6.1. 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 an lvalue 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 lvalue refers. If the lvalue 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 the `std::bad_typeid` exception (18.6.3).

3 When `typeid` is applied to an expression other than an lvalue 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. Types shall not be defined in the `type-id`.

5 The top-level cv-qualifiers of the lvalue expression or the `type-id` that is the operand of `typeid` are always ignored.

```cpp
    class D { ... };  
    D d1;            
    const D d2;      
    typeid(d1)  == typeid(d2);  // yields true  
    typeid(D)  == typeid(const D); // yields true  
    typeid(D)  == typeid(d2);   // yields true  
    typeid(D)  == typeid(const D&); // yields true
```

— end example ]

6 If the header `<typeinfo>` (18.6.1) is not included prior to a use of `typeid`, the program is ill-formed.

63) The recommended name for such a class is `extended_type_info`.

64) If `p` is an expression of pointer type, then `*p`, `(p)`, `(*p)`, `(*p)`, `((p))`, and so on all meet this requirement.
5.2 Postfix expressions

5.2.9 Static cast

The result of the expression \( \text{static	extunderscore cast}\langle T\rangle(v) \) is the result of converting the expression \( v \) to type \( T \). If \( T \) is an lvalue reference type, the result is an lvalue; otherwise, the result is an rvalue. Types shall not be defined in a \text{static	extunderscore cast}. The \text{static	extunderscore cast} operator shall not cast away constness (5.2.11).

An lvalue of type “\text{cv}\1 B,” where \( B \) is a class type, can be cast to type “\text{reference to cv}\2 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 \text{cv}-qualification as, or greater \text{cv}-qualification than, \( cv1 \), and \( B \) is neither a virtual base class of \( D \) nor a base class of a \text{virtual base class} of \( D \). The result has type “\text{cv}\2 D.” It is an lvalue if the type cast to is an lvalue reference; otherwise, it is an rvalue. An rvalue of type “\text{cv}\1 B” may be cast to type “\text{rvalue reference to cv}\2 D” with the same constraints as for an lvalue of type “\text{cv}\1 B.” The result is an rvalue. If the object of type “\text{cv}\1 B” is actually a subobject of an object of type \( D \), the result refers to the enclosing object of type \( D \). Otherwise, the result of the cast is undefined. [Example:]

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

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

Otherwise, an expression \( e \) can be explicitly converted to a type \( T \) using a \text{static	extunderscore cast} of the form \text{static	extunderscore cast}\langle T\rangle(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 result is an lvalue if \( T \) is an lvalue reference type (8.3.2), and an rvalue otherwise. The expression \( e \) is used as an lvalue if and only if the initialization uses it as an lvalue.

Otherwise, the \text{static	extunderscore cast} shall perform one of the conversions listed below. No other conversion shall be performed explicitly using a \text{static	extunderscore cast}.

Any expression can be explicitly converted to type \text{cv} \text{void}. The expression value is discarded. [Note: however, if the value is in a temporary variable (12.2), the destructor for that variable is not executed until the usual time, and the value of the variable is preserved for the purpose of executing the destructor. —end note] The \text{Ivalue-to-rvalue} (4.1), \text{array-to-pointer} (4.2), and \text{function-to-pointer} (4.3) standard conversions are not applied to the expression.

The inverse of any standard conversion sequence (clause 4), other than the \text{lvalue-to-rvalue} (4.1), \text{array-to-pointer} (4.2), \text{function-to-pointer} (4.3), and \text{boolean} (4.12) conversions, can be performed explicitly using \text{static	extunderscore cast}. A program is ill-formed if it uses \text{static	extunderscore cast} to perform the inverse of an ill-formed standard conversion sequence. [Example:]

```c
struct B {}
struct D : private B {}
void f() {
    static_cast<D*>(B*)0); // Error: B is a private base of D.
    static_cast<int B::* >((int D::* )0); // Error: B is a private base of D.
}
```

—end example]
The 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 integral or enumeration type can be explicitly converted to an enumeration type. The value is unchanged if the original value is within the range of the enumeration values (7.2). Otherwise, the resulting enumeration value is unspecified.

An rvalue of type “pointer to `cv1` `B`,” where `B` is a class type, can be converted to an rvalue 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 rvalue 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 result of the cast is undefined.

An rvalue of type “pointer to member of `D` of type `cv1` `T`” can be converted to an rvalue 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 class `B` contains the original member, or is a base or derived class of the class containing the original member, the resulting pointer to member points to the original member. Otherwise, the result of the cast is undefined. [Note: although class `B` need not contain the original member, the dynamic type of the object on which the pointer to member is dereferenced must contain the original member; see 5.5. — end note]

An rvalue of type “pointer to `cv1` `void`” can be converted to an rvalue 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. 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

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, the result is an lvalue; otherwise, the result is an rvalue and the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the the expression `v`. Types shall not be defined in a `reinterpret_cast`. Conversions that can be performed explicitly using `reinterpret_cast` are listed below. No other conversion can be performed explicitly using `reinterpret_cast`.

The `reinterpret_cast` operator shall not cast away constness. [Note: see 5.2.11 for the definition of “casting away constness”. Subject to the restrictions in this section, an expression may be cast to its own type using a `reinterpret_cast` operator. — end note]

The mapping performed by `reinterpret_cast` is implementation-defined. [Note: it might, or might not, produce a representation different from the original value. — end note]
5.2 Postfix expressions

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 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.

A pointer to a function can be explicitly converted to a pointer to a function 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 an rvalue 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]

A pointer to an object can be explicitly converted to a pointer to an object of different type. Except that converting an rvalue 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, the result of such a pointer conversion is unspecified.

Converting a pointer to a function into a pointer to an object 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 an rvalue of one type to the other type and back, possibly with different cv-qualification, shall yield the original pointer value.

The null pointer value (4.10) is converted to the null pointer value of the destination type. [Note: A null pointer constant, which has integral type, is not necessarily converted to a null pointer value. — end note]

An rvalue of type “pointer to member of X of type T1” can be explicitly converted to an rvalue of 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:

- converting an rvalue 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.

- converting an rvalue 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.

An lvalue 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. That is, 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)). The result refers to the same object as the source lvalue, but with a different type. The result is an lvalue for lvalue references or an rvalue for rvalue references. No temporary is created, no copy is made, and constructors (12.1) or conversion functions (12.3) are not called.

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

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

This is sometimes referred to as a type pun.
5.2.11 Const cast

The result of the expression `const_cast<T>(v)` is of type T. If T is an lvalue reference type, the result is an lvalue; otherwise, the result is an rvalue 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. Types shall not be defined in a `const_cast`. 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

3 For two pointer types T1 and T2 where

\[ T1 \text{ is } cv_{1,0} \text{ pointer to } cv_{1,1} \text{ pointer to } \cdots \text{cv}_{1,n-1} \text{ pointer to } cv_{1,n} T \]

and

\[ T2 \text{ is } cv_{2,0} \text{ pointer to } cv_{2,1} \text{ pointer to } \cdots \text{cv}_{2,n-1} \text{ pointer to } cv_{2,n} T \]

where T is any object type or the void type and where cv_{1,k} and cv_{2,k} may be different cv-qualifications, an rvalue of type T1 may be explicitly converted to the type T2 using a `const_cast`. The result of a pointer `const_cast` refers to the original object.

4 An lvalue of type T1 can be explicitly converted to an lvalue of type T2 using the cast `const_cast<T2&>` (where T1 and T2 are object types) if a pointer to T1 can be explicitly converted to the type “pointer to T2” using a `const_cast`. Similarly, for two object types T1 and T2, an expression of type T1 can be explicitly converted to an rvalue of type T2 using the cast `const_cast<T2&&>` if a pointer to T1 can be explicitly converted to the type “pointer to T2” using a `const_cast`. The result of a reference `const_cast` refers to the original object.

5 For a `const_cast` involving pointers to data members, multi-level pointers to data members and multi-level mixed pointers and pointers to data members (4.4), the rules for `const_cast` are the same as those used for pointers; the “member” aspect of a pointer to member is ignored when determining where the cv-qualifiers are added or removed by the `const_cast`. The result of a pointer to data member `const_cast` refers to the same member as the original (unconst) pointer to data member.

6 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 `const_cast` that casts away a const-qualifier may produce undefined behavior (7.1.6.1). — end note

8 The following rules define the process known as casting away constness. In these rules Tn and Xn represent types. For two pointer types:

\[ X1 \text{ is } T1cv_{1,1} \ast \cdots \ast cv_{1,N} \ast \text{ where } T1 \text{ is not a pointer type} \]

\[ X2 \text{ is } T2cv_{2,1} \ast \cdots \ast cv_{2,M} \ast \text{ where } T2 \text{ is not a pointer type} \]

\[ K \text{ is } \min(N,M) \]

\(\text{const\_cast}\) is not limited to conversions that cast away a const-qualifier.
5.3 Unary expressions

Expressions with unary operators group right-to-left.

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

unary-operator: one of

```
* & + - ! ~
```

5.3.1 Unary operators

The unary * operator performs indirection: the expression to which it is applied shall be a pointer to an object type, or a pointer to a function type and the result is an lvalue referring to the object or function to which the expression points. If the type of the expression is "pointer to T," the type of the result is "T." [ Note: a pointer to an incomplete type (other than cv void) can be dereferenced. The lvalue thus obtained can be used in limited ways (to initialize a reference, for example); this lvalue must not be converted to an rvalue, see 4.1. — end note ]
The result of the unary & operator is a pointer to its operand. The operand shall be an lvalue or a qualified-id. In the first case, if the type of the expression is "T," the type of the result is “pointer to T.” In particular, the address of an object of type “cv T” is “pointer to cv T,” with the same cv-qualifiers. For a qualified-id, if the member is a static member of type “T,” the type of the result is plain “pointer to T.” If the member is a non-static member of class C of type T, the type of the result is “pointer to member of class C of type T.” [Example:

```c
struct A { int i; };
struct B : A { };
... &B::i ...
// has type int A::*
```

— 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 ]

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 ]

The address of an object of incomplete type can be taken, but if the complete type of that object is a class type that declares operator&() as a member function, then the behavior is undefined (and no diagnostic is required). The operand of & shall not be a bit-field.

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 ]

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

The operand of the unary - operator shall have arithmetic or enumeration type and the result is the negation of its operand. Integral promotion is performed on integral or enumeration operands. The negative of an unsigned quantity is computed by subtracting its value from 2
\(n\), where \(n\) is the number of bits in the promoted operand. The type of the result is the type of the promoted operand.

The operand of the logical negation operator ! is implicitly 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.

The operand of ~ shall have integral or 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. The ambiguity is resolved in favor of treating "~" as a unary complement rather than treating "X" as referring to a destructor.

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
5.3 Unary expressions

not of type bool, the expression ++x is equivalent to x+=1 [Note: see the discussions of addition (5.7) and assignment operators (5.17) for information on conversions. — end note]

2 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

1 The sizeof operator yields the number of bytes in the object representation of its operand. The operand is either an expression, which is an unevaluated operand (clause 5), or a parenthesized type-id. The sizeof operator shall not be applied to an expression that has function or incomplete type, or to an enumeration type before all its enumerators have been declared, or to the parenthesized name of such types, or to an lvalue 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]

2 When applied to a reference or a reference type, the result is the size of the referenced type. When applied to a class, the result is the number of bytes in an object of that class including any padding required for placing objects of that type in an array. The size of a most derived class shall be greater than zero (1.8). The result of applying sizeof to a base class subobject is the size of the base class type. When applied to an array, the result is the total number of bytes in the array. This implies that the size of an array of n elements is n times the size of an element.

3 The sizeof operator can be applied to a pointer to a function, but shall not be applied directly to a function.

4 The lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are not applied to the operand of sizeof.

5 Types shall not be defined in a sizeof expression.

6 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. The parameter pack is expanded (14.5.3) by the sizeof... operator. [Example:

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

— end example ]
```

7 The result of sizeof and sizeof... is a constant of type std::size_t. [Note: std::size_t is defined in the standard header <cstdlib> (18.1). — end note]

5.3.4 New

1 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

---

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

71) 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.

Draft
thereof (1.8, 3.9, 10.4). It is implementation-defined whether over-aligned types are supported (3.11). [Note: because references are not objects, references cannot be created by new-expression s. — 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 )

ew-type-id:
 type-specifier-seq new-declarator_opt

new-declarator:
 ptr-operator new-declarator_opt
direct-new-declarator

direct-new-declarator:
[ expression ]
direct-new-declarator [ constant-expression ]

new-initializer:
( expression-list_opt )

Entities created by a new-expression have dynamic storage duration (3.7.3). [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 the auto type-specifier appears in the type-specifier-seq of a new-type-id or type-id of a new-expression, the type-specifier-seq shall contain no other type-specifiers except cv-qualifiers, and 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 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-declarator s. [Note: this prevents ambiguities between declarator operators &amp;, *, [], 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:

Draft
5.3 Unary expressions  

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

is ill-formed because the binding is

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

Instead, the explicitly parenthesized version of the `new` operator can be used to create objects of compound types (3.9.2):

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

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

5 The `type-specifier-seq` shall not contain class declarations, or enumeration declarations.

6 When the allocated object is an array (that is, the `direct-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 ]

7 Every `constant-expression` in a `direct-new-declarator` shall be an integral constant expression (5.19) and evaluate to a strictly positive value. The `expression` in a `direct-new-declarator` shall be of integral type, enumeration type, or a class type for which a single conversion function to integral or enumeration type exists (12.3). If the expression is of class type, the expression is converted by calling that conversion function, and the result of the conversion is used in place of the original expression. If the value of the expression is negative, the behavior is undefined. [ Example: if `n` is a variable of type `int`, then given the definition `int n = 42`, `new float[n][5]` is well-formed (because `n` is the `expression` of a `direct-new-declarator`), but `new float[5][n]` is ill-formed (because `n` is not a `constant-expression`). If `n` is negative, the effect of `new float[n][5]` is undefined. — end example ]

8 When the value of the `expression` in a `direct-new-declarator` is zero, the allocation function is called to allocate an array with no elements.

9 A `new-expression` obtains storage for the object by calling an `allocation function` (3.7.3.1). If the `new-expression` terminates by throwing an exception, it may release storage by calling a deallocation function (3.7.3.2). 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 (3.7.3, 18.5.1.1, 18.5.1.2). A C++ program can provide alternative definitions of these functions (17.4.3.4) and/or class-specific versions (12.5). — end note ]

10 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.

11 A `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 most stringent fundamental alignment requirement (3.93.11) 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]

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.

[Example:

— new T results in a call of operator new(sizeof(T)),
— new(2,f) T results in a call of operator new(sizeof(T),2,f),
— new T[5] results in a call of operator new[](sizeof(T)*5+x), and
— 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-expression s, 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 is declared with an empty exception-specification (15.4), throw(), it indicates failure to allocate storage by throwing a bad_alloc exception (clause 15, 18.5.2.1); it returns a non-null pointer otherwise. If the allocation function is declared with an empty exception-specification, throw(), it returns null to indicate failure to allocate storage and a non-null pointer otherwise. — end note] 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:

— If the new-initializer is omitted:
  — If T is a (possibly cv-qualified) non POD trivial class type (or array thereof), the object is default-initialized (8.5).
  If T is a const-qualified type, the underlying class type shall have a user-declared provided default constructor.
  — Otherwise, the object created has indeterminate value. If T is a const-qualified type, or a (possibly cv-qualified) POD trivial class type (or array thereof) containing (directly or indirectly) a member of const-qualified type, the program is ill-formed;
  — If the new-initializer is of the form (), the item is value-initialized (8.5);
  — If the new-initializer is of the form (expression-list) and T is a class type, the appropriate constructor is called, using expression-list as the arguments (8.5);
5.3 Unary expressions

— If the new-initializer is of the form (expression-list) and T is an arithmetic, enumeration, pointer, or pointer-to-

member type and expression-list comprises exactly one expression, then the object is initialized to the (possibly

converted) value of the expression (8.5);

— Otherwise the new-expression is ill-formed.

17 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, access and ambiguity control are done for the destructor (12.4).

18 If any part of the object initialization described above terminates by throwing an exception and a suitable deallocation

function can be found, the deallocation function is called to free the memory in which the object was being constructed,

after which the exception continues to propagate in the context of the new-expression. If no unambiguous matching

deallocation function can be found, propagating the exception does not cause the object’s memory to be freed. [ Note:

This is appropriate when the called allocation function does not allocate memory; otherwise, it is likely to result in a

memory leak. — end note ]

19 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.

20 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. Any non-placement deallocation function matches a non-placement allocation function. 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 (3.7.3.2) 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. [ Example:

```c
struct S {
    // Placement allocation function:
    static void* operator new(std::size_t, std::size_t);

    // Usual (non-placement) deallocation function:
    static void operator delete(void*, std::size_t);
};

S* p = new (0) S; // ill-formed: non-placement deallocation function matches
// placement allocation function
```

— end example ]

21 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.

72) This may include evaluating a new-initializer and/or calling a constructor.
Whether the allocation function is called before evaluating the constructor arguments or after evaluating the constructor arguments but before entering the constructor is unspecified. It is also unspecified whether the arguments to a constructor are evaluated if the allocation function returns the null pointer or exits using an exception.

### 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. The operand shall have a pointer type, or a class type having a single conversion function (12.3.2) to a pointer type. The 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 this section. In either alternative, the value of the operand of `delete` may be a null pointer value. If it is not a null pointer value, in the first alternative (`delete object`), the value of the operand of `delete` shall be a pointer to a non-array object 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` shall be the pointer value which resulted from a previous array `new-expression`.\(^{73}\) 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]

In the first alternative (`delete object`), if the static type of the operand is different from its dynamic type, the static type shall be a base class of the operand’s dynamic type 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.\(^{74}\)

The `cast-expression` in a `delete-expression` shall be evaluated exactly once.

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.

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).

If the value of the operand of the `delete-expression` is not a null pointer value, the `delete-expression` will call a deallocation function (3.7.3.2). 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]

[Note: An implementation provides default definitions of the global deallocation functions `operator delete()` for non-arrays (18.5.1.1) and `operator delete[]()` for arrays (18.5.1.2). A C++ program can provide alternative defini-

---

\(^{73}\)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.

\(^{74}\)This implies that an object cannot be deleted using a pointer of type `void*` because there are no objects of type `void`.
5.4 Explicit type conversion (cast notation) Expressions

The result of the expression \((T)\) \textit{cast-expression} is of type \(T\). The result is an lvalue if \(T\) is an lvalue reference type, otherwise the result is an rvalue. \textit{Note: if} \(T\) \textit{is a non-class type that is cv-qualified, the cv-qualifiers are ignored when determining the type of the resulting rvalue; see 3.10. — end note}.

An explicit type conversion can be expressed using functional notation (5.2.3), a type conversion operator (\texttt{dynamic	extunderscore cast}, \texttt{static	extunderscore cast}, \texttt{reinterpret	extunderscore cast}, \texttt{const	extunderscore cast}), or the \textit{cast} notation.

\begin{grammar}
\textit{cast-expression:}
\texttt{unary	extunderscore expression}
\texttt{(type	extunderscore id) cast	extunderscore expression}
\end{grammar}

Types shall not be defined in casts.

Any type conversion not mentioned below and not explicitly defined by the user (12.3) is ill-formed.

The conversions performed by

\begin{itemize}
\item a \texttt{const	extunderscore cast} (5.2.11),
\item a \texttt{static	extunderscore cast} (5.2.9),
\item a \texttt{static	extunderscore cast} followed by a \texttt{const	extunderscore cast},
\item a \texttt{reinterpret	extunderscore cast} (5.2.10), or
\item a \texttt{reinterpret	extunderscore cast} followed by a \texttt{const	extunderscore cast},
\end{itemize}

can be performed using the \textit{cast} notation of explicit type conversion. The same semantic restrictions and behaviors apply, with the exception that in performing a \texttt{static	extunderscore cast} in the following situations the conversion is valid even if the base class is inaccessible:

\begin{itemize}
\item 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;
\item 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;
\end{itemize}

Draft
99 Expressions  5.5 Pointer-to-member operators

— a pointer to an object of an unambiguous non-virtual base class type, an lvalue or rvalue 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:

```cpp
struct A {}
struct I1 : A {}
struct I2 : A {}
struct D : I1, I2 {}
A *foo( D *p ) {
    return (A*)( p );  // ill-formed static_cast interpretation
}
```

— end example ]

6 The operand of a cast using the cast notation can be an rvalue 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” (where T is a completely-defined class type) 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” (where T is a completely-defined class type) 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 result is an object or a function of the type specified by the second operand.

4 If the dynamic type of the object does not contain the member to which the pointer refers, the behavior is undefined.

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,

```
struct S {
    S() : i(0) {}  
    mutable int i;
```

Draft
5.6 Multiplicative operators

The multiplicative operators *, /, and % group left-to-right.

```
    multiplicative-expression:
        pm-expression
        multiplicative-expression * pm-expression
        multiplicative-expression / pm-expression
        multiplicative-expression % pm-expression
```

The operands of * and / shall have arithmetic or enumeration type; the operands of % shall have integral or 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; otherwise \((a/b)*b + a\%b\) is equal to \(a\). If both operands are nonnegative then the remainder is nonnegative; if not, the sign of the remainder is implementation-defined\(^{75}\).

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.

---

\(^{75}\)According to work underway toward the revision of ISO C, the preferred algorithm for integer division follows the rules defined in the ISO Fortran standard, ISO/IEC 1539:1991, in which the quotient is always rounded toward zero.
additive-expression:
  multiplicative-expression
  additive-expression + multiplicative-expression
  additive-expression − multiplicative-expression

For addition, either both operands shall have arithmetic or enumeration type, or one operand shall be a pointer to a completely defined object type and the other shall have integral or enumeration type.

2 For subtraction, one of the following shall hold:
    — both operands have arithmetic or enumeration type; or
    — both operands are pointers to cv-qualified or cv-unqualified versions of the same completely defined object type;
    or
    — the left operand is a pointer to a completely defined object type and the right operand has integral or 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 For the purposes of these operators, a pointer to a nonarray object behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

5 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.

6 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 (18.1). 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.\(^{76}\)

\(^{76}\)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.

Draft
5.8 Shift operators

The shift operators \(<\) and \(>>\) group left-to-right.

\[
\text{shift-expression:} \\
\text{additive-expression} \\
\text{shift-expression} <\text{additive-expression} \\
\text{shift-expression} >>\text{additive-expression}
\]

The operands shall be of integral or 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.

The value of \(E_1 \ll E_2\) is \(E_1\) (interpreted as a bit pattern) left-shifted \(E_2\) bit positions; vacated bits are zero-filled. If \(E_1\) has an unsigned type, the value of the result is \(E_1\) multiplied by the quantity 2 raised to the power \(E_2\), reduced modulo ULLONG_MAX+1 if \(E_1\) has type \text{unsigned long long int}\, ULONG_MAX+1 if \(E_1\) has type \text{unsigned long int}\, UINT_-MAX+1 otherwise. [Note: the constants ULLONG_MAX, ULONG_MAX, and UINT_MAX are defined in the header <climits>.—end note]

The value of \(E_1 >> E_2\) is \(E_1\) right-shifted \(E_2\) bit positions. If \(E_1\) has an unsigned type or if \(E_1\) has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of \(E_1\) divided by the quantity 2 raised to the power \(E_2\). If \(E_1\) has a signed type and a negative value, the resulting value is implementation-defined.

5.9 Relational operators

The relational operators group left-to-right. [Example: \(a<b<c\) means \((a<b)<c\) and not \((a<b)\&\&(b<c)\).—end example]

\[
\text{relational-expression:} \\
\text{shift-expression} \\
\text{relational-expression} <\text{shift-expression} \\
\text{relational-expression} >\text{shift-expression} \\
\text{relational-expression} <=\text{shift-expression} \\
\text{relational-expression} >=\text{shift-expression}
\]

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 \texttt{false} or \texttt{true}. The type of the result is \texttt{bool}.

The usual arithmetic conversions are performed on operands of arithmetic or enumeration type. Pointer conversions (4.10) and qualification conversions (4.4) are performed on pointer operands (or on a pointer operand and a null pointer constant) to bring them to their \textit{composite pointer type}. If one operand is a null pointer constant, the composite pointer type is the type of the other operand. Otherwise, if one of the operands has type “pointer to \(cv1\ void\)” then the other has type “pointer to \(cv2 T\)” and the composite pointer type is “pointer to \(cv/2 void\)” where \(cv/2\) is the union of \(cv1\) and \(cv2\). Otherwise, the composite pointer type is a pointer type similar (4.4) to the type of one of the operands, with

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.

Draft
a cv-qualification signature (4.4) that is the union of the cv-qualification signatures of the operand types. [Note: this implies that any pointer can be compared to a null pointer constant and that any object pointer can be compared to a pointer to (possibly cv-qualified) void. — end note]

Example:

```c
void *p;
const int *q;
int **pi;
const int *const *pci;
void ct()
{
    p <= q;  // Both converted to const void*
    pi <= pci;  // Both converted to const int *const *
}
```

— end example] Pointers to objects or functions of the same type (after pointer conversions) can be compared, with a result defined as follows:

— If two pointers `p` and `q` of the same type point to the same object or function, or both point one past the end of the same array, or are both null, then `p<=q` and `p>q` both yield `true`, and `p<q` and `p>q` both yield `false`.
— If two pointers `p` and `q` of the same type point to different objects that are not members of the same object or elements of the same array or to different functions, or if only one of them is null, the results of `p<q`, `p>q`, `p<=q`, and `p>=q` are unspecified.
— If two pointers point to non-static data members of the same object, or to subobjects or array elements of such members, recursively, the pointer to the later declared member compares greater provided the two members are not separated by an access-specifier label (11.1) have the same access control (clause 11) and provided their class is not a union.
— If two pointers point to non-static data members of the same object separated by an access-specifier label (11.1) with different access control (clause 11) the result is unspecified.
— If two pointers point to data members of the same union object, they compare equal (after conversion to `void*`, if necessary). If two pointers point to elements of the same array or one beyond the end of the array, the pointer to the object with the higher subscript compares higher.
— Other pointer comparisons are unspecified.

5.10 Equality operators

equality-expression:

  relational-expression

  equality-expression == relational-expression

  equality-expression != relational-expression

The `==` (equal to) and the `!=` (not equal to) operators have the same semantic restrictions, conversions, and result type as the relational operators except for their lower precedence and truth-value result. [Note: `a<b == c<d` is `true` whenever `a<b` and `c<d` have the same truth-value. — end note] Pointers to objects or functions of the same type (after pointer conversions) can be compared for equality. Two pointers of the same type compare equal if and only if they are both null, both point to the same function, or both represent the same address (3.9.2).
5.11 Bitwise AND operator

Expressions  104

In addition, pointers to members can be compared, or a pointer to member and a null pointer constant. Pointer to member conversions (4.11) and qualification conversions (4.4) are performed to bring them to a common type. If one operand is a null pointer constant, the common type is the type of the other operand. Otherwise, the common type is a pointer to member type similar (4.4) to the type of one of the operands, with a cv-qualification signature (4.4) that is the union of the cv-qualification signatures of the operand types. [Note: this implies that any pointer to member can be compared to a null pointer constant. —end note] If both operands are null, they compare equal. Otherwise if only one is null, they compare unequal. Otherwise if either is a pointer to a virtual member function, the result is unspecified. Otherwise they compare equal if and only if they would refer to the same member of the same most derived object (1.8) or the same subobject if they were dereferenced with a hypothetical object of the associated class type. [Example:

```c
struct B {
    int f();
};
struct L : B {};
struct R : B {};
struct D : L, R {};

int (B::*pb)() = &B::f;
int (L::*pl)() = pb;
int (R::*pr)() = pb;
int (D::*pdl)() = pl;
int (D::*pdr)() = pr;
bool x = (pdl == pdr); //false
```

—end example]

5.11 Bitwise AND operator

[expr.bit.and]

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 enumeration operands.

5.12 Bitwise exclusive OR operator

[expr.xor]

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 enumeration operands.

5.13 Bitwise inclusive OR operator

[expr.or]

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 enumeration operands.

5.14 Logical AND operator

\[ \text{logical-and-expression:} \]
\[ \text{inclusive-or-expression} \]
\[ \text{logical-and-expression} \&\& \text{inclusive-or-expression} \]

1 The \&\& operator groups left-to-right. The operands are both implicitly converted to type bool (clause 4). The result is \texttt{true} if both operands are \texttt{true} and \texttt{false} otherwise. Unlike \&, \&\& guarantees left-to-right evaluation: the second operand is not evaluated if the first operand is \texttt{false}.

2 The result is a bool. If the second expression is evaluated, every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second expression.

5.15 Logical OR operator

\[ \text{logical-or-expression:} \]
\[ \text{logical-and-expression} \]
\[ \text{logical-or-expression} \text{|| logical-and-expression} \]

1 The || operator groups left-to-right. The operands are both implicitly converted to bool (clause 4). It returns \texttt{true} if either of its operands is \texttt{true}, and \texttt{false} otherwise. Unlike |, || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the first operand evaluates to \texttt{true}.

2 The result is a bool. All side effects of the first expression except for destruction of temporaries (12.2) happen before the second expression is evaluated.

5.16 Conditional operator

\[ \text{conditional-expression:} \]
\[ \text{logical-or-expression} \]
\[ \text{logical-or-expression} \? \text{expression : assignment-expression} \]

1 Conditional expressions group right-to-left. The first expression is implicitly converted to bool (clause 4). It is evaluated and if it is \texttt{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 (possibly cv-qualified) \texttt{void}, then the 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, and one of the following shall hold:

   — The second or the third operand (but not both) is a \texttt{throw-expression} (15.1); the result is of the type of the other and is an rvalue.

   — Both the second and the third operands have type \texttt{void}; the result is of type \texttt{void} and is an rvalue. [Note: this includes the case where both operands are \texttt{throw-expression} s. — end note ]

3 Otherwise, if the second and third operand have different types, and either has (possibly cv-qualified) class type, an attempt is made to convert each of those operands to the type of the other. The process for determining whether an operand expression \texttt{E1} of type \texttt{T1} can be converted to match an operand expression \texttt{E2} of type \texttt{T2} is defined as follows:
5.16 Conditional operator

Expressions 106

— If E2 is an lvalue: E1 can be converted to match E2 if E1 can be implicitly converted (clause 4) to the type “lvalue reference to T2”, subject to the constraint that in the conversion the reference must bind directly (8.5.3) to E1.

— If E2 is an rvalue, or if the conversion above cannot be done:

  — 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 an rvalue of type T2 by copy-initializing a temporary of type T2 from E1 and using that temporary as the converted operand.

  — Otherwise (i.e., if E1 or E2 has a nonclass type, or if they both have class types but the underlying classes are not either the same or one a base class of the other): E1 can be converted to match E2 if E1 can be implicitly converted to the type that expression E2 would have if E2 were converted to an rvalue (or the type it has, if E2 is an rvalue).

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 lvalues and have the same type, the result is of that type and is an lvalue 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 an rvalue. 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:

  — The second and third operands have the same type; the result is of that type. If the operands have class type, the result is an rvalue 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.

  — 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.

  — The second and third operands have pointer type, or one has pointer type and the other is a null pointer constant; pointer conversions (4.10) and qualification conversions (4.4) are performed to bring them to their composite pointer type (5.9). The result is of the composite pointer type.

  — The second and third operands have pointer to member type, or one has pointer to member type and the other is a null pointer constant; pointer to member conversions (4.11) and qualification conversions (4.4) are performed to bring them to a common type, whose cv-qualification shall match the cv-qualification of either the second or the third operand. The result is of the common type.
5.17 Assignment and compound assignment operators

The assignment operator (=) and the compound assignment operators all group right-to-left. All require a modifiable lvalue as their left operand 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 assignment-expression
throw-expression

assignment-operator: one of
= *= /= %= += -= >>= <<= &= ^= |=

In simple assignment (=), the value of the expression replaces that of the object referred to by the left operand.

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.

If the left operand is of class type, the class shall be complete. Assignment to objects of a class is defined by the copy assignment operator (12.8, 13.5.3).

[Note: For class objects, assignment is not in general the same as initialization (8.5, 12.1, 12.6, 12.8). — end note]

When the left operand of an assignment operator denotes a reference to T, the operation assigns to the object of type T denoted by the reference.

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.

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.

5.18 Comma operator

The comma operator groups left-to-right.

expression:
assignment-expression
expression, assignment-expression

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are not applied to the left expression. 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 an lvalue if its right operand is an lvalue, and is a bit-field if its right operand is an lvalue and a bit-field.
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)] the comma operator as described in clause 5 can appear only in parentheses. [Example:] 
\[
f(a, (t=3, t+2), c);
\]
has three arguments, the second of which has the value 5. — end example ]

5.19 Constant expressions

In several places, C++ requires expressions that evaluate to an integral or enumeration constant: as array bounds (8.3.4, 5.3.4), as case expressions (6.4.2), as bit-field lengths (9.6), as enumerator initializers (7.2), as static member-initializers (9.4.2), as integral or enumeration non-type template arguments (14.3), and as the first argument in a static_assert declaration (7).

constant_expression:
conditional-expression

An integral constant expression shall involve only literals of arithmetic types (2.13, 3.9.1), enumerators, non-volatile const variables and static data members of integral and enumeration types initialized with constant expressions (8.5), non-type template parameters of integral and enumeration types, sizeof expressions, and sizeof... expressions. Floating literals (2.13.3) shall appear only if they are cast to integral or enumeration types. Only type conversions to integral and enumeration types shall be used. In particular, except in sizeof expressions, functions, class objects, pointers, or references shall not be used, and assignment, increment, decrement, function call (including new expressions and delete expressions), comma operators, and throw expressions shall not be used.

Other expressions are considered constant expressions only for the purpose of non-local static object initialization (3.6.2). Such constant expressions shall evaluate to one of the following:

— a null pointer value (4.10),
— a null member pointer value (4.11),
— an arithmetic constant expression,
— an address constant expression,
— a reference constant expression,
— an address constant expression for a complete object type, plus or minus an integral constant expression, or
— a pointer to member constant expression.

An arithmetic constant expression shall satisfy the requirements for an integral constant expression, except that

— floating literals need not be cast to integral or enumeration type, and
— conversions to floating point types are permitted.

An address constant expression is a pointer to an lvalue designating an object of static storage duration, a string literal (2.13.4), or a function. The pointer shall be created explicitly, using the unary & operator, or implicitly using a non-type template parameter of pointer type, or using an expression of array (4.2) or function (4.3) type. The subscripting operator [] and the class member access . and -> operators, the & and * unary operators, and pointer casts (except dynamic_casts, 5.2.7) can be used in the creation of an address constant expression, but the value of an object shall not be accessed by the use of these operators. If the subscripting operator is used, one of its operands shall be an integral
constant expression. An expression that designates the address of a subobject of a non-POD class object (clause 9) is not an address constant expression (12.7). Function calls shall not be used in an address constant expression, even if the function is inline and has a reference return type.

A reference constant expression is an lvalue designating an object of static storage duration, a non-type template parameter of reference type, or a function. The subscripting operator [], the class member access — and -> operators, the & and * unary operators, and reference casts (except those invoking user-defined conversion functions (12.3.2) and except dynamic casts (5.2.7)) can be used in the creation of a reference constant expression, but the value of an object shall not be accessed by the use of these operators. If the subscripting operator is used, one of its operands shall be an integral constant expression. An lvalue expression that designates a member or base class of a non-POD class object (clause 9) is not a reference constant expression (12.7). Function calls shall not be used in a reference constant expression, even if the function is inline and has a reference return type.

A pointer to member constant expression shall be created using the unary & operator applied to a qualified-id operand (5.3.1), optionally preceded by a pointer to member cast (5.2.9).

Certain contexts require expressions that satisfy additional requirements as detailed in this sub-clause. Such expressions are called constant expressions. [Note: Those expressions can be evaluated during translation. — end note]

\[
\text{constant-expression:} \\
\text{conditional-expression}
\]

A conditional-expression is a constant expression unless it involves one of the following as a potentially evaluated subexpression (3.2), but subexpressions of logical AND (5.14), logical OR (5.15), and conditional (5.16) operations that are not evaluated are not considered [Note: an overloaded operator invokes a function — end note]:

- this (5.1) unless it appears as the postfix-expression in a class member access expression, including the result of the implicit transformation in the body of a non-static member function (9.3.1);
- an invocation of a function other than a constexpr function or a constexpr constructor [Note: overload resolution (13.3) is applied as usual — end note];
- an lvalue-to-rvalue conversion (4.1) unless it is applied to
  - an lvalue of integral type that refers to a non-volatile const variable or static data member initialized with constant expressions, or
  - an lvalue of literal type that refers to a non-volatile object defined with constexpr, or that refers to a sub-object of such an object;
- an id-expression that refers to a variable or data member of reference type;
- a type conversion from a floating-point type to an integral type (4.9) unless the conversion is directly applied to a floating-point literal;
- a dynamic cast (5.2.7);
- a type conversion from a pointer or pointer-to-member type to a literal type [Note: a user-defined conversion invokes a function — end note];
- a pseudo-destructor call (5.2.4);
- a class member access (5.2.5) unless its postfix-expression is of trivial or literal type or of pointer to trivial or literal type;
5.19 Constant expressions

Expressions

— increment or decrement operations (5.2.6, 5.3.2);
— a typeid expression (5.2.8) whose operand is of a polymorphic class type;
— a new-expression (5.3.4);
— a delete-expression (5.3.5);
— a subtraction (5.7) where both operands are pointers;
— a relational (5.9) or equality (5.10) operator where at least one of the operands is a pointer;
— an assignment or a compound assignment (5.17); or
— a throw-expression (15.1).

A constant expression is an integral constant expression if it is of integral or enumeration type. [Note: such expressions may be used as array bounds (8.3.4, 5.3.4), as case expressions (6.4.2), as bit-field lengths (9.6), as enumerator initializers (7.2), as static member initializers (9.4.2), and as integral or enumeration non-type template arguments (14.3). — end note]

If an expression of literal class type is used in a context where an integral constant expression is required, then that class type shall have a single conversion function to an integral or enumeration type and that conversion function shall be constexpr. [Example:

```c
struct A {
    constexpr A(int i) : val(i) { }
    constexpr operator int() { return val; }
    constexpr operator long() { 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
```
— end example ]

An expression is a potential constant expression if it is a constant expression when all occurrences of function parameters are replaced by arbitrary constant expressions of the appropriate type.
Chapter 6  Statements

1 Except as indicated, statements are executed in sequence.

   statement:
       labeled-statement
       expression-statement
       compound-statement
       selection-statement
       iteration-statement
       jump-statement
       declaration-statement
       try-block

6.1 Labeled statement

1 A statement can be labeled.

   labeled-statement:
       identifier : statement
       case constant-expression : statement
       default : statement

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

   expression-statement:
       expression_opt ;

The expression is evaluated and its value is discarded. The lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are not applied to the expression. 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
6.4 Selection statements

A compound statement defines a local scope (3.3). [Note: a declaration is a statement (6.7). — end note]

Selection statements choose one of several flows of control.

```c
if (condition) statement
if (condition) statement else statement
switch (condition) statement
```

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 local 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:

```c
if (x) {
    int i;
}
```

can be equivalently rewritten as

```c
if (x) {
    int i;
}
```

Thus after the `if` statement, `i` is no longer in scope. — end example]

The rules for `condition` s apply both to `selection-statement` s and to the `for` and `while` statements (6.5). The `declarator` shall not specify a function or an array. The `type-specifier-seq` shall not contain `typedef` and shall not declare a new class or enumeration. If the `auto` `type-specifier` appears in the `type-specifier-seq`, the `type-specifier-seq` shall contain no other `type-specifiers` except `cv-qualifiers`, and the type of the identifier being declared is deduced from the `assignment-expression` as described in 7.1.6.4.

A name introduced by a declaration in a `condition` (either introduced by the `type-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:

```c
if (int x = f()) {
    int x; // ill-formed, redeclaration of x
}
else {
```

Draft
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 implicitly converted to type bool. 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, implicitly converted to bool for statements other than switch; if that conversion is ill-formed, the program is ill-formed. The value of the condition will be referred to as simply “the condition” where the usage is unambiguous.

If a condition can be syntactically resolved as either an expression or the declaration of a local name, it is interpreted as a declaration.

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. 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.\(^\text{77}\)

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 of a class type for which a single conversion function to integral or enumeration type exists (12.3). If the condition is of class type, the condition is converted by calling that conversion function, and the result of the conversion is used in place of the original condition for the remainder of this section. Integral promotions are performed. 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 an integral constant-expression (5.19). The integral constant-expression is implicitly converted to the promoted type of the switch condition. No two of the case constants in the same switch shall have the same value after conversion to the promoted type of the switch condition.

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.

\(^{77}\)In other words, the else is associated with the nearest un-elsed if.
6.5 Iteration statements

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

1 Iteration statements specify looping.

```
iteration-statement:
  while (condition) statement
  do statement while (expression) ;
  for (for-init-statement condition; expression; expression) statement

for-init-statement:
  expression-statement
  simple-declaration

[Note: a for-init-statement ends with a semicolon. — end note]
```

2 The substatement in an iteration-statement implicitly defines a local 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]

4 [Note: The requirements on condition s in iteration statements are described in 6.4. — end note]

6.5.1 The while statement

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 (3.3.1) 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 object created in a condition is destroyed and created with each iteration of the loop. [Example:

struct A {
    int val;
    A(int i) : val(i) { }
    ~A() { }
    operator bool() { return val != 0; }
};

int i = 1;
while (A a = i) {
    // ...
    i = 0;
}

In the while-loop, the constructor and destructor are each called twice, once for the condition that succeeds and once for the condition that fails. —end example]

6.5.2 The do statement [stmt.do]

1 The expression is implicitly converted to bool; if that is not possible, 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 [stmt.for]

1 The for statement

    for (for-init-statement conditionopt ; expressionopt ) 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:

```c
int i = 42;
int a[10];
for (int i = 0; i < 10; i++)
  a[i] = i;
int j = i; // j = 42
```

— end example ]

### 6.6 Jump statements

Jump statements unconditionally transfer control.

- **jump-statement**: `break;`  
  `continue;`  
  `return expression_opt;`  
  `goto identifier;`

2 On exit from a scope (however accomplished), destructors (12.4) are called for all constructed objects with automatic storage duration (3.7.2) (named objects or temporaries) that are declared in that scope, in the reverse order of their declaration. Transfer out of a loop, out of a block, or back past an initialized variable with automatic storage duration involves the destruction of variables 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()` (18.4), for example) without destroying class objects with automatic storage duration. — end note ]

#### 6.6.1 The `break` statement

1 The `break` statement shall occur only in an `iteration-statement` or a `switch` statement and causes termination of the smallest enclosing `iteration-statement` or `switch` statement; control passes to the statement following the terminated statement, if any.

#### 6.6.2 The `continue` statement

1 The `continue` statement shall occur only in an `iteration-statement` and causes control to pass to the loop-continuation portion of the smallest enclosing `iteration-statement`, that is, to the end of the loop. More precisely, in each of the statements

Draft
while (foo) {
    do {
        for (; ;) {
            // ...
        }
    }
    continue;
    // ...
} while (foo);
}

for (;;) {
    // ...
} continue;

continue not contained in an enclosed iteration statement is equivalent to goto contin.

6.6.3 The return statement

1 A function returns to its caller by the return statement.

2 A return statement without an expression can be used only in functions that do not return a value, that is, a function with the return type void, a constructor (12.1), or a deSTRUCTOR (12.4). A return statement with an expression of non-void type can be used only in functions returning a value; the value of the expression is returned to the caller of the function. The expression is implicitly converted to the return type of the function in which it appears. A return statement can involve the construction and copy of a temporary object (12.2). [Note: A copy 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] 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.

3 A return statement with an expression of type “cv void” can be used only in functions with a return type of cv void; the expression is evaluated just before the function returns to its caller.

6.6.4 The goto statement

1 The goto statement unconditionally transfers control to the statement labeled by the identifier. The identifier shall be a label (6.1) located in the current function.

6.7 Declaration statement

1 A declaration statement introduces one or more new identifiers into a block; it has the form

\[
\text{declaration-statement:}
\]

\[
\text{block-declaration}
\]

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.

2 declaration-statement Variables with automatic storage duration (3.7.2) are initialized each time their is executed. Variables with automatic storage duration declared in the block are destroyed on exit from the block (6.6).

3 It is possible to transfer into a block, but not in a way that bypasses declarations with initialization. A program that jumps\(^78\) from a point where a local variable with automatic storage duration is not in scope to a point where it is in scope is ill-formed unless the variable has POD type (3.9) and is declared without an initializer (8.5). [Example:

\[
\text{void f()}
\]

\[
\{\text{ // ...}
\]

\[
\text{goto lx; // ill-formed: jump into scope of a}
\]

\(^78\)The transfer from the condition of a switch statement to a case label is considered a jump in this respect.
6.8 Ambiguity resolution

1 There is an ambiguity in the grammar involving expression-statement s and declaration s: 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: To disambiguate, the whole statement might have to be examined to determine if it is an expression-statement or a declaration. This disambiguates 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

2 The zero-initialization (8.5) of all local objects with static storage duration (3.7.1) is performed before any other initialization takes place. A local object of POD trivial or literal type (3.9) with static storage duration initialized with constant-expressions is initialized before its block is first entered. An implementation is permitted to perform early initialization of other local objects with static storage duration under the same conditions that an implementation is permitted to statically initialize an object with static storage duration in namespace scope (3.6.2). Otherwise such an object is initialized the first time control passes through its declaration; such an object 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 re-enters the declaration (recursively) while the object is being initialized, the behavior is undefined. [Example:

int foo(int i)
{
    static int s = foo(2*i);  // recursive call - undefined
    return i+1;
}

5 The destructor for a local object with static storage duration will be executed if and only if the variable was constructed. [Note: 3.6.3 describes the order in which local objects with static storage duration are destroyed. — end note]
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:

```cpp
class T {
   //...
   public:
   T();
   T(int);
   T(int, int);
};
T(a); // declaration
T(*b)(); // declaration
T(c)=7; // declaration
T(d), e, f=3; // declaration
extern int h;
T(g)(h,2); // declaration
```
— end example ] — end note]

The disambiguation is purely syntactic; that is, the meaning of the names occurring in such a statement, beyond whether they are type-names or not, is not generally used in or changed by the disambiguation. Class templates are instantiated as necessary to determine if a qualified name is a type-name. Disambiguation precedes parsing, and a statement disambiguated as a declaration may be an ill-formed declaration. If, during parsing, a name in a template parameter is bound differently than it would be bound during a trial parse, the program is ill-formed. No diagnostic is required. [Note: This can occur only when the name is declared earlier in the declaration. — end note ] [Example:

```cpp
struct T1 {
   T1 operator()(int x) { return T1(x); }
   int operator=(int x) { return x; }
   T1(int) {}
};
struct T2 { T2(int){ }; int a, (**b)(T2))(int), c, d;

void f() {
   // disambiguation requires this to be parsed
   // as a declaration
   T1(a) = 3,
   T2(4), // T2 will be declared as
   (**b)(T2(c))(int(d)); // 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 ]
Chapter 7  Declarations

1  Declarations specify how names are to be interpreted. Declarations have the form

\[
\text{declaration-seq:}
\text{declaration}
\text{declaration-seq declaration}
\]

\[
\text{declaration:}
\text{block-declaration}
\text{function-definition}
\text{template-declaration}
\text{explicit-instantiation}
\text{explicit-specialization}
\text{linkage-specification}
\text{namespace-definition}
\]

\[
\text{block-declaration:}
\text{simple-declaration}
\text{asm-definition}
\text{namespace-alias-definition}
\text{using-declaration}
\text{using-directive}
\text{static_assert-declaration}
\text{alias-declaration}
\]

\[
\text{alias-declaration:}
\text{using identifier = type-id}
\]

\[
\text{simple-declaration:}
\text{decl-specifier-seqopt init-declarator-listopt ;}
\]

\[
\text{static_assert-declaration:}
\text{static_assert ( constant-expression , string-literal ) ;}
\]

\[\text{Note: asm-definition s are described in 7.4, and linkage-specification s are described in 7.5. Function-definition s are described in 8.4 and template-declaration s are described in clause 14. Namespace-definition s are described in 7.3.1, using-declaration s are described in 7.3.3 and using-directive s are described in 7.3.4. — end note}\]

The \text{simple-declaration}

\[
\text{decl-specifier-seqopt init-declarator-listopt ;}
\]

is divided into two parts: \text{decl-specifier s, the components of a decl-specifier-seq, are described in 7.1 and declarator s, the components of an init-declarator-list, are described in clause 8.}
7.1 Specifiers

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.

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, and except for the declaration of an unnamed bit-field (9.6), 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 ]

In a static_assert-declaration the constant-expression shall be an integral constant expression (5.19). If the value of the expression when converted to bool 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, except that characters not in the basic source character set (2.2) 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 ]

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.

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.

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.

Only in function declarations for constructors, destructors, and type conversions can the decl-specifier-seq be omitted.79)

7.1 Specifiers

The specifiers that can be used in a declaration are

79) The “implicit int” rule of C is no longer supported.
The longest sequence of `decl-specifier` s that could possibly be a type name is taken as the `decl-specifier-seq` of a `declaration`. The sequence shall be self-consistent as described below. [Example:

```c
typedef char* Pc;
static Pc; // error: name missing
```

Here, the declaration `static Pc` is ill-formed because no name was specified for the static variable of type `Pc`. To get a variable called `Pc`, a `type-specifier` (other than `const` or `volatile`) has to be present to indicate that the `typedef-name` `Pc` is the name being (re)declared, rather than being part of the `decl-specifier` sequence. For another example,

```c
void f(const Pc); // void f(char* const) (not const char*)
void g(const int Pc); // void g(const int)
```

— end example ]

[Note: since `signed`, `unsigned`, `long`, and `short` by default imply `int`, a `type-name` appearing after one of those specifiers is treated as the name being (re)declared. [Example:

```c
void h(unsigned Pc); // void h(unsigned int)
void k(unsigned int Pc); // void k(unsigned int)
```

— end example ] — end note ]

### 7.1.1 Storage class specifiers

The storage class specifiers are

```c
storage-class-specifier:
    register
    static
typedef
    extern
to specifies that the named object has automatic storage duration (3.7.2). An object declared without a `storage-class-specifier` at block scope or declared as a function parameter has automatic storage duration by default.
```

Draft
A register specifier is a hint to the implementation that the object 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 object is taken. — end note]

The static specifier can be applied only to names of objects 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 an object declares the object to have static storage duration (3.7.1). 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.

The extern specifier can be applied only to the names of objects 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]

A name declared in a namespace scope without a storage-class-specifier has external linkage unless it has internal linkage because of a previous declaration and provided it is not declared const. Objects declared const and not explicitly declared extern have internal linkage.

The linkages implied by successive declarations for a given entity shall agree. That is, within a given scope, each declaration declaring the same object 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
```

Draft
Declared but undefined class can be used in an `extern` declaration. Such a declaration can only be used in ways that do not require a complete class type. [Example:

```c
struct S;
extern S a;
extern S f();
extern void g(S);

void h()
{
  g(a); // error: S is incomplete
  f();  // error: S is incomplete
}
```

The `mutable` specifier can be applied only to names of class data members (9.2) and cannot be applied to names declared `const` or `static`, and cannot be applied to reference members. [Example:

```c
class X {
    mutable const int* p; // OK
    mutable int* const q; // ill-formed
};
```

The `mutable` specifier on a class data member nullifies a `const` specifier applied to the containing class object and permits modification of the mutable class member even though the rest of the object is `const` (7.1.6.1).

### 7.1.2 Function specifiers

Function-specifiers can be used only in function declarations.

```
function-specifier:
    inline
    virtual
    explicit
```

A function declaration (8.3.5, 9.3, 11.4) with an `inline` specifier declares an *inline function*. The inline specifier indicates to the implementation that inline substitution of the function body at the point of call is to be preferred to the usual function call mechanism. An implementation is not required to perform this inline substitution at the point of call; however, even if this inline substitution is omitted, the other rules for inline functions 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 (80) 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.

80) The `inline` keyword has no effect on the linkage of a function.
An inline function shall be defined in every translation unit in which it is 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 string literal in the body of an extern inline function is the same object in different translation units. [Note: A string literal appearing in a default argument expression is not in the body of an inline function merely because the expression is used in a function call from that inline function. — end note]

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 within its class definition; see 12.3.1.

7.1.3 The typedef specifier

decl.typedef

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 used in a function-definition (8.4), and it shall not be combined in a decl-specifier-seq with any other kind of specifier except a type-specifier.

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
typedef int MILES, *KLICKSP;
the constructions
MILES distance;
extern KLICKSP metricp;

are all correct declarations; the type of distance is int and that of metricp is “pointer to int.” — end example]

A typedef-name can also be introduced by an alias-declaration. The identifier following the using keyword becomes a typedef-name. 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:
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:
typedef struct s { /* ... */ } s;
typedef int I;

Draft
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:

```
struct S {
  typedef struct A {} A;  // OK
  typedef struct B B;     // OK
  typedef A A;            // error

— end example ]
```

5 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:

```
class complex { /* ... */ };  // OK
typedef int complex;          // error: redefinition

— end example ]
```

6 Similarly, in a given scope, a class or enumeration shall not be declared with the same name as a typedef-name that is declared in that scope and refers to a type other than the class or enumeration itself. [ Example:

```
typedef int complex;
class complex { /* ... */ };  // error: redefinition

— end example ]
```

[ 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:

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

typedef struct S T;
S a = T();  // OK
struct T * p;  // error

— end example ]
```

8 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:

```
typedef struct {} *ps, S;  // S is the class name for linkage purposes
```

Draft
If a typedef TD names a type that is a reference to a type T, an attempt to create the type “lvalue reference to cv TD” creates the type “lvalue reference to T,” while an attempt to create the type “rvalue reference to cv TD” creates the type TD. [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
```
— end example ]

7.1.4 The friend specifier

The friend specifier is used to specify access to class members; see 11.4.

7.1.5 The constexpr specifier

The constexpr specifier shall be applied only to the definition of an object, function, or function template, or to the declaration of a static data member of a literal type (3.9). [Note: function parameters cannot be declared constexpr. — end note ] [Example:

```c
constexpr int square(int x) // OK
{ 
  return x * x;
}
constexpr int bufsz = 1024; // OK
constexpr struct pixel { // error: pixel is a type
  int x;
  int y;
};
int next(constexpr int x) // error
{ 
  return x + 1;
}
extern constexpr int memsz; // error: not a definition
```
— end example ]

A constexpr specifier used in a function declaration 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). A constexpr function shall not be virtual (10.3).

The definition of a constexpr function shall satisfy the following constraints:

— its return type shall be a literal type
— each of its parameter types shall be a literal type
— its function-body shall be a compound-statement of the form
  \[
  \{ \text{return expression;} \}
  \]
  
  where expression is a potential constant expression (5.19)
— every implicit conversion used in converting expression to the function return type (8.5) shall be one of those allowed in a constant expression (5.19).

[Example:

```cpp
constexpr int square(int x)
{ return x * x; }  // OK
constexpr long long_max()
{ return 2147483647; }  // OK
constexpr int abs(int x)
{ return x < 0 ? -x : x; }  // OK
constexpr void f(int x)  // error: return type is void
{ /* ... */ }
constexpr int prev(int x)
{ return --x; }  // error: use of decrement
constexpr int g(int x, int n)  // error: body not just "return expr"
{
    int r = 1;
    while (--n > 0) r *= x;
    return r;
}

— end example]

4 The definition of a constexpr constructor shall satisfy the following constraints:
— the compound-statement of its function-body shall be empty
— every non-static data member and base class sub-object shall be initialized (12.6.2)
— every constructor involved in initializing non-static data members and base class sub-objects shall be a constexpr constructor invoked with potential constant expression arguments, if any.

A trivial copy constructor is also a constexpr constructor.

[Example:

```cpp
struct Length {
    explicit constexpr Length(int i = 0) : val(i) { }
private:
    int val;
};

— end example]

Draft
7.1 Specifiers Declarations

5 If the instantiated template specialization of a constexpr function template would fail to satisfy the requirements for a constexpr function, the constexpr specifier is ignored and the specialization is not a constexpr function.

6 A constexpr specifier used in a non-static member function definition declares that member function to be const (9.3.1). [Note: the constexpr specifier has no other effect on the function type. —end note] The class of which that function is a member shall be a literal type (3.9). [Example:

```cpp
class debug_flag {
    public:
        explicit debug_flag(bool);
        constexpr bool is_on();  // error: debug_flag not // literal type
    private:
        bool flag;
    };
    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]

7 A constexpr specifier used in an object declaration declares the object as const. Such an object shall be initialized, and every expression that appears in its initializer (8.5) shall be a constant expression. Every implicit conversion used in converting the initializer expressions shall be one of those allowed in a constant expression (5.19). [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 [dcl.type]

1 The type-specifiers are

```cpp
type-specifier:
    simple-type-specifier
class-specifier
enum-specifier
elaborated-type-specifier
typename-specifier
cv-qualifier
```

2 As a general rule, at most one type-specifier is allowed in the complete decl-specifier-seq of a declaration. The only exceptions to this rule are the following:
— `const` or `volatile` can be combined with any other `type-specifier`. However, redundant `cv-qualifiers` are prohibited except when introduced through the use of `typedefs` (7.1.3) or template type arguments (14.3), in which case the redundant `cv-qualifiers` are ignored.

— `const` can be combined with any type specifier except itself.

— `volatile` can be combined with any type specifier except itself.

— `signed` or `unsigned` can be combined with `char`, `long`, `short`, or `int`.

— `short` or `long` can be combined with `int`.

— `long` can be combined with `double`.

— `long` can be combined with `long`.

— `auto` can be combined with any type specifier except itself.

At least one `type-specifier` that is not a `cv-qualifier` is required in a declaration unless it declares a constructor, destructor or conversion function.\(^{(81)}\)

[Note: class-specifiers and `enum-specifiers` are discussed in clause 9 and 7.2, respectively. The remaining `type-specifiers` are discussed in the rest of this section. — end note]

### 7.1.6.1 The `cv-qualifiers` \(^{[dcl.type.cv]}\)

There are two `cv-qualifiers`, `const` and `volatile`. If a `cv-qualifier` appears in a `decl-specifier-seq`, the `init-declarator-list` of the declaration shall not be empty. [Note: \(^{3.9.3}\) describes 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]

An object declared in namespace scope with a `const-qualified` type has internal linkage unless it is explicitly declared `extern` or unless it was previously declared to have external linkage. A variable of non-volatile `const-qualified` integral or `enumeration type initialized by an integral constant expression` can be used in integral constant `expressions` (5.19). [Note: as described in 8.5, the definition of an object or subobject of `const-qualified` type must specify an initializer or be subject to `default-initialization`. — 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:

\begin{verbatim}
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
\end{verbatim}

\(^{(81)}\)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-qualifier`s. The “implicit int” rule of C is no longer supported.
7.1 Specifiers Declarations

```c
*cip = 4; // ill-formed: attempt to modify through ptr to const
typ* 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
class X {
    public:
    mutable int i;
    int j;
};
class Y {
    public:
    X x;
    Y 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
```

If an attempt is made to refer to an object defined with a volatile-qualified type through the use of an lvalue with a non-volatile-qualified type, the program behaviour 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. 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.

7.1.6.2 Simple type specifiers

The simple type specifiers are...
The auto specifier is a placeholder for a type to be deduced (7.1.6.4). The other simple-type-specifier s specify either a previously-declared user-defined type or one of the fundamental types (3.9.1). Table 9 summarizes the valid combinations of simple-type-specifier s and the types they specify.

Table 9: simple-type-specifier s and the types they specify

<table>
<thead>
<tr>
<th>Specifier(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>type-name</td>
<td>the type named</td>
</tr>
<tr>
<td>char</td>
<td>“char”</td>
</tr>
<tr>
<td>unsigned char</td>
<td>“unsigned char”</td>
</tr>
<tr>
<td>signed char</td>
<td>“signed char”</td>
</tr>
<tr>
<td>char16_t</td>
<td>“char16_t”</td>
</tr>
<tr>
<td>char32_t</td>
<td>“char32_t”</td>
</tr>
<tr>
<td>bool</td>
<td>“bool”</td>
</tr>
<tr>
<td>unsigned</td>
<td>“unsigned int”</td>
</tr>
<tr>
<td>unsigned int</td>
<td>“unsigned int”</td>
</tr>
<tr>
<td>signed</td>
<td>“int”</td>
</tr>
<tr>
<td>signed int</td>
<td>“int”</td>
</tr>
<tr>
<td>int</td>
<td>“int”</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>“unsigned short int”</td>
</tr>
<tr>
<td>unsigned short</td>
<td>“unsigned short int”</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>“unsigned long int”</td>
</tr>
<tr>
<td>unsigned long</td>
<td>“unsigned long int”</td>
</tr>
<tr>
<td>unsigned long long int</td>
<td>“unsigned long long int”</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>“unsigned long long int”</td>
</tr>
</tbody>
</table>
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 and certain bit-fields (9.6) are represented as signed or unsigned quantities. The signed specifier forces char objects and bit-fields to be signed; it is redundant in other contexts. — end note]

The type denoted by decltype(e) is defined as follows:

- if e is an id-expression or a 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;
- otherwise, if e is a function call (5.2.2) or an invocation of an overloaded operator (parentheses around e are ignored), decltype(e) is the return type of that function;
- otherwise, if e is an lvalue, decltype(e) is T&, where T is the type of e;
- otherwise, decltype(e) is the type of e.

The operand of the decltype specifier is an unevaluated operand (clause 5).

[Example:

```c
const int&& foo();
int i;
struct A { double x; }
const A* a = new A();
decltype(foo()); // type is const int&&
decltype(i); // type is int
```

Draft


dcltype(a->x);  // type is double
dcltype(a->x)); // type is const double&

— end example ]

7.1.6.3 Elaborated type specifiers

elaborated-type-specifier:
   class-key ::opt nested-name-specifier opt identifier
class-key ::opt nested-name-specifieropt templateopt simple-template-id
enum enum-key ::opt nested-name-specifier opt identifier

1 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 identifier ;
   friend class-key ::opt identifier ;
   friend class-key ::opt simple-template-id ;
   friend class-key ::opt nested-name-specifier identifier ;
   friend class-key ::opt nested-name-specifier templateopt simple-template-id ;

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, 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. — end note ]

3 The class-key or enum-key enum-key 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-key enum-key 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. The enum-key used in an elaborated-type-specifier need not match the one in the enumeration’s definition. [ Example:

   enum class E { a, b };  
   enum E x = E::a;  // OK

— end example ]

7.1.6.4 auto specifier

auto type-specifier has two meanings depending on the context of its use. In a decl-specifier-seq that contains at least one type-specifier (in addition to auto) that is not a cv-qualifier, the auto type-specifier specifies that the object named in the declaration has automatic storage duration. The decl-specifier-seq shall contain no storage-class-specifiers. This use of the auto specifier shall only be applied to names of objects declared in a block (6.3) or to function parameters (8.4).
Otherwise (auto appearing with no type specifiers other than cv-qualifiers), the auto type-specifier signifies that the type of an object being declared shall be deduced from its initializer. The name of the object being declared shall not appear in the initializer expression.

This use of auto is allowed when declaring objects in a block (6.3), in namespace scope (3.3.5), and in a for-init-statement (6.5.3). The decl-specifier-seq shall be followed by one or more init-declarators, each of which shall have a non-empty initializer of either of the following forms:

\[
= \text{assignment-expression} \\
( \text{assignment-expression} )
\]

[Example:

auto x = 5; \hspace{1em} // OK: x has type int
const auto *v = &x, u = 6; \hspace{1em} // OK: v has type const int*, u has type const int
static auto y = 0.0; \hspace{1em} // OK: y has type double
static auto int z; \hspace{1em} // error: auto and static conflict
auto int r; \hspace{1em} // OK: r has type int

—end example]

The auto type-specifier can also be used in declaring an object in the condition of a selection statement (6.4) or an iteration statement (6.5), in the type-specifier-seq in a new-type-id (5.3.4), and in declaring a static data member with a constant-initializer that appears within the member-specification of a class definition (9.4.2).

A program that uses auto in a context not explicitly allowed in this section is ill-formed.

Once the type of a declarator-id has been determined according to 8.3, the type of the declared variable using the declarator-id is determined from the type of its initializer using the rules for template argument deduction. Let \( T \) be the type that has been determined for a variable identifier \( d \). Obtain \( P \) from \( T \) by replacing the occurrences of auto with a new invented type template parameter \( U \). Let \( A \) be the type of the initializer expression for \( d \). The type deduced for the variable \( d \) is then the deduced type determined using the rules of template argument deduction from a function call (14.8.2.1), where \( P \) is a function template parameter type and \( A \) is the corresponding argument type. If the deduction fails, the declaration is ill-formed.

7 If the list of declarators contains more than one declarator, the type of each declared variable is determined as described above. If the type deduced for the template parameter \( U \) is not the same in each deduction, the program is ill-formed.

[Example:

const auto &i = expr;

The type of \( i \) is the deduced type of the parameter \( u \) in the call \( f(\text{expr}) \) of the following invented function template:

\[
\text{template <class } U \text{> void } f(\text{const } U & u);
\]

—end example]

7.1.7 Alignment specifier

The alignment specifier has the form

\[
\text{alignment-specifier:} \\
\text{alignaaa ( constant-expression )} \\
\text{alignaaa ( type-id )}
\]

Draft
When the alignment specifier is of the form `alignas(constant-expression)`:
- the constant expression shall be an integral constant expression
- if the constant expression evaluates to a fundamental alignment, the alignment requirement of the declared object shall be the specified fundamental alignment
- 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 object shall be that alignment
- 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
- if the constant expression evaluates to zero, the alignment specifier shall have no effect
- otherwise, the program is ill-formed.

When the alignment specifier is of the form `alignas(type-id)`, it shall have the same effect as `alignas(alignment(type-id))` (5.3.6).

When multiple alignment specifiers are specified for an object, the alignment requirement shall be set to the strictest specified alignment.

The combined effect of all alignment specifiers in a declaration shall not specify an alignment that is less strict than the alignment that would otherwise be required for the object being declared.

An alignment specifier shall not be specified in a declaration of a `typedef`, a bit-field, a reference, a function parameter or return type, or an object declared with the `register` storage-class specifier. [Note: in short, the specifier can be used on automatic variables, namespace scope variables, and members of class types (as long as they are not bit-fields). In other words, it cannot be used in contexts where it would become part of a type so it would affect name mangling, name lookup, or ordering of function templates. —end note]

If the defining declaration of an object has an alignment specifier, any non-defining declaration of that object shall either specify equivalent alignment or have no alignment specifier. No diagnostic is required if declarations of an object have different alignment specifiers in different translation units.

[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
T alignas(T) alignas(A) buffer[N];
```

Specifying `alignas(T)` in the alignment specifier list ensures that the final requested alignment will not be weaker than `alignment(T)`, and therefore the program will not be ill-formed. —end example]

[Note: the alignment of a union type can be strengthened by applying the alignment specifier to any member of the union. —end note]

[Note: the `std::aligned_union` template (20.4.7) can be used to create a union containing a type with a non-trivial constructor or destructor. —end note]

### 7.2 Enumeration declarations

An enumeration is a distinct type (3.9.1) with named constants. Its name becomes an `enum-name`, within its scope.
7.2 Enumeration declarations

Each enumeration defines a type that is different from all other types. An 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 `type-specifier-seq` of an `enum-base` shall name an integral type; any cv-qualification is ignored. The identifiers in an `enumerator-list` are declared as constants, and can appear wherever constants are required. An `enumerator-definition` with `=` gives the associated `enumerator` the value indicated by the `constant-expression`. The `constant-expression` shall be an integral constant expression.

If the first `enumerator` has no `initializer`, the value of the corresponding constant is zero. An `enumerator-definition` without an `initializer` gives the `enumerator` the value obtained by increasing the value of the previous `enumerator` by one.

[ Example:

```c
enum { a, b, c=0 };
enum { d, e, f=e+2 };
```

defines `a`, `c`, and `d` to be zero, `b` and `e` to be 1, and `f` to be 3. — end example ]

The 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 ]

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 `enum-base`; if not explicitly specified, the underlying type of a scoped enumeration type is `int`. In 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` is the underlying type; if the initializing value of an `enumerator` cannot be represented by the underlying type, the program is ill-formed. If the underlying type is not fixed, the type of each `enumerator` is the type of its initializing value.
The underlying type of an enumeration 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 for an enumeration 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. The value of sizeof() applied to an enumeration type, an object of enumeration type, or an enumerator is the value of sizeof() applied to the underlying type.

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 \( e_{min} \) is the smallest enumerator and \( e_{max} \) is the largest, the values of the enumeration are the values in the range \( b_{min} \) to \( b_{max} \), defined as follows: Let \( K \) be 1 for a two’s complement representation and 0 for a one’s complement or sign-magnitude representation. \( b_{\max} \) is the smallest value greater than or equal to \( max(|e_{min}| - K, |e_{max}|) \) and equal to \( 2^M - 1 \), where \( M \) is a non-negative integer. \( b_{\min} \) is zero if \( e_{\min} \) is non-negative and \( -(b_{\max} + K) \) otherwise. The size of the smallest bit-field large enough to hold all the values of the enumeration type is \( max(M, 1) \) if \( b_{\min} \) is zero and \( M + 1 \) otherwise. It is possible to define an enumeration that has values not defined by any of its enumerators.

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:

```c
enum color { red, yellow, green=20, blue };  
color col = red;  
color* cp = &col;  
if (*cp == blue)  // ...
```

makes color a type describing various colors, and then declares col as an object of that type, and cp as a pointer to an object of that type. The possible values of an object of type color are red, yellow, green, blue these values can be converted to the integral values 0, 1, 20, and 21. Since enumerations are distinct types, objects of type color can be assigned only values of type color.

```c
color c = 1;  // error: type mismatch,  
              // no conversion from int to color
int i = yellow;  // OK: yellow converted to integral value 1  
                 // integral promotion
```

Note that this implicit enum to int conversion is not provided for a scoped enumeration:

```c
enum class Col { red, yellow, green };  
int x = Col::red;  // error: no Col to int conversion
Col y = Col::red;  
if (y) {}  // error: no Col to bool conversion
```
An expression of arithmetic or enumeration type can be converted to an enumeration type explicitly. The value is unchanged if it is in the range of enumeration values of the enumeration type; otherwise the resulting enumeration value is unspecified.

The each enum-name and each unscoped enumerator declared by an enum-specifier is declared in the scope that immediately contains the enum-specifier. Each scoped enumerator is declared in the scope of the enumeration. These names obey the scope rules defined for all names in (3.3) and (3.4).

```cpp
eenum direction { left='l', right='r' };  
void g()
{
    direction d;       // OK
    d = left;          // OK
    d = direction::right;  // OK
}

eenum class altitude { high='h', low='l' };  
void h()
{
    altitude a;        // OK
    a = high;          // error: high not in scope
    a = altitude::low; // OK
}
```

An enumerator declared in class scope can be referred to using the class member access operators (::, . (dot) and -> (arrow)), see 5.2.5.

```cpp
class X {
public:
    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
    // ...
}
```

Draft
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 3.3.5.

7.3.1 Namespace definition

The grammar for a namespace-definition is

```
namespace-name:
    original-namespace-name
    namespace-alias
original-namespace-name:
    identifier
namespace-definition:
    named-name-space-definition
    unnamed-name-space-definition
named-name-space-definition:
    original-namespace-definition
    extension-name-space-definition
extension-name-space-definition:
    namespace original-namespace-name { namespace-body }
unnamed-name-space-definition:
    namespace { namespace-body }
namespace-body:
    declaration-seq
```

The identifier in an original-name-space-definition shall not have been previously defined in the declarative region in which the original-name-space-definition appears. The identifier in an original-name-space-definition is the name of the namespace. Subsequently in that declarative region, it is treated as an original-name-space-name.

The original-name-space-name in an extension-name-space-definition shall have previously been defined in an original-name-space-definition in the same declarative region.

Every namespace-definition shall appear in the global scope or in a namespace scope (3.3.5).

Because a namespace-definition contains declarations in its namespace-body and a namespace-definition is itself a declaration, it follows that namespace-definitions can be nested. [Example:

```
namespace Outer {
    int i;
    namespace Inner {
        void f() { i++; } //Outer::i
        int i;
        void g() { i++; } //Inner::i
    }
}
```
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:

```cpp
namespace Q {
    namespace V {
        void f(); // enclosing namespaces are the global namespace, Q, and Q::V
        class C { void m(); }; } 
    void V::f() { // enclosing namespaces are the global namespace, Q, and Q::V
        extern void h(); // ... so this declares Q::V::h
    } 
    void V::C::m() { // enclosing namespaces are the global namespace, Q, and Q::V
    }
}
```
— end example ]

7.3.1.1 Unnamed namespaces [namespace.unnamed]

An unnamed-namespace-definition behaves as if it were replaced by

```cpp
namespace unique { /* empty body */ } 
using namespace unique ;
namespace unique { namespace-body }
```

where all occurrences of `unique` in a translation unit are replaced by the same identifier and this identifier differs from all other identifiers in the entire program.82) [Example:

```cpp
namespace { int i; } // unique::i
void f() { i++; } // unique::i++

namespace A {
    namespace {
        int i; // A::unique::i
        int j; // A::unique::j
    }
    void g() { i++; } // A::unique::i++
}

using namespace A;
void h() {
    i++; // error: unique::i or A::unique::i
    A::i++; // A::unique::i
    j++; // A::unique::j
}
```
— end example ]

82) Although entities in an unnamed namespace might have external linkage, they are effectively qualified by a name unique to their translation unit and therefore can never be seen from any other translation unit.

Draft
2 The use of the static keyword is deprecated when declaring objects in a namespace scope (see annex D); the unnamed-
namespace provides a superior alternative.

7.3.1.2 Namespace member definitions

Members (including explicit specializations of templates (14.7.3)) of a namespace can be defined within that namespace. [Example:

```cpp
namespace X {
    void f() { /* ... */ }
}
```

— end example ]

Members (including explicit specializations of templates (14.7.3)) 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. [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();
    }
}
```

```cpp
namespace R {
    void Q::V::g() { /* ... */ }  // error: R doesn't enclose Q
}
```

— end example ]

Every name first declared in a namespace is a member of that namespace. If a friend declaration in a non-local class first declares a class or function the friend class or function is a member of the innermost enclosing namespace. The name of the friend is not found by unqualified lookup (3.4.1) or by qualified lookup (3.4.3) until a matching declaration is provided in that namespace scope (either before or after the class definition granting friendship). If a friend function 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 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 defined.
void h(int);
```

83) this implies that the name of the class or function is unqualified.
template <class T> void f2(T);
namespace A {
  class X {
    friend void f(X);    // is a friend
    class Y {
      friend void g();    // is a friend
      friend void h(int); // is a friend
      // not considered
      friend void f2<>(int); // is a friend
    };
  };
  // , , are not visible here
  X x;
  void g() { f(x); }    // definition of
  void f(X) { /* ... */ }    // definition of
  void h(int) { /* ... */ }    // definition of
  // , , are visible here and known to be friends
}

using A::x;

void h()
{
  A::f(x);
  A::X::f(x);    // error: is not a member of A::X
  A::X::Y::g();    // error: is not a member of A::X::Y
}

— end example ]

7.3.2 Namespace alias

1 A namespace-alias-definition declares an alternate name for a namespace according to the following grammar:

    namespace-alias:
      identifier
    namespace-alias-definition:
      namespace identifier = qualified-name-specifier ;
    qualified-name-specifier:
      ::opt  nested-name-specifieropt  namespace-name

2 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 ]

3 In a declarative region, a namespace-alias-definition can be used to redefine a namespace-alias declared in that declarative region to refer only to the namespace to which it already refers. [Example: the following declarations are well-formed:

    namespace Company_with_very_long_name { /* ... */ }
    namespace CWVLN = Company_with_very_long_name;

Draft
A namespace-name or namespace-alias shall not be declared as the name of any other entity in the same declarative region. A namespace-name defined at global scope shall not be declared as the name of any other entity in any global scope of the program. No diagnostic is required for a violation of this rule by declarations in different translation units.

7.3.3 The using declaration

A using-declaration introduces a name into the declarative region in which the using-declaration appears. That name is a synonym for the name of some entity declared elsewhere.

using-declaration:
    using typenameopt ::opt nested-name-specifier unqualified-id ;
    using :: 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]

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. Such a using-declaration 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 ]
4 [Note: since constructors and destructors do not have names, a using-declaration cannot refer to a constructor or 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-assignment operator for the derived class (12.8), the using-declaration does not by itself suppress the implicit declaration of the derived class copy-assignment operator; the copy-assignment operator from the base class is hidden or overridden by the implicitly-declared copy-assignment operator of the derived class, as described below.

5 A using-declaration shall not name a template-id. [Example:

```cpp
class A {
public:
    template <class T> void f(T);
    template <class T> struct X { }; 
};
class B : public A {
public:
    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). In a using-declaration, a prefix :: refers to the global namespace. [Example:

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

```
void g();
}

namespace X {
    using ::f;          // global f
    using A::g;         // A's g
}

namespace X {
    using A::g;
// A's g
}

void h()
{
    X::f();           // calls ::f
    X::g();           // calls A::g
}

— end example ]

A using-declaration is a declaration and can therefore be used repeatedly where (and only where) multiple declarations are allowed. [ Example:

namespace A {
    int i;
}

namespace A1 {
    using A::i;   // OK: double declaration
    using A::i;
}

void f()
{
    using A::i;   // error: double declaration
    using A::i;
}

class B {
public:
    int i;
};

class X : public B {
    using B::i;   // error: double member declaration
    using B::i;
};

— end example ]

The entity declared by a using-declaration shall be known in the context using it according to its definition at the point of the using-declaration. Definitions added to the namespace after the using-declaration are not considered when a use of the name is made. [ Example:

namespace A {

Draft
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 ]

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-declaration s. [ Example:

namespace A {
    int x;
}

namespace B {
    int i;
    struct g { }; // OK: hides struct g
    struct x { };
    void f(int);
    void f(double);
    void g(char); // error: i declared twice
}

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)
}

Draft
using B::g;
g('a');  // calls B::g(char)
struct g g1;  // g1 has class type B::g
using B::x;
using A::x;  // OK: hides struct B::x
x = 99;  // assigns to A::x
struct x x1;  // x1 has class type B::x
}

— end example ]

14 If a function declaration in namespace scope or block scope has the same name and the same parameter types as a function introduced by a using-declaration, and the declarations do not declare the same function, the program is ill-formed. [Note: two using-declaration s may introduce functions with the same name and the same parameter types. If, for a call to an unqualified function name, function overload resolution selects the functions introduced by such using-declaration s, 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)
    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 ]

15 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), and cv-qualification in a base class (rather than conflicting). [Example:

```c
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 ]

[ Note: two using-declaration s may introduce functions with the same name and the same parameter types. If, for a call to an unqualified function name, function overload resolution selects the functions introduced by such using-declaration s, the function call is ill-formed. — end note ]

17  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.

18  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
    }
The alias created by the using-declaration has the usual accessibility for a member-declaration. [Example:

```cpp
class A {
private:
    void f(char);
public:
    void f(int);
protected:
    void g();
};

class B : public A {
    using A::f; // error: A::f(char) is inaccessible
    public:
    using A::g; // B::g is a public synonym for A::g
};
```

— end example ]

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

**using-directive:**

```
using namespace ::opt 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 ]

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:

```cpp
namespace A {
    int i;
}
namespace B {
    namespace C {
        int i;
    }
    using namespace A::B::C;
    void f1() {
        i = 5; // OK, C::i visible in B and hides A::i
    }
```
namespace D {
    using namespace B;
    using namespace C;
    void f2() {
        i = 5; // ambiguous, B::C::i or A::i?
    }
}
void f3() {
    i = 5; // uses A::i
}
void f4() {
    i = 5; // ill-formed; neither i is visible
}

— end example ]

The using-directive is transitive: if a scope contains a using-directive that nominates a second namespace that itself contains using-directive s, the effect is as if the using-directive s from the second namespace also appeared in the first.

[ Example:

namespace M {
    int i;
}
namespace N {
    int i;
    using namespace M;
}
void f() {
    using namespace N;
    i = 7; // error: both M::i and N::i are visible
}

For another example,

namespace A {
    int i;
}
namespace B {
    int i;
    int j;
} namespace C {
    namespace D {
        using namespace A;
        int j;
        int k;
    }
}

Draft
int a = i; // B::i hides A::i
}
using namespace D;
int k = 89;   // no problem yet
int l = k;   // ambiguous: C::k or D::k
int m = i;   // B::i hides A::i
int n = j;   // D::j hides B::j
} }

— end example ]

If a namespace is extended by an extension-name-space-definition 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 extension-name-space-definition can be used after the extension-name-space-definition.

If name lookup finds a declaration for a name in two different namespaces, and the declarations do not declare the same entity and do not declare functions, the use of the name is ill-formed. [ Note: in particular, the name of an object, 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();
}
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();      // error: name h found in two namespaces
}
 — end note ]

During overload resolution, all functions from the transitive search are considered for argument matching. The set of declarations found by the transitive search is unordered. [ Note: in particular, the order in which namespaces were considered and the relationships among the namespaces implied by the using-directives do not cause preference to be given to any of the declarations found by the search. — end note ] An ambiguity exists if the best match finds two functions with the same signature, even if one is in a namespace reachable through using-directives in the namespace of the other.\(^{84}\) [ Example:

\(^{84}\) 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.
namespace D {
    int d1;
    void f(char);
}
using namespace D;

int d1;                   // OK: no conflict with D::d1

namespace E {
    int e;
    void f(int);
}

namespace D {             // namespace extension
    int d2;
    using namespace E;
    void f(int);
}

void f() {
    d1++;                // error: ambiguous ::d1 or D::d1?
    ::d1++;              // OK
    D::d1++;              // OK
    d2++;                // OK: D::d2
    e++;                 // OK: E::e
    f(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

```
asm-definition:
    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:
linkage-specification:
   extern string-literal { declaration-seqopt }

The string-literal indicates the required language linkage. This International Standard specifies the semantics for the string-literal's "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:

    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:

    extern "C" void f1(void(*)(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();   // the name f2 has C++ language linkage and the
                                        // function's type has C language linkage
    FUNC f2;                         // the name of function f3 and the function's type
                                        // have C language linkage
    extern "C" FUNC f3;              // the name of the variable pf2 has C++ linkage and
                                        // the type of pf2 is pointer to C++ function that
                                        // takes one parameter of type pointer to C function
    void (*pf2)(FUNC*);              // the name of the function f4 has
                                        // internal linkage (not C language
                                        // linkage) and the function's type
                                        // has C language linkage.
    }

    extern "C" void f5() {          // OK: Name linkage (internal)
                                        // and function type linkage (C
                                        // language linkage) gotten from
                                        // previous declaration.
        extern void f4();
    }

    — end example]
extern void f4(); // OK: Name linkage (internal)
// and function type linkage (C
// language linkage) gotten from
// previous declaration.
}

void f6() {
    extern void f4(); // OK: Name linkage (internal)
    // and function type linkage (C
    // language linkage) gotten from
    // previous declaration.
}

— end example ] A C language linkage is ignored for the names of class members and the member function type of class member functions. [ Example:

extern "C" typedef void FUNC_c();
class C {
    void mf1(FUNC_c*); // the name of the function mf1 and the member
    // function's type have C++ language linkage: the
    // parameter has type pointer to C function
    FUNC_c mf2; // the name of the function mf2 and the member
    // function's type have C++ language linkage
    static FUNC_c* q; // the name of the data member q has C++ language
    // linkage and the data member's type is pointer to
    // C function
};

extern "C" {
    class X {
    void mf(); // the name of the function mf and the member
    // function's type have C++ language linkage
    void mf2(void(*)()); // the name of the function mf2 has C++ language
    // linkage; the parameter has type pointer to
    // C function
    
    }
}

— end example ]

5 If two declarations of the same function or object specify different linkage-specification s (that is, the linkage-specification s of these declarations specify different string-literal s), the program is ill-formed if the declarations appear in the same translation unit, and the one definition rule (3.2) applies if the declarations appear in different translation units. Except for functions with C++ linkage, a function declaration without a linkage specification shall not precede the first linkage specification for that function. A function can be declared without a linkage specification after an explicit linkage specification has been seen; the linkage explicitly specified in the earlier declaration is not affected by such a function declaration.

6 At most one function with a particular name can have C language linkage. Two declarations for a function with C
language linkage with the same function name (ignoring the namespace names that qualify it) that appear in different namespace scopes refer to the same function. Two declarations for an object 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 object. [Note: because of the one definition rule (3.2), only one definition for a function or object with C linkage may appear in the program; that is, such a function or object must not be defined in more than one namespace scope. For example,

```cpp
namespace A {
    extern "C" int f();
    extern "C" int g() { return 1; }
    extern "C" int h();
}

namespace B {
    extern "C" int f();  // k::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
    // k::h and ::h refer to the same function
```

— end note ]

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:

```cpp
extern "C" double f();
static double f();  // error
extern "C" int i;  // declaration
extern "C" {
    int i;  // definition
}
extern "C" static void g();  // error
```

— end example ]

8 [Note: because the language linkage is part of a function type, when a pointer to C function (for example) is dereferenced, the function to which it 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.
Chapter 8  Declarators

A declarator declares a single object, 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 two components of a declaration are 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 objects, functions or typedefs being declared. The declarators specify the names of these objects, functions or typedefs, 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.  

Declarators have the syntax

declarator:
  direct-declarator
  ptr-operator declarator

direct-declarator:
  declarator-id
  direct-declarator ( parameter-declaration-clause ) cv-qualifier-seq_opt exception-specification_opt
  direct-declarator [ constant-expression_opt ]
  ( declarator )

85) A declaration with several declarators are 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. The 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
8.1 Type names

To specify type conversions explicitly, and as an argument of `sizeof`, `alignof`, `new`, or `typeid`, the name of a type shall be specified. This can be done with a `type-id`, which is syntactically a declaration for an object or function of that type that omits the name of the object or function.

```
type-id:
  type-specifier-seq abstract-declarator

abstract-declarator:
  ptr-operator abstract-declarator
  direct-abstract-declarator

  ( parameter-declaration-clause ) cv-qualifier-seq cv-qualifier-seq_opt
  exception-specification
  [ constant-expression_opt ]

  [ abstract-declarator ]
```

It is possible to identify uniquely the location in the `abstract-declarator` where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier.

```
Example:

int // int i
int * // int *pi
int *[3] // int *[p][3]
int (*)(double) // int (*)(double)
```

name respectively the types “int,” “pointer to int,” “array of 3 pointers to int,” “pointer to array of 3 int,” “function of (no parameters) returning pointer to int,” and “pointer to a function of (double) returning int.” — end example

Draft
A type can also be named (often more easily) by using a `typedef` (7.1.3).

### 8.2 Ambiguity resolution

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 ]

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.

[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 expression
  new (int(*[x]));   // new type-id
}
```

For another example,

```c
template <class T>
struct S {
  T *p;
};
S<int()> x; //type-id
S<int(1)> y; // expression (ill-formed)
```

For another example,
8.3 Meaning of declarators

Declarators

`void foo()
{
    sizeof(int(1));  // expression
    sizeof(int());  // type-id (ill-formed)
}

For another example,

`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:

`class C { }; void f(int(C)) { }  // void f(int(*fp)(C c)) { }
    // not: void f(int C);
    int g(C);
    void foo() {
        f(1);  // error: cannot convert 1 to function pointer
        f(g);  // OK
    }

For another example,

`class C { }; void h(int *(C[10]));  // void h(int *(fp)(C _parm[10]));
    // not: void h(int *C[10]);

— 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 (12.3, 12.4, 13.5) and for the declaration of template specializations or partial specializations (14.7). A declarator-id shall not be qualified except for the definition of a member function (9.3) or static data member (9.4) outside of its class, the definition or explicit instantiation of a function or variable member of a namespace outside of its namespace, or the definition of a previously declared explicit specialization outside of its namespace, or the declaration of a friend function that is a member of another class or namespace (11.4). When the declarator-id is qualified, the declaration shall refer to a previously declared member

Draft
of the class or namespace to which the qualifier refers, and the member shall not 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. [ Note: if the qualifier is the global :: scope resolution operator, the declarator-id refers to a name declared in the global namespace scope. — end note ]

2 An auto, static, 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.

3 Thus, a declaration of a particular identifier has the form

\[ T \ D \]

where \( T \) is a 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

\[ \text{int unsigned i;} \]

the type specifiers int unsigned determine the type “unsigned int” (7.1.6.2). — end example ]

5 In a declaration \( T \ D \) where \( D \) is an unadorned identifier the type of this identifier is “\( T \).”

6 In a declaration \( T \ D \) where \( D \) has the form

\[ ( D_1 ) \]

the type of the contained declarator-id is the same as that of the contained declarator-id in the declaration

\[ T \ D_1 \]

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

\[ * \ cv-qualifier-seq_{opt} \ D_1 \]

and the type of the identifier in the declaration \( T \ D_1 \) 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-qualifier s apply to the pointer and not to the object pointed to.

2 [ Example: the declarations

\[ \text{const int ci = 10, *pc = &ci, *const cpc = pc, **ppc; int i, *p, *const cp = &i;} \]

Draft
8.3 Meaning of declarators

Declarators 164

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

```c
i = ci;
*cp = ci;
pc++;
pc = cpc;
pc = p;
ppc = &pc;
```

Examples of ill-formed operations are

```c
ci = 1;  // error
ci++;   // error
*p = 2;  // error
*p = pc; // error
ppc = &pc; // error
```

Each is unacceptable because it would either change the value of an object declared const or allow it to be changed through a cv-unqualified pointer later, for example:

```c
*ppc = &ci;               // OK, but would make p point to ci ...
                           // ... because of previous error
*p = 5;                   // clobber ci
```

— end example ]

3 See also 5.17 and 8.5.

4 [ Note: there are no pointers to references; see 8.3.2. 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

```c
& D1
&& 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.” Cv-qualified references are ill-formed except when the cv-qualifiers are introduced through the use of a typedef (7.1.3) or of a template type argument (14.3), in which case the cv-qualifiers are ignored. [ Example: ]

```c
typedef int& A;
const A aref = 3;  // ill-formed; non-const reference initialized with rvalue
```

Draft
The type of `aref` is “reference to int”, not “const reference to int”. — end example] [Note: a reference can be thought of as a name of an object. — end note] A declarator that specifies the type “reference to cv void” is ill-formed.

A reference type that is declared using `&` is called an lvalue reference, and a reference type that is declared using `&&` is called an rvalue reference. Lvalue references and rvalue references are distinct types. Except where explicitly noted, they are semantically equivalent and commonly referred to as references.

Example:

```c
void f(double& a) { a += 3.14; }
// ...
double d = 0;
f(d);
```

declares `a` to be a reference parameter of `f` so the call `f(d)` will add 3.14 to `d`.

```c
int v[20];
// ...
int& g(int i) { return v[i]; }
// ...
g(3) = 7;
```

declares the function `g()` to return a reference to an integer so `g(3)=7` will assign 7 to the fourth element of the array `v`. For another example,

```c
struct link {
    link* next;
};

link* first;

void h(link*& p) // p is a reference to pointer
{
    p->next = first;
    first = p;
    p = 0;
}

void k()
{
    link* q = new link;
    h(q);
}
```

declares `p` to be a reference to a pointer to `link` so `h(q)` will leave `q` with the value zero. See also 8.5.3. — end example]

It is unspecified whether or not a reference requires storage (3.7).

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);
8.3 Meaning of declarators

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 dereferencing a null pointer, which causes undefined behavior. As described in 9.6, a reference cannot be bound directly to a bit-field. —end note]

8.3.3 Pointers to members

In a declaration \( T D \) where \( D \) has the form

\[
::_{opt} \text{nested-name-specifier} \ast \text{cv-qualifier-seq}_{opt} \ D1
\]

and the nested-name-specifier names a class, and the type of the identifier in the declaration \( T D1 \) is “derived-declarator-type-list \( T \),” then the type of the identifier of \( D \) is “derived-declarator-type-list cv-qualifier-seq pointer to member of class nested-name-specifier of type \( T \).”

[Example:

class X {
    public:
        void f(int);
        int a;
    class Y;

    int X::* pmi = &X::a;
    void (X::* pmf)(int) = &X::f;
    double X::* pmd;
    char Y::* pmc;
}

declares pmi, pmf, pmd and pmc to be a pointer to a member of X of type int, a pointer to a member of X of type void(int), a pointer to a member of X of type double and a pointer to a member of Y of type char respectively. The declaration of pmd is well-formed even though X has no members of type double. Similarly, the declaration of pmc is well-formed even though Y is an incomplete type. pmi and pmf can be used like this:

\[
X \text{ obj;}
// ... 
\text{obj.*pmi = 7; // assign 7 to an integer}
\text{obj.*pmf}(7); // call a function member of obj}
\]

—end example]

A pointer to member shall not point to a static member of a class (9.4), a member with reference type, or “cv void.” [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

\[
D1 [\text{constant-expression}_{opt}]
\]

Draft
and the type of the identifier in the declaration `T D1` is “derived-declarator-type-list T,” then the type of the identifier of `D` is an array type; if the type of the identifier of `D` contains the `auto` type deduction type-specifier, the program is ill-formed. `T` is called the array element type; this type shall not be a reference type, the (possibly cv-qualified) type `void`, a function type or an abstract class type. If the constant-expression (5.19) is present, it shall be an integral constant expression 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 `NT`.” An object of array type contains a contiguously allocated non-empty set of `N` subobjects of type `T`. 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 `NT`” 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 `NT`” is adjusted to “array of `N` cv-qualifier-seq `T`,” and similarly for “array of unknown bound of `T`.”

Example:

```c
typedef int A[5], AA[2][3];
typedef const A CA;
// type is “array of 5 const int"
typedef const AA CAA;
// type is “array of 2 array of 3 const int"
```

—end example] [Note: an “array of `N` cv-qualifier-seq `T`” has cv-qualified type; see 3.9.3. —end note]

2 An array can be constructed from one of the fundamental types (except `void`), from a pointer, from a pointer to member, from a class, from an enumeration type, or from another array.

3 When several “array of” specifications are adjacent, a multidimensional array is created; the constant expressions that specify the bounds of the arrays can be omitted only for the first member of the sequence. [Note: this elision is useful for function parameters of array types, and when the array is external and the definition, which allocates storage, is given elsewhere. —end note] The first constant-expression can 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.”

Example:

```c
float fa[17], *afp[17];
```

declares an array of `float` numbers and an array of pointers to `float` numbers. For another example,

```c
static int x3d[3][5][7];
```

declares 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. —end example]

[ Note: conversions affecting lvalues of array type are described in 4.2. Objects of array types cannot be modified, see 3.10. —end note]

6 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 \times j \times \ldots \times k`, then `E` appearing in an expression is converted to a pointer to an `(n-1)`-dimensional array with rank `j \times \ldots \times 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.

[Example: consider

```c
int x[3][5];
```

Here `x` is a \(3 \times 5\) 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[i]` 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]

[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

```c
D1 ( parameter-declaration-clause ) cv-qualifier-seq\textsubscript{opt} exception-specification\textsubscript{opt} returning \textsubscript{T}
```

and the type of the contained declarator-id in the declaration `T D1` is “derived-declarator-type-list \textsubscript{T},” the type of the declarator-id in `D` is “derived-declarator-type-list function of (parameter-declaration-clause ) cv-qualifier-seq\textsubscript{opt} returning \textsubscript{T}”; a type of this form is a function type \textsuperscript{86}.

- `parameter-declaration-clause`: 
  - `parameter-declaration-list\textsubscript{opt} ...\textsubscript{opt}
  - `parameter-declaration-list , ...

- `parameter-declaration-list`:
  - `parameter-declaration`
  - `parameter-declaration-list , parameter-declaration`

- `parameter-declaration`:
  - `decl-specifier-seq declarator`
  - `decl-specifier-seq declarator = assignment-expression`
  - `decl-specifier-seq abstract-declarator\textsubscript{opt} returning \textsubscript{T} = assignment-expression`

\textsuperscript{86} As indicated by the syntax, cv-qualifiers are a significant component in function return types.

The `parameter-declaration-clause` determines the arguments that can be specified, and their processing, when the func-
tion is called. [Note: the `parameter-declaration-clause` is used to convert the arguments specified on the function call;
see \textsuperscript{5.2.2.} — end note] If the `parameter-declaration-clause` is empty, the function takes no arguments. The parameter
list (void) is equivalent to the empty parameter list. Except for this special case, void shall not be a parameter type (though types derived from void, such as void*, can). If the `parameter-declaration-clause` terminates with an ellipsis
or a function parameter pack (\textsuperscript{14.5.3}), the number of arguments shall be equal to or greater than the number of param-
eters that do not have a default argument and are not function parameter packs. Where syntactically correct and where
“…” is not part of an abstract-declarator, “…” is synonymous with “…” [Example: the declaration

```c
int printf(const char*, ...);
```
declares a function that can be called with varying numbers and types of arguments.

```c
printf("hello world");
printf("a=%d b=%d", a, b);
```

However, the first argument must be of a type that can be converted to a `const char*` — end example ] [ Note: the standard header `<stdarg.h>` contains a mechanism for accessing arguments passed using the ellipsis (see 5.2.2 and 18.8). — end note ]

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, several transformations take place upon these types to determine the function type. Any `cv-qualifier` modifying a parameter type is deleted. [ Example: the type `void(*)(const int)` becomes `void(*)(int)` — end example ] Such `cv-qualifier` s affect only the definition of the parameter within the body of the function; they do not affect the function type. If a `storage-class-specifier` modifies a parameter type, the specifier is deleted. [ Example: `register char*` becomes `char*` — end example ] Such `storage-class-specifier` s affect only the definition of the parameter within the body of the function; they do not affect 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`.

A `cv-qualifier-seq` shall only be part of the function type for a non-static member function, the function type to which a pointer to member refers, or the top-level function type of a function typedef declaration. The effect of a `cv-qualifier-seq` in a function declarator is not the same as adding cv-qualification on top of the function type. In the latter case, the cv-qualifiers are ignored. [ Example:

```c
typedef void F();
struct S {
    const F f;
    // OK: equivalent to: void f();
};
```

— end example ] The return type, the parameter-type-list and the `cv-qualifier-seq`, but not the default arguments (8.3.6) or the exception specification (15.4), are part of the function type. [ Note: function types are checked during the assignments and initializations of pointer-to-functions, reference-to-functions, and pointer-to-member-functions. — end note ]

[ Example: the declaration

```c
int fseek(FILE*, long, int);
```

declares a function taking three arguments of the specified types, and returning `int` (7.1.6). — end example ]

If the type of a parameter includes a type of the form “pointer to array of unknown bound of `T`” or “reference to array of unknown bound of `T`,” the program is ill-formed. Functions shall not have a return type of type array or function, although they may have a return type of type pointer or reference to such things. There shall be no arrays of functions, although there can be arrays of pointers to functions. Types shall not be defined in return or parameter types. The type

---

87) This excludes parameters of type “ptr-arr-seq T2” where T2 is “pointer to array of unknown bound of `T`” and where `ptr-arr-seq` means any sequence of “pointer to” and “array of” derived declarator types. This exclusion applies to the parameters of the function, and if a parameter is a pointer to function or pointer to member function then to its parameters also, etc.

Draft
of a parameter or the return type for a function definition shall not be an incomplete class type (possibly cv-qualified) unless the function definition is nested within the member-specification for that class (including definitions in nested classes defined within the class).

7 A typedef of function type may be used to declare a function but shall not be used to define a function (8.4). [Example:

```c
typedef void F();
F   fv;       // OK: equivalent to void fv();
F   fv {}    // ill-formed
void fv() {} // OK: definition of fv
```

—end example] A typedef of a function type whose declarator includes a cv-qualifier-seq shall be used only to declare the function type for a non-static member function, to declare the function type to which a pointer to member refers, or to declare the top-level function type of another function typedef declaration. [Example:

```c
typedef int FIC(int) const;
FIC f;                           // ill-formed: does not declare a member function
struct S {
    FIC f;                       // OK
};
FIC S::*pm = &S::f;              // OK
```

—end example]

8 An identifier can optionally be provided as a parameter name; if present in a function definition (8.4), it names a parameter (sometimes called “formal argument”). [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 the parameter-declaration-clause since it goes out of scope at the end of the function declarator (3.3). —end note]

9 [Example: the declaration

```c
int i,
   *pi,
   f(),
   *fpi(int),
   (*fpi(const char*, const char*),
    (*fpi(int))(int);
```

declares an integer i, a pointer pi to an integer, a function f taking no arguments and returning an integer, a function fpi taking an integer argument and returning a pointer to an integer, a pointer pif to a function which takes two pointers to constant characters and returns an integer, a function fpif taking an integer argument and returning a pointer to a function that takes an integer argument and returns an integer. It is especially useful to compare 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 (*fpi(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 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);
```

Draft
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, if present, shall occur at the end of the parameter-declaration-list. The type $T$ of the declarator-id of the function parameter pack shall contain a template parameter pack; each template parameter pack in $T$ is expanded by the function parameter pack. [Example:

```c
template<typename... T> void f(T (* ...t)(int, int);
int add(int, int);
float subtract(int, int);
void g() {
    f(add, subtract);
}
```
— end example]

There is a syntactic ambiguity when an ellipsis occurs at the end of a parameter-declaration-clause without a preceding comma. In this case, the ellipsis is parsed as part of the abstract-declarator if the type of the parameter names a template parameter pack that has not been expanded; otherwise, it is parsed as part of the parameter-declaration-clause.\(^{88}\)

### 8.3.6 Default arguments

[decl.fct.default]

If an expression is specified in a parameter declaration this expression is used as a default argument. Default arguments will be used in calls where trailing arguments are missing.

[Example: the declaration

```c
void point(int = 3, int = 4);
```

declares a function that can be called with zero, one, or two arguments of type `int`. It can be called in any of these ways:

```c
point(1,2); point(1); point();
```

The last two calls are equivalent to `point(1,4)` and `point(3,4)`, respectively. — end example]

A default argument expression shall be specified only in the parameter-declaration-clause of a function declaration or in a template-parameter (14.1). It 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.\(^{89}\)

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, all

---

\(^{88}\)One can explicitly disambiguate the parse either by introducing a comma (so the ellipsis will be parsed as part of the parameter-declaration-clause) or by introducing a name for the parameter (so the ellipsis will be parsed as part of the declarator-id).

\(^{89}\)This means that default arguments cannot appear, for example, in declarations of pointers to functions, references to functions, or `typedef` declarations.
parameters subsequent to a parameter with a default argument shall have default arguments supplied in this or previous
declarations. A default argument shall not be redefined by a later declaration (not even to the same value). [Example:

```c
void g(int = 0, ...);
void f(int, int);
void f(int, int = 7);
void h()
{
    f(3);
    // OK, calls f(3, 7)
    void f(int = 1, int);
    // error: does not use default
    // from surrounding scope
}

void m()
{
    void f(int, int);
    // has no defaults
    f(4);
    // error: wrong number of arguments
    void f(int, int = 5);  // OK
    f(4);
    // OK, calls f(4, 5);
    void f(int, int = 5);  // error: cannot redefine, even to
    // same value
}
void n()
{
    f(6);
    // OK, calls f(6, 7)
}
```
— end example] For a given inline function defined in different translation units, the accumulated sets of default
arguments at the end of the translation units shall be the same; see 3.2. If a friend declaration specifies a default
argument expression, that declaration must be a definition and shall be the only declaration of the function or function
template in the translation unit.

A default argument expression is implicitly converted (clause 4) to the parameter type. The default argument expression
has the same semantic constraints as the initializer expression in a declaration of a variable of the parameter type,
using the copy-initialization semantics (8.5). The names in the expression are bound, and the semantic constraints are
checked, at the point where the default argument expression 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))
    }
```

Draft
6 Except for member functions of class templates, the default arguments in a member function definition that appears outside of the class definition are added to the set of default arguments provided by the member function declaration in the class definition. Default arguments for a member function of a class template shall be specified on the initial declaration of the member function within the class template. [Example:

```cpp
class C {
    void f(int i = 3);
    void g(int i, int j = 99);
};

void C::f(int i = 3) // error: default argument already
{}  // specified in class scope
void C::g(int i = 88, int j) // in this translation unit,
{}  // C::g can be called with no argument

— end example ]
```

7 Local variables shall not be used in default argument expressions. [Example:

```cpp
void f()
{
    int i;
    extern void g(int x = i);  //error
    // ...
}

— end example ]
```

8 The keyword this shall not be used in a default argument of a member function. [Example:

```cpp
class A {
    void f(A* p = this) { }  // error
};

— end example ]
```

9 Default arguments are evaluated each time the function is called. The order of evaluation of function arguments is unspecified. Consequently, parameters of a function shall not be used in default argument expressions, even if they are not evaluated. Parameters of a function declared before a default argument expression 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
```

Draft
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 expression, 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.

    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:

    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.

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:

    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 [dcl.fct.def]

1 Function definitions have the form

\[
\text{function-definition:}
\begin{align*}
& \text{decl-specifier-seq_opt} \quad \text{declarator} \quad \text{ctor-initializer_opt} \quad \text{function-body} \\
& \text{decl-specifier-seq_opt} \quad \text{declarator function-try-block}
\end{align*}
\]

\[
\text{function-body:}
\begin{align*}
& \text{ctor-initializer_opt} \quad \text{compound-statement} \\
& \text{function-try-block}
\end{align*}
\]

Any informal reference to the body of a function should be interpreted as a reference to the non-terminal function-body.

2 The declarator in a function-definition shall have the form

\[
D1 \ (\text{parameter-declaration-clause}) \ cv-qualifier-seq_opt \ exception-specification_opt
\]

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

\[
\text{int max(int a, int b, int c) }
\]
\[
\{ \\
\text{ int m = (a > b) ? a : b; } \\
\text{ return (m > c) ? m : c; } \\
\}
\]

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 A cv-qualifier-seq can be part of a non-static member function declaration, non-static member function definition, or pointer to member function only; see 9.3.2. It is part of the function type.

6 [Note: unused parameters need not be named. For example,

\[
\text{void print(int a, int) }
\]
\[
\{ \\
\text{ std::printf("a = %d\n",a); } \\
\}
\]

— end note ]

7 In the function-body, a function-local predefined variable denotes a local object of static storage duration that is implicitly defined (see 3.3.2).
The function-local predefined variable `__func__` is defined as if a definition of the form

```c
static const char __func__[] = "function-name";
```

had been provided, where `function-name` is an implementation-defined string. It is unspecified whether such a variable has an address distinct from that of any other object in the program.¹

[Example:

```c
struct S {
    S() : s(__func__) { } // OK
    const char *s;
};
void f(const char * s = __func__); // error: __func__ is undeclared
```

—end example]

A function definition of the form:

```c
decl-specifier-seqopt declarator = default;
```

is called an **explicitly-defaulted** definition. Only special member functions may be explicitly defaulted, and the implementation shall define them as if they had implicit definitions (12.1, 12.4, 12.8). A special member function is **user-provided** if it is user-declared and not explicitly defaulted on its first declaration. A user-provided explicitly-defaulted function is defined at the point where it is explicitly defaulted. [Note: while an implicitly-declared special member function is inline (clause 12), an explicitly-defaulted definition may be non-inline. Non-inline definitions are user-provided, and hence non-trivial (12.1, 12.4, 12.8). This rule enables efficient execution and concise definition while enabling a stable binary interface to an evolving code base. —end note]

[Example:

```c
struct trivial {
    trivial() = default;
    trivial(const trivial&) = default;
    trivial& operator =(const trivial&) = default;
    ~trivial() = default;
};

struct nontrivial1 {
    nontrivial1();
};
nontrivial1::nontrivial1() = default; // not inline

struct nontrivial2 {
    nontrivial2();
};
inline nontrivial2::nontrivial2() = default; // not first declaration

struct nontrivial3 {
    virtual ~nontrivial3() = 0;
    // virtual
};
```

¹Implementations are permitted to provide additional predefined variables with names that are reserved to the implementation (17.4.3.1.2). If a predefined variable is not used (3.2), its string value need not be present in the program image.

Draft
inline nontrivial3::nontrivial3() = default; // not first declaration

— end example ]

A function definition of the form:

dcl-specifier-seqopt declarator = delete;

is called a deleted definition. A function with a deleted definition is also called a deleted function. A deleted definition of a function shall be the first declaration of the function. [ Example:

struct sometype {
    sometype();
};
sometype::sometype() = delete; // ill-formed; not first declaration

— end example ] A deleted function is implicitly inline. [ Note: the one-definition rule (3.2) applies to deleted definitions. — end note ] 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 ] [ Example: One can enforce non-default initialization and non-integral initialization with

struct sometype {
    sometype() = delete;  // redundant, but legal
    sometype(std::intmax_t) = delete;
    sometype(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.

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 *p = new sometype[3];  // error, deleted class operator new[]

— end example ]

8.5 Initializers [dcl.init]

A declarator can specify an initial value for the identifier being declared. The identifier designates an object or reference 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).

initializer:
    = initializer-clause
    ( expression-list )

Draft
8.5 Initializers

Declarators 178

initializer-clause:
  assignment-expression
  \{ initializer-list , opt \} \{
  \}

initializer-list:
  initializer-clause \ldots opt
  initializer-list , initializer-clause \ldots opt

2 Automatic, register, static, and external variables of namespace scope can be initialized by arbitrary expressions involving literals and previously declared variables and functions. [Example:

    int f(int);
    int a = 2;
    int b = f(a);
    int c(b);

    — end example ]

3 [ Note: default argument expressions are more restricted; see 8.3.6.]

4 The order of initialization of static objects is described in 3.6 and 6.7. — end note ]

5 To zero-initialize an object of type T means:
  — if T is a scalar type (3.9), the object is set to the value 0 (zero), taken as an integral constant expression, converted to T;\(^{91}\)
  — if T is a non-union class type, each non-static data member and each base-class subobject is zero-initialized;
  — if T is a union type, the object’s first named data member\(^{92}\) is zero-initialized;
  — if T is an array type, each element is zero-initialized;
  — if T is a reference type, no initialization is performed.

To default-initialize an object of type T means:
  — if T is a non-POD\(^{91}\) trivial class type (clause 9), the default constructor for T is called (and the initialization is ill-formed if T has no accessible default constructor);\(^{93}\)
  — if T is an array type, each element is default-initialized;
  — otherwise, the object is zero-initialized.

To value-initialize an object of type T means:
  — if T is a class type (clause 9) with a user-declared\(^{94}\) provided constructor (12.1), then the default constructor for T is called (and the initialization is ill-formed if T has no accessible default constructor);
  — if T is a non-union class type without a user-declared\(^{94}\) provided constructor, then every non-static data member and base-class component of T is value-initialized;\(^{95}\)

---

\(^{91}\)As specified in 4.10, converting an integral constant expression whose value is 0 to a pointer type results in a null pointer value.

\(^{92}\)This member must not be static, by virtue of the requirements in 9.5.

\(^{93}\)Value-initialization for such a class object may be implemented by zero-initializing the object and then calling the default constructor.

Draft
— if \( T \) is an array type, then each element is value-initialized;
— otherwise, the object is zero-initialized

6 A program that calls for default-initialization or value-initialization of an entity of reference type is ill-formed. If \( T \) is a cv-qualified type, the cv-unqualified version of \( T \) is used for these definitions of zero-initialization, default-initialization, and value-initialization.

7 Every object of static storage duration shall be zero-initialized at program startup before any other initialization takes place. [Note: in some cases, additional initialization is done later. — end note]

8 An object whose initializer is an empty set of parentheses, i.e., \((\)\), shall be value-initialized.

[Note: since \((\)\) is not permitted by the syntax for initializer,

\[ X\;\text{a}(); \]

is not the declaration of an object of class \( X \), but the declaration of a function taking no argument and returning an \( X \). The form \((\)\) is permitted in certain other initialization contexts (5.3.4, 5.2.3, 12.6.2). — end note]

9 If no initializer is specified for an object, and the object is of (possibly cv-qualified) non-POD trivial class type (or array thereof), the object shall be default-initialized; if the object is of const-qualified type, the underlying class type shall have a user-declared default constructor. Otherwise, if no initializer is specified for a non-static object, the object and its subobjects, if any, have an indeterminate initial value\(^{94}\); if the object or any of its subobjects are of const-qualified type, the program is ill-formed.

10 An initializer for a static member is in the scope of the member’s class. [Example:

\begin{verbatim}
int a;

struct X {
  static int a;
  static int b;
};

int X::a = 1;
int X::b = a; //X::b = X::a
\end{verbatim}

— end example]

11 The form of initialization (using parentheses or =) is generally insignificant, but does matter when the entity being initialized has a class type; see below. A parenthesized initializer can be a list of expressions only when the entity being initialized has a class type.

12 The initialization that occurs in argument passing, function return, throwing an exception (15.1), handling an exception (15.3), and brace-enclosed initializer lists (8.5.1) is called \textit{copy-initialization} and is equivalent to the form

\begin{verbatim}
T\;x = a;
\end{verbatim}

13 The initialization that occurs in \texttt{new} expressions (5.3.4), \texttt{static_cast} expressions (5.2.9), functional notation type conversions (5.2.3), and base and member initializers (12.6.2) is called \textit{direct-initialization} and is equivalent to the form

\(^{94}\) This does not apply to aggregate objects with automatic storage duration initialized with an incomplete brace-enclosed initializer-list; see 8.5.1.


\[ T \ x(a); \]

14 If \( T \) is a scalar type, then a declaration of the form

\[
T \ x = \{ \ a \ }; \\
\]

is equivalent to

\[
T \ x = a; \\
\]

15 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. The source type is not defined when the initializer is brace-enclosed or when it is a parenthesized list of expressions.

- If the destination type is a reference type, see 8.5.3.
- 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.
- Otherwise, if the destination type is an array, see 8.5.1.
- If the destination type is a (possibly cv-qualified) class type:
  - If the class is an aggregate (8.5.1), and the initializer is a brace-enclosed list, see 8.5.1.
  - 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(s) as its argument(s). If no constructor applies, or the overload resolution is ambiguous, the initialization is ill-formed.
  - Otherwise (i.e., for the remaining copy-initialization cases), user-defined conversion sequences that can convert from the source type to the destination type or (when a conversion function is used) to a derived class thereof are enumerated as described in 13.3.1.4, and the best one is chosen through overload resolution (13.3). 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 an rvalue. 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. [Note: an expression of type “cv1 T” can initialize an object of type “cv2 T” independently of the cv-qualifiers cv1 and cv2.]

```c
int a;
const int b = a;
int c = b;
```

— end note ] 16 An `initializer-clause` followed by an ellipsis is a pack expansion (14.5.3).

### 8.5.1 Aggregates
[decl.init.aggr]

1 An `aggregate` is an array or a class (clause 9) with no user-declared 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).

2 When an aggregate is initialized the `initializer` can contain an `initializer-clause` consisting of a brace-enclosed, comma-separated list of `initializer-clause`s for the members of the aggregate, written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the subaggregate. [Example:

```c
struct A {
    int x;
    struct B {
        int i;
        int j;
    } b;
} a = { 1, { 2, 3 } };
```

initializes `a.x` with 1, `a.b.i` with 2, `a.b.j` with 3. — end example ]

3 An aggregate that is a class can also be initialized with a single expression not enclosed in braces, as described in 8.5.

4 An array of unknown size initialized with a brace-enclosed `initializer-list` containing `n` `initializer-clause`s, where `n` shall be greater than zero, is defined as having `n` elements (8.3.4). [Example:

```c
int x[] = { 1, 3, 5 };
```

declares and initializes `x` as a one-dimensional array that has three elements since no size was specified and there are three initializers. — end example ] An empty `initializer-list` `{}` shall not be used as the `initializer-clause` for an array of unknown bound.\(^{(95)}\)

5 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;
```

\(^{(95)}\) The syntax provides for empty `initializer-list`s, but nonetheless C++ does not have zero length arrays.

Draft
8.5 Initializers

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-clause s exceeds the number of members or elements to initialize. [Example:

char cv[4] = { 'a', 's', 'd', 'f', 0 }; // error

is ill-formed. — end example]

If there are fewer initializer-clause s in the list than there are members in the aggregate, then each member not explicitly initialized shall be value-initialized (8.5). [Example:

struct S { int a; char* b; int c; }; S ss = { 1, "asdf" );

initializes ss.a with 1, ss.b with "asdf", and ss.c with the value of an expression of the form int(), that is, 0. — 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-clause s for all members of C following m are also omitted. [Example:

struct S { } s;
struct A {
    S s;
    int i;
} a = { { }, 3 };

If an incomplete or empty initializer-list leaves a member of reference type uninitialized, the program is ill-formed.

Draft
When initializing a multi-dimensional array, the *initializer-clause* s initialize the elements with the last (rightmost) 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 ]

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-clause* s initializes the members of a subaggregate; it is erroneous for there to be more *initializer-clause* s than members. If, however, the *initializer-list* for a subaggregate does not begin with a left brace, then only enough *initializer-clause* s from the list are taken to initialize the members of the subaggregate; any remaining *initializer-clause* s 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 `float()`, that is, are initialized with \( 0.0 \). In the following example, braces in the *initializer-list* are elided; however the *initializer-list* has the same effect as the completely-braced *initializer-list* of the above example,

```c
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for \( y \) begins with a left brace, but the one for \( y[0] \) does not, therefore three elements from the list are used. Likewise the next three are taken successively for \( y[1] \) and \( y[2] \). — end example ]

All implicit type conversions (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();
};
```
8.5 Initializers

struct B {
    A a1, a2;
    int z;
};

A a;
B b = { 4, a, a };

Braces are elided around the initializer-clause for b.a1. 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-declared constructor (12.1). Initialization of these aggregate objects is described in 12.6.1. — end note ]

When an aggregate with static storage duration is initialized with a brace-enclosed initializer-list, if all the member initializer expressions are constant expressions, and the aggregate is a POD trivial type, the initialization shall be done during the static phase of initialization (3.6.2); otherwise, it is unspecified whether the initialization of members with constant expressions takes place during the static phase or during the dynamic phase of initialization.

When a union is initialized with a brace-enclosed initializer, the braces shall only contain an initializer-clause for the first member of the union. [ Example:

    union u { int a; 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 ]

The full-expressions in an initializer-clause are evaluated in the order in which they appear.

8.5.2 Character arrays [dcl.init.string]

A char array (whether plain char, signed char, or unsigned char), char16_t array, char32_t array, or wchar_t array can be initialized by a string-literal (optionally enclosed in braces) with no prefix, with a u prefix, with a U prefix, or with an L prefix, respectively; successive characters of the string-literal initialize the members of the array. [ Example:

    char msg[] = "Syntax error on line %s\n";

shows a character array whose members are initialized with a string-literal. Note that because '\n' is a single character and because a trailing '\0' is appended, sizeof(msg) is 25. — end example ]

There shall not be more initializers than there are array elements. [ Example:

    char cv[4] = "asdf"; // error

Draft
is ill-formed since there is no space for the implied trailing ‘\0’. — end example ]

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. For purposes of overload resolution, cases for which cv1 is greater cv-qualification than cv2 are identified as reference-compatible with added qualification (see 13.3.3.2). 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:

— If the initializer expression
  — is an lvalue (but is not a bit-field), and “cv1 T1” is reference-compatible with “cv2 T2,” or
— has a class type (i.e., T2 is a class type) and can be implicitly converted to an lvalue of type “cv3 T3,”
where “cv1 T1” is reference-compatible with “cv3 T3”\(^{96}\) (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 directly to the initializer expression lvalue in the first case, and the reference is bound
to the lvalue result of the conversion in the second case. In these cases the reference is said to bind directly to the
initializer expression. [Note: the usual lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3)
standard conversions are not needed, and therefore are suppressed, when such direct bindings to lvalues are done.
— end note ]

[Example:]

double d = 2.0;
double& rd = d; // rd refers to d
const double& rcd = d; // rcd refers to d

struct A { };
struct B : public A { } b;
A& ra = b; // ra refers to a subobject in b
const A& rca = b; // rca refers to a subobject in b

— end example ]

— Otherwise, the reference shall be an lvalue reference to a non-volatile const type (i.e., cv1 shall be const), or shall
be an rvalue reference. [ Example: ]

double& rd2 = 2.0; // error: not an lvalue and reference not const
int i = 2;
double& rd3 = i; // error: type mismatch and reference not const
double&& rd4 = i; // OK: reference bound to temporary double

— end example ]

— If the initializer expression is an rvalue, with T2 a class type, and “cv1 T1” is reference-compatible with “cv2 T2,”
the reference is bound to the object represented by the rvalue (see 3.10) or to a sub-object within that
object.

[ Example: ]

struct A { };
struct B : public A { } b;
extern B f();
const A& rca = f(); // Bound to the A subobject of the B rvalue.
A&& rcb = f(); // Same as above

— end example ]

— If the initializer expression is an rvalue, with T2 an array type, and “cv1 T1” is reference-compatible with “cv2 T2,”
the reference is bound to the object represented by the rvalue (see 3.10).


\(^{96}\) This requires a conversion function (12.3.2) returning a reference type.
— Otherwise, a temporary of type "cv1 T1" is created and initialized from the initializer expression using the rules for a non-reference copy initialization (8.5). The reference is then bound to the temporary. If T1 is reference-related to T2, cv1 must be the same cv-qualification as, or greater cv-qualification than, cv2; otherwise, the program is ill-formed. [Example:

```c
const double& rcd2 = 2;  // rcd2 refers to temporary with value 2.0
double&& rcd3 = 2;        // rcd3 refers to temporary with value 2.0
const volatile int cvi = 1;
const int& r = cvi;        // error; type qualifiers dropped
```

— end example]

[Note: 12.2 describes the lifetime of temporaries bound to references. — end note]
Chapter 9  Classes

1 A class is a type. Its name becomes a class-name (9.1) within its scope.

   class-name:
       identifier
       simple-template-id

Class-specifier s and elaborated-type-specifier s (7.1.6.3) are used to make class-name s. An object of a class consists of a (possibly empty) sequence of members and base class objects.

   class-specifier:
       class-head { member-specification_opt }

   class-head:
       class-key identifier_opt base-clause_opt
       class-key nested-name-specifier identifier base-clause_opt
       class-key nested-name-specifier_opt simple-template-id base-clause_opt

   class-key:
       class
       struct
       union

A class-specifier where the class-head omits the optional identifier defines an unnamed class.

2 A class-name is inserted into the scope in which it is declared immediately after the class-name is seen. The class-name is also inserted into the scope of the class itself; this is known as the injected-class-name. For purposes of access checking, the injected-class-name is treated as if it were a public member name. A class-specifier is commonly referred to as a class definition. A class is considered defined after the closing brace of its class-specifier has been seen even though its member functions are in general not yet defined.

3 Complete objects and member subobjects of class type shall have nonzero size.\(^\text{97}\) [Note: class objects can be assigned, passed as arguments to functions, and returned by functions (except objects of classes for which copying has been restricted; see 12.8). Other plausible operators, such as equality comparison, can be defined by the user; see 13.5. — end note]

4 A union is a class defined with the class-key union; it holds only one data member at a time (9.5). [Note: aggregates of class type are described in 8.5.1. — end note]

5 A trivial class is a class that:
   — has a trivial default constructor (12.1).

\(^{97}\)Base class subobjects are not so constrained.
— has a trivial copy constructor (12.8).
— has a trivial copy assignment operator (13.5.3, 12.8), and
— has a trivial destructor (12.4).

[Note: in particular, a trivial class does not have virtual functions or virtual base classes. — end note]

6 A standard-layout class is a class that:
— has no non-static data members of type non-standard-layout class (or array of such types) or reference,
— has no virtual functions (10.3) and no virtual base classes (10.1),
— has the same access control (clause 11) for all non-static data members,
— has no non-standard-layout base classes,
— either has no non-static data members in the most-derived class and at most one base class with non-static data members, or has no base classes with non-static data members, and
— has no base classes of the same type as the first non-static data member.\(^{98}\)

7 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.

8 [Note: standard-layout classes are useful for communicating with code written in other programming languages. Their layout is specified in 9.2. — end note]

9 A POD class is an aggregate class that has no non-static data members of non-POD type (or array of such a type) or reference, and has no user declared copy assignment operator and no user declared destructor. A POD struct is a POD class defined with the class-key struct or the class-key class. A POD union is a POD class defined with the class-key union. A POD struct is a class that is both a trivial class and a standard-layout class, and has no non-static data members of type non-POD 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:

```c
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
```]

\(^{98}\): 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).
9.1 Class names

1 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

```c
a1 = a2;       // error: Y assigned to X
a1 = a3;       // error: int assigned to X
```

are type mismatches, and that

```c
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 ]

2 A class declaration introduces the class name into the scope where it is declared and hides any class, object, function, or other declaration of that name in an enclosing scope (3.3). If a class name is declared in a scope where an object, 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 {
    // ...
};
```

— end example ]

10 If a class-head 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 (i.e., neither inherited nor introduced by a using-declaration), and the class-specifier shall appear in a namespace enclosing the previous declaration.
9.1 Class names

stat gstat; // use plain stat to
    // define variable

int stat(struct stat*); // redeclare stat as function

void f()
{
    struct stat* ps; // struct prefix needed
    // to name struct stat
    // ...
    stat(ps);       // call stat()
    // ...
}

—end example—

A declaration consisting solely of class-key identifier; is either a redeclaration of the name in the
current scope or a forward declaration of the identifier as a class name. It introduces the class name into the current
scope. [Example:

struct s { int a; };

void g()
{
    struct s; // hide global struct s
    // with a local declaration
    s* p;     // refer to local struct s
    struct s { char* p; }; // define local struct s
    struct s; // redeclaration, has no effect
}

—end example—

[Note: Such declarations allow definition of classes that refer to each other. [Example:

class Vector;

class Matrix {
    // ...
    friend Vector operator*(Matrix&, Vector&);
};

class Vector {
    // ...
    friend Vector operator*(Matrix&, Vector&);
};

Declaration of friends is described in 11.4, operator functions in 13.5. —end example— end note]
void g(int s)
{
    struct s* p = new struct s; // global s
    p->a = s; // local s
}
— end example ]

[ 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,

```cpp
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 ]

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

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
9.2 Class members

Classes

9.2.1 Class members

Enumeration (7.2) defined in the class are members of the class. Except when used to declare friends (11.4) or to introduce the name of a member of a base class into a derived class (7.3.3, 11.3), 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.

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, and exception-specifications, and constructor-ctor-initializers (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 member-declarator can contain a constant-initializer only if it declares a static member (9.4) of const integral or const enumeration literal type, see 9.4.2.

A member can be initialized using a constructor; see 12.1. [Note: see clause 12 for a description of constructors and other special member functions. — end note]

A member shall not be declared to have automatic storage duration (auto, register) or with the extern storage-class-specifier.

The decl-specifier-seq is omitted in constructor, destructor, and conversion function declarations only. The member-declarator-list can be omitted only after a class-specifier or an enum-specifier or in a friend declaration (11.4). A pure-specifier shall be used only in the declaration of a virtual function (10.3).

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.

[Note: See 5.1 for restrictions on the use of non-static data members and non-static member functions. — end note]

[Note: the type of a non-static member function is an ordinary function type, and the type of a non-static data member is an ordinary object type. There are no special member function types or data member types. — end note]

[Example: A simple example of a class definition is

```
struct tnode {
    char tword[20];
    int count;
    tnode *left;
    tnode *right;
};
```

which contains an array of twenty characters, an integer, and two pointers to objects of the same type. Once this definition has been given, the declaration

```
tnode s, *sp;
```

declares s to be a tnode and sp to be a pointer to a tnode. With these declarations, sp->count refers to the count member of the object to which sp points; s.left refers to the left subtree pointer of the object s; and s.right->tword[0] refers to the initial character of the tword member of the right subtree of s. — end example]
Nonstatic data members of a (non-union) class declared without an intervening access specifier 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 separated by an access specifier with different access control is unspecified (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).

If T is the name of a class, then each of the following shall have a name different from T:

- every static data member of class T;
- every member function of class T [Note: this restriction does not apply to constructors, which do not have names (12.1) — end note];
- every member of class T that is itself a type;
- every enumerator of every member of class T that is an enumerated type; and
- every member of every anonymous union that is a member of class T.

In addition, if class T has a user-declared constructor (12.1), every non-static data member of class T shall have a name different from T.

Two POD-standard-layout struct (clause 9) types are layout-compatible if they have the same number of non-static data members, and corresponding non-static data members (in declaration order) have layout-compatible types (3.9).

Two POD-standard-layout union (clause 9) types 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).

If a POD-standard-layout union contains two or more POD-standard-layout structs that share a common initial sequence, and if the POD-standard-layout union object currently contains one of these POD-standard-layout structs, it is permitted to inspect the common initial part of any of them. Two POD-standard-layout structs share a common initial sequence if corresponding members have layout-compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members.

A pointer to a POD-standard-layout struct object, suitably converted using a reinterpret_cast, points to its initial member (or if that member is a bit-field, then to the unit in which it resides) and vice versa. [Note: There might therefore be unnamed padding within a POD-standard-layout struct object, but not at its beginning, as necessary to achieve appropriate alignment. — end note]

9.3 Member functions

Functions declared in the definition of a class, excluding those declared with a friend specifier (11.4), 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).

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.
9.3 Member functions

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 external linkage. Member functions of a local class (9.8) have no linkage. See 3.5. — end note]

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.

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, or for a constructor function (12.1), in a mem-initializer expression (12.6.2)) 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 `count` refers to the static data member `count` declared in class `X`. — end example]

A static local variable in a member function always refers to the same object, whether or not the member function is inline.

Member functions may be mentioned in friend declarations after their class has been defined.

Member functions of a local class shall be defined inline in their class definition, if they are defined at all.

[Note: a member function can be declared (but not defined) using a typedef for a function type. The resulting member function has exactly the same type as it would have if the function declarator were provided explicitly, see 8.3.5. For example,

```c
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

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.

1. from within the body of a member function of its class or of a class derived from its class, or
2. from within the body of a member function of its class or of a class derived from its class, or
3. from a mem-initializer (12.6.2) for a constructor for its class or for a class derived from its class.

If a non-static member function of a class X is called for an object that is not of type X, or of a type derived from X, the behavior is undefined.

When an id-expression (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 the body of a non-static member function of class X or used in the mem-initializer for a constructor of class X, if name lookup (3.4.1) resolves the name in the id-expression to a non-static non-type member of some class C, 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:

```c
struct tnode {
    char tword[20];
    int count;
    tnode *left;
    tnode *right;
    void set(char*, tnode* l, tnode* r);
};

void tnode::set(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.][99] —end example]

[99]See, for example, <cstring> (21.4).
9.3 Member functions

4 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:

```c++
struct X {
    void g() const;
    void h() const volatile;
};
```

X::g is a const member function and X::h is a const volatile member function. — end example]

5 A non-static member function may be declared virtual (10.3) or pure virtual (10.4).

9.3.2 The this pointer

1 In the body of a non-static (9.3) member function, the keyword this is a non-va

lue rvalue 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*.

2 In a const member function, the object for which the function is called is accessed through a const access path; therefore, a const member function shall not modify the object and its non-static data members. [Example:

```c++
struct s {
    int a;
    int f() const;
    int g() { return a++; } // error
    int h() const { return a++; }
};
```

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]

3 Similarly, volatile semantics (7.1.6.1) apply in volatile member functions when accessing the object and its non-static data members.

4 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:

```c++
void k(s& x, const s& y)
{
    x.f();
    x.g();
    y.f();
    y.g(); // error
}
```

Draft
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

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
class process {
    public:
        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 ]
```

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]
9.4 Static members

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 [class.static.mfct]

1 [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 [class.static.data]

1 A static data member is not part of the subobjects of a class. There is only one copy of a static data member 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 (3.3.6). [Example:

```cpp
class process {
    static process* run_chain;
    static process* running;
};
```

```cpp
process* process::running = get_main();
process* process::run_chain = running;
```

The static data member run_chain of class process is defined in global scope; the notation process::run_chain specifies that the member run_chain is a member of class process and in the scope of class process. In the static data member definition, the initializer expression refers to the static data member running of class process. —end example]

[Note: once the static data member has been defined, it exists even if no objects of its class have been created. [Example: in the example above, run_chain and running exist even if no objects of class process are created by the program. —end example] —end note]

3 If a static data member is of const integral or const enumeration literal type, its declaration in the class definition may can specify a constant-initializer whose constant-expression shall be an integral constant expression (5.19). 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 constant-initializer. In that case, both these cases, the member may appear in integral constant expressions. The member shall still be defined in a namespace scope if it is used in the program and the namespace scope definition shall not contain an initializer.

4 There shall be exactly one definition of a static data member that is used in a program; no diagnostic is required; see 3.2. 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 external linkage (3.5). A local class shall not have static data members.

Static data members are initialized and destroyed exactly like non-local objects (3.6.2, 3.6.3).

A static data member shall not be mutable (7.1.1).

Unions

In a union, at most one of the data members can be active at any time, that is, the value of at most one of the 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 POD-standard-layout union contains several POD-standard-layout structs that share a common initial sequence (9.2), and if an object of this POD-standard-layout union type contains one of the POD-standard-layout structs, it is permitted to inspect the common initial sequence of any of POD-standard-layout struct members; see 9.2. — end note] The size of a union is sufficient to contain the largest of its data members. Each data member is allocated as if it were the sole member of a struct. 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. An object of a non-trivial class (clause 9) with a non-trivial default constructor (12.1), a non-trivial copy constructor (12.8), a non-trivial destructor (12.4), or a non-trivial copy assignment operator (13.5.3, 12.8) cannot be a member of a union, nor can an array of such objects. If a union contains a static data member, or a member of reference type, the program is ill-formed.

A union of the form

```c
union { member-specification }; 
```

is called an anonymous union; it defines an unnamed object of unnamed type. The member-specification of an anonymous union shall only define non-static data members. [Note: nested types 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; 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 or pointers are declared is not an anonymous union. [Example:

```c
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]

9.6 Bit-fields

A member-declarator of the form

```
identifier_opt : constant-expression
```

specifies a bit-field; its length is set off from the bit-field name by a colon. The bit-field attribute is not part of the type of the class member. The `constant-expression` shall be an integral `constant-expression` `constant expression` with a value greater than or equal to zero. The `constant-expression` 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. [Note: bit-fields straddle allocation units on some machines and not on others. Bit-fields are assigned right-to-left on some machines, left-to-right on others. — end note]

A declaration for a bit-field that omits the `identifier` declares an unnamed bit-field. Unnamed bit-fields are not members and cannot be initialized. [Note: an unnamed bit-field is useful for padding to conform to externally-imposed layouts. — end note] As a special case, an unnamed bit-field with a width of zero specifies alignment of the next bit-field at an allocation unit boundary. Only when declaring an unnamed bit-field may the `value of the constant-expression` be a `value equal to zero`.

A bit-field shall not be a static member. A bit-field shall have integral or enumeration type (3.9.1). It is implementation-defined whether a plain (neither explicitly signed nor unsigned) `char`, `short`, `int` or `long` bit-field is signed or unsigned. 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). [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. [Example:

```c
enum BOOL { FALSE=0, TRUE=1 };  
struct A {
    BOOL b:1;
};
    A a;
    void f() {
        a.b = TRUE;
    }
```
A class can be declared within another class. A class declared within another is called a *nested* class. The name of a nested class is local to its enclosing class. The nested class is in the scope of its enclosing class. [Note: see 5.1 for restrictions on the use of non-static data members and non-static member functions. —end note]

**Example:**

```cpp
int x;
int y;

class enclose {
   public:
      int x;
      static int s;

      class 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
   }
};
```

Member functions and static data members of a nested class can be defined in a namespace scope enclosing the definition of their class. [Example:

```cpp
class enclose {
   public:
      class 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
};
```
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 can use only type names, static variables, extern variables and functions, and enumerators from the enclosing scope. [Example:

```c
int x;
void f() {
    static int s;
    int x;
    extern int g();

    struct local {
        int g() { return x; } // error: x is auto
        int h() { return s; } // OK
        int k() { return ::x; } // OK
        int l() { return g(); } // OK
    };
    // ...
}

local* p = 0; // error: local not in scope
```

Draft
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.

If class X is a local class a nested class Y may be declared in class X and later defined in the definition of class X or be later defined in the same scope as the definition of class X. A class nested within a local class is a local class.

A local class shall not have static data members.

9.9 Nested type names

Type names obey exactly the same scope rules as other names. In particular, type names defined within a class definition cannot be used outside their class without qualification. [Example:

```cpp
class X {
public:
    typedef int I;
    class Y { /* ... */ };  // error
    I a;
};

I b;  // error
Y c;  // error
X::Y d;  // OK
X::I e;  // OK
```

— end example ]
Chapter 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:
      :: opt nested-name-specifier opt class-name
      virtual access-specifier opt :: opt nested-name-specifier opt class-name
      access-specifier virtual opt :: opt nested-name-specifier opt class-name

   access-specifier:
      private
      protected
      public

2 The class-name in a base-specifier shall not be an incompletely defined class (clause class); 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 (3.3.8). 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 class.access 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 class.access.base. 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:

   class Base {
      public:
         int a, b, c;
   };

   A list of base classes can be specified in a class definition using the notation:
10.1 Multiple base classes

A class can be derived from any number of base classes. [Note: the use of more than one direct base class is often called multiple inheritance. — end note] [Example:

```cpp
class A { /* ... */
};
class B { /* ... */
};
class C { /* ... */
};
class D : public A, public B, public C { /* ... */
};
```
— end example ]

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."

```
Base
   ↑
Derived1
   ↑
Derived2
```

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 class.base.init. — 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 class); 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

A class can be derived from any number of base classes. [Note: the use of more than one direct base class is often called multiple inheritance. — end note] [Example:

```cpp
class Derived : public Base {
public:
    int b;
};

class Derived2 : public Derived {
public:
    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 ]

Draft
2 [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, class.access.spec). — end note]

3 A class shall not be specified as a direct base class of a derived class more than once. [Note: a class can be an indirect base class more than once and can be a direct and an indirect base class. There are limited things that can be done with such a class. The non-static data members and member functions of the direct base class cannot be referred to in the scope of the derived class. However, the static members, enumerations and types can be unambiguously referred to. — end note] [Example:

```cpp
class X { /* ... */
};
class Y : public X, public X { /* ... */ }; // ill-formed

class L { public: int next; /* ... */
};
class A : public L { /* ... */
};
class B : public L { /* ... */
};
class C : public A, public B { void f(); /* ... */ }; // well-formed

class D : public A, public L { void f(); /* ... */ }; // well-formed

— end example ]

4 A base class specifier that does not contain the keyword virtual, specifies a non-virtual base class. A base class specifier that contains the keyword virtual, specifies a virtual base class. For each distinct occurrence of a non-virtual base class in the class lattice of the most derived class, the most derived object (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.

```
    C
   / \  
  A   B
 /   \
 L   L
```

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).

5 For another example,

```cpp
class V { /* ... */
};
class A : virtual public V { /* ... */
};
class B : virtual public V { /* ... */
};
class C : public A, public B { /* ... */
};
```

Draft
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.

![Diagram showing class C with virtual base class V and subobjects A, B, and C sharing a single subobject of type V]

7 A class can have both virtual and non-virtual base classes of a given type.

```cpp
class B { /* ... */ };  // non-virtual base
class X : virtual public B { /* ... */ };  // virtual base
class Y : virtual public B { /* ... */ };  // virtual base
class Z : public B { /* ... */ };  // non-virtual base
class AA : public X, public Y, public Z { /* ... */ };  // AA inherits from X, Y, and 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.

![Diagram showing class AA with virtual base B and subobjects B, X, Y, and Z sharing a single subobject of type B]

— end example ]

10.2 Member name lookup

Member name lookup determines the meaning of a name (id-expression) in a class scope (3.3.6). 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 class.access).

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-declaration s) were found. In the declaration set, using-declaration s are replaced by the members they designate, and type declarations (including injected-class-names) are replaced by the types they designate. \( S(f, C) \) is calculated as follows:
If \( C \) contains a declaration of the name \( f \), the declaration set contains every declaration of \( f \) declared in \( C \) that satisfies the requirements of the language construct in which the lookup occurs. [Note: Looking up a name in an elaborated-type-specifier (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, object, 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 in the same scope. — end note] If the resulting declaration set is not empty, the subobject set contains \( C \) itself, and calculation is complete.

Otherwise (i.e., \( C \) does not contain a declaration of \( f \) or the resulting declaration set is empty), \( S(f,C) \) is initially empty. If \( C \) has base classes, calculate the lookup set for \( f \) in each direct base class subobject \( B_i \), and merge each such lookup set \( S(f,B_i) \) in turn into \( S(f,C) \).

The following steps define the result of merging lookup set \( S(f,B_i) \) into the intermediate \( S(f,C) \):

- If each of the subobject members of \( S(f,B_i) \) is a base class subobject of at least one of the subobject members of \( S(f,C) \), or if \( S(f,B_i) \) is empty, \( S(f,C) \) is unchanged and the merge is complete. Conversely, if each of the subobject members of \( S(f,C) \) is a base class subobject of at least one of the subobject members of \( S(f,B_i) \), or if \( S(f,C) \) is empty, the new \( S(f,C) \) is a copy of \( S(f,B_i) \).

- Otherwise, if the declaration sets of \( S(f,B_i) \) and \( S(f,C) \) differ, the merge is ambiguous: the new \( S(f,C) \) is a lookup set with an invalid declaration set and the union of the subobject sets. In subsequent merges, an invalid declaration set is considered different from any other.

- Otherwise, the new \( S(f,C) \) is a lookup set with the shared set of declarations and the union of the subobject sets.

The result of name lookup for \( f \) in \( C \) is the declaration set of \( S(f,C) \). If it is an invalid set, the program is ill-formed.

[Example:

```cpp
draft
struct A { int x; };  // S(x,A) = { { A::x }, [ A ] }
draft
struct B { float x; };  // S(x,B) = { { B::x }, [ B ] }
draft
struct C: public A, public B { };  // S(x,C) = { invalid, [ A in C, B in C ] }
draft
struct D: public virtual C { };  // S(x,D) = S(x,C)
draft
struct E: public virtual C { char x; };  // S(x,E) = { { E::x }, [ E ] }
draft
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]

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:

```cpp
class A {
    public:
        int f();
};

class B {
    public:
};
```
10.2 Member name lookup

```cpp
    int f();
};

class C : public A, public B {
    int f() { return A::f() + B::f(); }
};

— end example ]

[ Note: A static member, a nested type or an enumerator defined in a base class T can unambiguously be found even
if an object has more than one base class subobject of type T. Two base class subobjects share the non-static member
subobjects of their common virtual base classes. — end note ] [ Example:

    class V { public: int v; };  
    class A {
        public:
            int a;
            static int s;
            enum { e };  
    };  
    class B : public A, public virtual V {};  
    class C : public A, public virtual V {};  

    class D : public B, public 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:

    class V { public: int f(); int x; };  
    class W { public: int g(); int y; };  
    class B : public virtual V, public W {
        public:
            int f(); int x;
            int g(); int y;
    };  
    class C : public virtual V, public W {};  

    class D : public B, public C { void glorp(); };

Draft
11 [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]

```cpp
void D::glorp()
{
    x++;                        // OK: B::x hides V::x
    f();                        // OK: B::f() hides V::f()
    y++;                        // error: B::y and C's W::y
    g();                        // error: B::g() and C's W::g()
}
```

—end example]

12 An explicit or implicit conversion from a pointer to or an lvalue of a derived class to a pointer or reference to one of its base classes shall unambiguously refer to a unique object representing the base class. [Example:

```cpp
class V { };  
class A { };  
class B : public A, public virtual V { }; 
class C : public A, public virtual V { };  
class D : public B, public C { }; 

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]

13 [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:

```cpp
struct B1 {
    void f();
    static void f(int);
    int i;
};
struct B2 {

Draft
10.3 Virtual functions

Virtual functions support dynamic binding and object-oriented programming. A class that declares or inherits a virtual function is called a polymorphic class.

If a virtual member function \( vf \) is declared in a class \( Base \) and in a class \( Derived \), derived directly or indirectly from \( Base \), a member function \( vf \) with the same name, parameter-type-list (8.3.5), and cv-qualification as \( Base::vf \) is declared, then \( Derived::vf \) is also virtual (whether or not it is so declared) and it overrides \( Base::vf \). For convenience we say that any virtual function overrides itself. Then in any well-formed class, for each virtual function declared in that class or any of its direct or indirect base classes there is a unique final overrider that overrides that function and every other overrider of that function. The rules for member lookup (10.2) are used to determine the final overrider for a virtual function in the scope of a derived class but ignoring names introduced by using-declarations.

[Example:

```cpp
struct A {
    virtual void f();
};
struct B : virtual A {
    virtual void f();
};

struct C : B, virtual A {
    using A::f;
};
void foo() {
    C c;
    c.f(); // calls B::f, the final overrider
}
```

A function with the same name but a different parameter list (clause over) as a virtual function is not necessarily virtual and does not override. The use of the `virtual` specifier in the declaration of an overriding function is legal but redundant (has empty semantics). Access control (clause class.access) is not considered in determining overriding.

Draft
c.C::f(); // calls k::f because of the using-declaration
}

— end example ]

3 [ Note: a virtual member function does not have to be visible to be overridden, for example,

    struct B {
        virtual void f();
    };
    struct D : B {
        void f(int);
    };
    struct D2 : D {
        void f();
    };

    the function f(int) in class D hides the virtual function f() in its base class B; D::f(int) is not a virtual function. However, f() declared in class D2 has the same name and the same parameter list as B::f(), and therefore is a virtual function that overrides the function B::f() even though B::f() is not visible in class D2. — end note ]

4 Even though destructors are not inherited, a destructor in a derived class overrides a base class destructor declared virtual; see class.dtor and class.free.

5 The return type of an overriding function shall be either identical to the return type of the overridden function or covariant with the classes of the functions. If a function D::f overrides a function B::f, the return types of the functions are covariant if they satisfy the following criteria:

   — both are pointers to classes or references to classes

   — 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

   — 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.

6 If the return type of D::f differs from the return type 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();
    };

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;  // calls Derived::pf() and does not
                        // convert the result to B*
    dp->vf2();      // ill-formed: argument mismatch
}

— end example ]

7  [ 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 ]

8  [ 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 ]

9  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).

10  [ 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* b1p = &d;
    A* ap = b1p;
    D* dp = &d;
    ap->f(); // calls D::B1::f
    dp->f(); // ill-formed: ambiguous
}

In class D above there are two occurrences of class A and hence two occurrences of the virtual member function A::f. The final overrider of B1::A::f is B1::f and the final overrider of B2::A::f is B2::f.

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
};
10.4 Abstract classes

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:

```cpp
class point { /* ... */ };  // abstract class
class shape { // abstract class
    point center;
    // ...

    public:
    point where() { return center; }
    void move(point p) { center=p; draw(); }  // pure virtual
    virtual void rotate(int) = 0;  // pure virtual
    virtual void draw() = 0;  // pure virtual
    // ...
};
```

— end example] [Note: a function declaration cannot provide both a pure-specifier and a definition — end note]

[Example:

```cpp
struct Da : VB1a, VB2 {
};

void foe()
{
    VB1a* vb1ap = new Da;
    vb1ap->f();  // calls VB2::f
}
```

— end example]

Explicit qualification with the scope operator (5.1) suppresses the virtual call mechanism. [Example:

```cpp
class B { public: virtual void f(); };  // abstract class

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.
struct C {
    virtual void f() = 0;  // ill-formed
};

— end example ]

3 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
g(void);  // error
shape& h(shape&);  // OK

— end example ]

4 A class is abstract if it contains or inherits at least one pure virtual function for which the final overrider is pure virtual. [ Example:

class ab_circle : public shape {
    int radius;
public:
    void rotate(int) {}  // ab_circle::draw() is a pure virtual
};

Since shape::draw() is a pure virtual function ab_circle::draw() is a pure virtual by default. The alternative declaration,

class circle : public shape {
    int radius;
public:
    void rotate(int) {}  // a definition is required somewhere
    void draw();  // a definition is required somewhere
};

would make class circle nonabstract and a definition of circle::draw() must be provided. — end example ]

5 [ Note: an abstract class can be derived from a class that is not abstract, and a pure virtual function may override a virtual function which is not pure. — end note ]

6 Member functions can be called from a constructor (or destructor) of an abstract class; the effect of making a virtual call (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.
A member of a class can be

— private; that is, its name can be used only by members and friends of the class in which it is declared.

— 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 11.5).

— public; that is, its name can be used anywhere without access restriction.

A member of a class can also access all the names to which the class has access. A local class of a member function
may access the same names that the member function itself may access.\(^{(102)}\)

Members of a class defined with the keyword class are private by default. Members of a class defined with the
keywords struct or union are public by default. [Example:

```cpp
class X {
    int a; //X::a is private by default
};

struct S {
    int a; //S::a is public by default
};

— end example ]
```

Access control is applied uniformly to all names, whether the names are referred to from declarations or expressions.
[Note: access control applies to names nominated by friend declarations (11.4) 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 { }; 
    public: 
        typedef B BB; 
};
```

\(^{(102)}\) Access permissions are thus transitive and cumulative to nested and local classes.
void f()
{
    A::BB x; // OK, typedef name A::BB is public
    A::B y; // access error, A::B is private
}

— end note ]

5 It should be noted that it is access to members and base classes that is controlled, not their visibility. Names of members are still visible, and implicit conversions to base classes are still considered, when those members and base classes are inaccessible. The interpretation of a given construct is established without regard to access control. If the interpretation established makes use of inaccessible member names or base classes, the construct is ill-formed.

6 All access controls in clause 11 affect the ability to access a class member name from a particular scope. For purposes of access control, the base-specifier s of a class and the definitions of class members that appear outside of the class definition are considered to be within the scope of that class. In particular, access controls apply as usual to member names accessed as part of a function return type, even though it is not possible to determine the access privileges of that use without first parsing the rest of the function declarator. Similarly, access control for implicit calls to the constructors, the conversion functions, or the destructor called to create and destroy a static data member is performed as if these calls appeared in the scope of the member’s class. [Example:

class A {
    typedef int I; // private member
    I f();
    friend I g(I);
    static I x;
    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;

    struct D: A::B, A {};

7 Here, all the uses of A::I are well-formed because A::f and A::x are members of class A and g is a friend of class A. This implies, for example, that access checking on the first use of A::I must be deferred until it is determined that this use of A::I is as the return type of a member of class A. Similarly, the use of A::B as a base-specifier is well-formed because D is derived from A, so checking of base-specifier s must be deferred until the entire base-specifier-list has been seen. — end example ]

8 The names in a default argument expression (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 expression. Access checking for default arguments in function templates and in member functions of class templates is performed as described in 14.7.1.

9 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:

Draft
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):

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 ]
```

2 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 ]
```

3 The order of allocation of data members with separate access-specifier labels is unspecified (9.2). [Note: the effect of access control on the order of allocation of data members is described in 9.2. — end note ]

4 When a member is redeclared within its class definition, the access specified at its redeclaration shall be the same as at its initial declaration. [Example:
11.2 Accessibility of base classes and base class members

```
struct S {
    class A;
private:
    class A { };  // error: cannot change access
};
```

— end example ]

[Note: In a derived class, the lookup of a base class name will find the injected-class-name instead of the name of the base class in the scope in which it was declared. The injected-class-name might be less accessible than the name of the base class in the scope in which it was declared. — end note ]

[Example:
```
class A { };
class B : private A { };
class C : public B {
    A *p;  // error: injected-class-name A is inaccessible
    ::A *q;  // OK
};
```
— end example ]

### 11.2 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:
```
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 ]

(103) 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.

Draft
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() {
    mi = 3;       // error: mi is private in D
    si = 3;       // error: si is private in D
    ::B b;
    b.mi = 3;     // OK (b.mi is different from this->mi)
    b.si = 3;     // OK (b.si is different from this->si)
    ::B::si = 3;  // OK
    ::B* bp1 = this;     // error: B is a private base class
    ::B* bp2 = (::B*)this; // OK with cast
    bp2->mi = 3;       // OK: access through a pointer to B.
    }
}
```

— end note ]

A base class B of N is accessible at R, if

— an invented public member of B would be a public member of N, or

— 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

— 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

— 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 named by the nested-name-specifier of the qualified-id (that is, T). — end note ] A member m is accessible at the point R when named in class N if

— m as a member of N is public, or
— m as a member of N is private, and R occurs in a member or friend of class N, or
— m as a member of N is protected, and R occurs in a member or friend of class N, or in a member or friend of a class P derived from N, where m as a member of P is public, private, or protected, or
— there exists a base class B of N that is accessible at R, and m is accessible at R when named in class B. [ Example:

    class B;
    class A {
        private:
        int i;
        friend void f(B*);
    };
    class B : public A {
    void f(B* p) {
        p->i = 1;  // OK: B* can be implicitly cast to A*,
        // and f has access to i in A
    }

    — end example ]

If a class member access operator, including an implicit “this->;” is used to access a non-static data member or non-static member function, the reference is ill-formed if the left operand (considered as a pointer in the “.” operator case) cannot be implicitly converted to a pointer to the naming class of the right operand. [ Note: this requirement is in addition to the requirement that the member be accessible as named. — end note ]
11.3 Access declarations

The access of a member of a base class can be changed in the derived class by mentioning its qualified-id in the derived class definition. Such mention is called an access declaration. The effect of an access declaration qualified-id ; is defined to be equivalent to the declaration using qualified-id ;.\[104\]

[Example:

```cpp
class A {
public:
  int z;
  int z1;
};

class B : public A {
  int a;
public:
  int b, c;
  int bf();
protected:
  int x;
  int y;
};

class D : private B {
  int d;
public:
  B::c;           // adjust access to B::c
  B::z;           // adjust access to A::z
  A::z1;          // adjust access to A::z1
  int e;
  int df();
protected:
  B::x;           // adjust access to B::x
  int g;
};

class X : public D {
  int xf();
};

int ef(D&);
int ff(X&);
```

The external function ef can use only the names c, z, z1, e, and df. Being a member of D, the function df can use the names b, c, z, z1, bf, x, y, d, e, df, and g, but not a. Being a member of B, the function bf can use the members a, b, c, z, z1, bf, x, and y. The function xf can use the public and protected names from D, that is, c, z, z1, e, and

\[104\] Access declarations are deprecated; member using-declarations (7.3.3) provide a better means of doing the same things. In earlier versions of the C++ language, access declarations were more limited; they were generalized and made equivalent to using-declarations in the interest of simplicity. Programmers are encouraged to use using-declarations, rather than the new capabilities of access declarations, in new code.

Draft
11.4 Friends

A friend of a class is a function or class that is given permission to use the private and protected member names from the class. A class specifies its friends, if any, by way of friend declarations. Such declarations give special access rights to the friends, but they do not make the nominated friends members of the befriending class. [Example: the following example illustrates the differences between members and friends:

```c++
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-specifier`s and member declarations of the befriended class. [Example:}

```c++
class A {
    class B { }; // OK: A::B accessible to friend
    friend class X;
};

struct X : A::B {
    A::B mx; // OK: A::B accessible to member of friend
    class Y {
        A::B my; // OK: A::B accessible to nested member of friend
    }
};
```

— end example ] A class shall not be defined in a friend declaration. [Example:}

```c++
class X {
    enum { a=100 }; // OK: A::B accessible to friend
    friend class Y;
};
```
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 14.5.4). — 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:

```cpp
class C;
typedef C Ct;

class X1 {
    friend C;  // OK: class C is a friend
};

class X2 {
    friend Ct;  // OK: class C is a friend
    friend D;  // error: no type-name D in scope
    friend class D;  // OK: elaborated-type-specifier declares new class
};

template <typename T> class R {
    friend T;
};

R<C> rc;  // class C is a friend of R<C>
R<int> Ri;  // OK: "friend int;" is ignored
```

— end example ]

A function first declared in a friend declaration has external linkage (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:

```cpp
class Y {
    friend char* X::foo(int);
    friend X::X(char);  // constructors can be friends
```
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:

```cpp
class M {
    friend void f() { } // definition of global f, a friend of M.
    // not the definition of a member function
};
```

— end example ]

Such a function is implicitly 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:

```cpp
class A {
    friend class B;
    int a;
};

class B {
    friend class C;
};

class C {
    void f(A* p)
    {
        p->a++; // error: C is not a friend of A
        // despite being a friend of a friend
    }
};

class D : public B {
    void f(A* p)
    {
        p->a++; // error: D is not a friend of A
        // despite being derived from a friend
    }
};
```

Draft
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 nonclass 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 ]

11.5 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)\(^{105}\). 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 name 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*);

\(^{105}\) This additional check does not apply to other members, e.g. static data members or enumerator member constants.
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:

```cpp
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 }

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.

11.7 Multiple access [class.paths]

1 If a name can be reached by several paths through a multiple inheritance graph, the access is that of the path that gives
most access. [Example:

    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 ]

11.8 Nested classes [class.access.nest]

1 A nested class is a member and as such has the same access rights as any other member. The members of an enclosing
class have no special access to members of a nested class; the usual access rules (clause 11) shall be obeyed. [Example:

    class E {
        int x;
        class B { };  

        class I {
            B b;  // OK: E::I can access E::B
            int y;
            void f(E* p, int i)  
            {
                p->x = i;  // OK: E::I can access E::x
            };

            int g(I* p)  
            {

    Draft
11.8 Nested classes

return p->y;  // error: I::y is private
};

— end example ]
Chapter 12  Special member functions

1 The default constructor (12.1), copy constructor and copy assignment operator (12.8), and destructor (12.4) are special member functions. The implementation will implicitly declare these member functions for a class type when the program does not explicitly declare them, except as noted in 12.1. The implementation will implicitly define them if they are used, as specified in 12.1, 12.4 and 12.8. Programs shall not define implicitly-declared special member functions. Programs may explicitly refer to implicitly declared special member functions. [Example: a program may explicitly call, take the address of or form a pointer to member to an implicitly declared special member function.

```cpp
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 ]

2 [Note: the special member functions affect the way objects of class type are created, copied, and destroyed, and how values can be converted to values of other types. Often such special member functions are called implicitly. — end note ]

3 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 ]

12.1 Constructors

Constructors do not have names. A special declarator syntax using an optional sequence of function-specifier s (7.1.2) followed by the constructor's class name followed by a parameter list is used to declare or define the constructor. In such a declaration, optional parentheses around the constructor class name are ignored. [Example:

```cpp
class C {
public:
    C(); // declares the constructor
};
C::C() { } // defines the constructor
```

— end example ]
A constructor is used to initialize objects of its class type. Because constructors do not have names, they are never found during name lookup; however an explicit type conversion using the functional notation (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 ]

A typedef-name shall not be used as the class-name in the declarator-id for a constructor declaration.

A constructor shall not be virtual (10.3) or static (9.4). A constructor can be invoked for a const, volatile or const volatile object. A constructor shall not be declared const, volatile, or const volatile (9.3.2). 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.

A default constructor for a class X is a constructor of class X that can be called without an argument. If there is no user-declared constructor for class X, a default constructor is implicitly declared. An implicitly-declared default constructor is an inline public member of its class. A default constructor is trivial if it is implicitly-declared not user-provided (8.4) and if:

— its class has no virtual functions (10.3) and no virtual base classes (10.1), and
— all the direct base classes of its class have trivial default constructors, and
— for all the non-static data members of its class that are of class type (or array thereof), each such class has a trivial default constructor.

Otherwise, the default constructor is non-trivial.

An implicitly-declared non-user-provided default constructor for a class is implicitly defined when it is used (3.2) to create an object of its class type (1.8). The implicitly-defined or explicitly-defaulted default constructor performs the set of initializations of the class that would be performed by a user-written default constructor for that class with an empty mem-initializer-list no ctor-initializer (12.6.2) and an empty function body compound-statement. If that user-written default constructor would be ill-formed, the program is ill-formed. If that user-written default constructor would satisfy the requirements of a constexpr constructor (7.1.5), the implicitly-defined default constructor is constexpr. Before the implicitly-declared non-user-provided default constructor for a class is implicitly defined, all the implicitly-declared non-user-provided default constructors for its base classes and its non-static data members shall have been implicitly defined. [Note: an implicitly-declared default constructor has an exception-specification (15.4). An explicitly-defaulted definition has no implicit exception-specification. — end note ]

Default constructors are called implicitly to create class objects of static or automatic storage duration (3.7.1, 3.7.2) defined without an initializer (8.5), are called to create class objects of dynamic storage duration (3.7.3) created by a new-expression in which the new-initializer 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).

[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 these constructors. — end note ]

A copy constructor (12.8) is used to copy objects of class type.

A union member shall not be of a class type (or array thereof) that has a non-trivial constructor.

No return type (not even void) shall be specified for a constructor. A return statement in the body of a constructor shall not specify a return value. The address of a constructor shall not be taken.
A functional notation type conversion (5.2.3) can be used to create new objects of its type. [Note: The syntax looks like an explicit call of the constructor. — end note] [Example:

```cpp
complex zz = complex(1,2.3);
cprint( complex(7.8,1.2) );
```

— end example ]

An object created in this way is unnamed. [Note: 12.2 describes the lifetime of temporary objects. — end note] [Note: explicit constructor calls do not yield lvalues, see 3.10. — end note]

[Note: some language constructs have special semantics when used during construction; see 12.6.2 and 12.7. — end note]

During the construction of a const object, if the value of the object or any of its subobjects is accessed through an lvalue that is not obtained, directly or indirectly, from the constructor’s this pointer, the value of the object or subobject thus obtained is unspecified. [Example:

```cpp
struct C;
void no_opt(C*);

struct C {
  int c;
  C() : c(0) { no_opt(this); }
};

const C cobj;

void no_opt(C* cptr) {
  int i = cobj.c * 100; // value of cobj.c is unspecified
  cptr->c = 1;
  cout << cobj.c * 100 // value of cobj.c is unspecified
    << 'n';
}
```

— end example ]

12.2 Temporary objects [class.temporary]

Temporaries of class type are created in various contexts: binding an rvalue to a reference (8.5.3), returning an rvalue (6.6.3), a conversion that creates an rvalue (4.1, 5.2.9, 5.2.11, 5.4), throwing an exception (15.1), entering a handler (15.3), 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 avoided (12.8), all the semantic restrictions must be respected as if the temporary object had been created. [Example: even if the copy constructor is not called, all the semantic restrictions, such as accessibility (clause 11), shall be satisfied. — end example ]

[Example:

```cpp
class X {
  // ...
  public:
};
```

Draft
12.2 Temporary objects

Special member functions

// ...
X(int);
X(const X&);
"X();
}

X f(X);

void g()
{
    X a(1);
    X b = f(X(2));
    a = f(a);
}

Here, 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. 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 \(b\). 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 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.

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 expression is sequenced before the construction of the next array element, if any.

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 as specified below. A temporary bound to a reference member in a constructor’s ctor-initializer (12.6.2) persists until the constructor exits. A temporary bound to a reference parameter in a function call (5.2.2) persists until the completion of the full expression containing the call. A temporary bound to the returned value in a function return statement (6.6.3) persists until the function exits. 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 or automatic storage duration (3.7.1, 3.7.2); that is, if \(\text{obj}1\) is an object with the same storage duration as the temporary and created before the temporary is created the temporary shall be destroyed before \(\text{obj}1\) is destroyed; if \(\text{obj}2\) is an object with the same storage duration as the temporary and created after the temporary is created the temporary shall be destroyed after \(\text{obj}2\) is destroyed. [Example:

```cpp
class C {

Draft
public:
    C();
    C(int);
    friend C operator+(const C&, const C&);
    ~C();
};

C obj1;
const C& cr = C(16)+C(23);
C obj2;

the expression C(16)+C(23) creates three temporaries. A first temporary T1 to hold the result of the expression C(16), a second temporary T2 to hold the result of the expression C(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, it is guaranteed that T2 is destroyed before T1. The temporaries T1 and T2 are bound to the reference parameters of operator+; these temporaries are destroyed at the end of the full expression containing the call to operator+. The temporary T3 bound to the reference cr is destroyed at the end of cr’s lifetime, that is, at the end of the program. In addition, the order in which T3 is destroyed takes into account the destruction order of other objects with static storage duration. That is, because obj1 is constructed before T3, and T3 is constructed before obj2, it is guaranteed that obj2 is destroyed before T3, and that T3 is destroyed before obj1. — end example ]

12.3 Conversions [class.conv]

1 Type conversions of class objects can be specified by constructors and by conversion functions. These conversions are called user-defined conversions and are used for implicit type conversions (clause 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:

    class X {
        // ...
        public:
            operator int();
    };

    class Y {
        // ...
        public:
            operator X();
    };

    Y a;
    int b = a;  // error

Draft
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:]

```cpp
class X {
public:
    // ...
    operator int();
};

class Y : public X {
public:
    // ...
    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 [class.convctor]

A constructor declared without the function-specifier explicit that can be called with a single parameter specifies a conversion from the type of its first parameter to the type of its class. Such a constructor is called a converting constructor. [Example:]

```cpp
class X {
    // ...
    public:
    X(int);
    X(const char*, int =0);
};

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))
}
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:

```cpp
class Z {
    public:
        explicit Z();
        explicit Z(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
```

— end example ]

A non-explicit copy-constructor (12.8) is a converting constructor. An implicitly-declared copy 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 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 member functions are called conversion functions. Classes, enumerations, and `typedef-name`s shall not be declared in the `type-specifier-seq`. Neither parameter types nor return type can be specified. The type of a 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.¹⁰⁶

[Example:

```cpp
class X {
    // ...
    public:
```

¹⁰⁶ 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.
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 ]

2 User-defined conversions are not restricted to use in assignments and initializations. [ Example:

    void g(X a, X b)
    {
        int i = (a) ? 1+a : 0;
        int j = (a&&b) ? a+b : i;
        if (a) // ...
    }

    — end example ]

3 The conversion-type-id shall not represent a function type nor an array type. The conversion-type-id in a conversion-function-id is the longest possible sequence of conversion-declarator s. [ Note: this prevents ambiguities between the declarator operator * and its expression counterparts. [ Example:

    &ac.operator int*i;  // syntax error;
    // parsed as: &(ac.operator int *)i
    // not as: &(ac.operator int)*i

    The * is the pointer declarator and not the multiplication operator. — end example ] — end note ]

Conversion functions are inherited.

5 Conversion functions can be virtual.

6 Conversion functions cannot be declared static.

12.4 Destructors [class.dtor]

1 A special declarator syntax using an optional function-specifier (7.1.2) followed by ~ followed by the destructor’s class name followed by an empty parameter list is used to declare the destructor in a class definition. In such a declaration, the ~ followed by the destructor’s class name can be enclosed in optional parentheses; such parentheses are ignored. A typedef-name shall not be used as the class-name following the ~ in the declarator for a destructor declaration.

2 A destructor is used to destroy objects of its class type. A destructor takes no parameters, and no return type can be specified for it (not even void). The address of a destructor shall not be taken. A destructor shall not be static. A destructor can be invoked for a const, volatile or const volatile object. A destructor shall not be declared const, volatile or const volatile (9.3.2). 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.

Draft
If a class has no user-declared destructor, a destructor is declared implicitly. An implicitly-declared destructor is an inline public member of its class. A destructor is trivial if it is implicitly declared not user-provided and if:

- all of the direct base classes of its class have trivial destructors and
- for all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor.

Otherwise, the destructor is non-trivial.

An implicitly-declared non-user-provided destructor is implicitly defined when it is used to destroy an object of its class type (3.7). A program is ill-formed if the class for which a destructor is implicitly defined or explicitly defaulted has:

- a non-static data member of class type (or array thereof) with an inaccessible destructor, or
- a base class with an inaccessible destructor.

Before the implicitly-declared non-user-provided destructor for a class is implicitly defined, all the implicitly-declared non-user-defined destructors for its base classes and its non-static data members shall have been implicitly defined. [Note: an implicitly-declared destructor has an exception-specification (15.4). An explicitly defaulted definition has no implicit exception-specification. — end note]

After executing the body of the destructor and destroying any automatic objects allocated within the body, a destructor for class X calls the destructors for X’s direct members, the destructors for X’s direct base classes and, if X is the type of the most derived class (12.6.2), its destructor calls the destructors for X’s virtual base classes. All destructors are called as if they were referenced with a qualified name, that is, ignoring any possible virtual overriding destructors in more derived classes. Bases and members are destroyed in the reverse order of the completion of their constructor (see 12.6.2). A 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).

A destructor can be declared 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.

[Note: some language constructs have special semantics when used during destruction; see 12.7. — end note]

A union member shall not be of a class type (or array thereof) that has a non-trivial destructor.

Destructors are invoked implicitly (1) for a constructed object with static storage duration (3.7.1) at program termination (3.6.3), (2) for a constructed object with automatic storage duration (3.7.2) when the block in which the object is created exits (6.7), (3) for a constructed temporary object when the lifetime of the temporary object ends (12.2), (4) for a constructed object allocated by a new-expression (5.3.4), through use of a delete-expression (5.3.5), (5) in several situations due to the handling of exceptions (15.3). A program is ill-formed if an object of class type or array thereof is declared and the destructor for the class is not accessible at the point of the declaration. Destructors can also be invoked explicitly.

At the point of definition of a virtual destructor (including an implicit definition (12.8)), the non-array deallocation function is looked up in the scope of the destructor’s class (10.2), and, if no declaration is found, the function is looked up in the global scope. If the result of this lookup is ambiguous or inaccessible, or if the lookup selects a placement
deallocation function, the program is ill-formed. [Note: this assures that a deallocation function corresponding to the dynamic type of an object is available for the `delete-expression` (12.5). — end note]

In an explicit destructor call, the destructor name appears as a `~` followed by a `type-name` that names 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, the program has undefined behavior (except that invoking `delete` on a null pointer has no effect). [Example:

```c
struct B {
  virtual ~B() { }
};
struct D : B {
  ~D() { }
};
D D_object;
typedef B B_alias;
B* B_ptr = &D_object;

void f() {
  D_object.B::~B();    // calls B’s destructor
  B_ptr->~B();        // calls D’s destructor
  B_ptr->B_alias::~B(); // calls B’s destructor
  B_ptr->B_alias::~B_alias(); // calls B’s destructor
}
```

— end example] [Note: an explicit destructor call must always be written using a member access operator (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 `new-expression` with the placement option. 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();  // cleanup
}
```
14 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]

15 [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,

```cpp
typedef int I;
I* p;
// ...
p->I::~I();
```

—end note]

12.5 Free store [class.free]

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 (3.7.3.2).

4 If a delete-expression begins with a unary :: operator, the deallocation function’s name is looked up in global scope. Otherwise, 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). Otherwise, if the delete-expression is used to deallocate an object of class T or array thereof, the static and dynamic

107) 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 delete-expression’s operand be the same as its dynamic type.
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, the name is looked up in the global scope. If the result of the lookup is ambiguous or inaccessible, or if the lookup selects a placement deallocation function, the program is ill-formed.

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 style is used) the size of the block as its second argument.\(^{(108)}\)

Any deallocation function for a class X is a static member (even if not explicitly declared static). [Example:

```cpp
class X {
    // ...
    void operator delete(void*);
    void operator delete[](void*, std::size_t);
};

class Y {
    // ...
    void operator delete(void*, std::size_t);
    void operator delete[](void*);
};

— end example]
```

Since member allocation and deallocation functions are static they cannot be virtual. [Note: however, when the cast-expression of a delete-expression refers to an object of class type, because the deallocation function actually called is looked up in the scope of the class that is the dynamic type of the object, if the destructor is virtual, the effect is the same. For example,

```cpp
struct B {
    virtual ~B();
    void operator delete(void*, std::size_t);
};

struct D : B {
    void operator delete(void*);
};

void f()
{
    B* bp = new D;
    delete bp;           // 1: uses D::operator delete(void*)
}
```

Here, storage for the non-array object of class D is deallocated by D::operator delete(), due to the virtual destructor. — end note] [Note: virtual destructors have no effect on the deallocation function actually called when the cast-expression of a delete-expression refers to an array of objects of class type. For example,

\(^{(108)}\) If the static type in the delete-expression is different from the dynamic type and the destructor is not virtual the size might be incorrect, but that case is already undefined; see 5.3.5.
struct B {
    virtual ~B();
    void operator delete[](void*, std::size_t);
};

struct D : B {
    void operator delete[](void*, std::size_t);
};

void f(int i) {
    D* dp = new D[i];
    delete [] dp; // uses D::operator delete[](void*, std::size_t)
    B* bp = new D[i];
    delete[] bp; // undefined behavior
}

--- end note ---

Access to the deallocation function is checked statically. Hence, even though a different one might actually be executed, the statically visible deallocation function is required to be accessible. [Example: for the call on line //1 above, if B::operator delete() had been private, the delete expression would have been ill-formed. —end example]

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), 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 [class.expl.init]

1. An object of class type can be initialized with a parenthesized expression-list, where the expression-list is construed as an argument list for a constructor that is called to initialize the object. Alternatively, a single assignment-expression can be specified as an initializer using the = form of initialization. Either direct-initialization semantics or copy-initialization semantics apply; see 8.5. [Example:

```cpp
class complex {
    // ...
public:
    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 it into c
complex d = sqrt(b,c); // call sqrt(complex,complex)
    // and copy the result into d
complex e;     // initialize by a call of
    // complex()
complex f = 3;   // construct complex(3) using
    // complex(double)
    // copy it into f
complex g = { 1, 2 }; // error: constructor is required

— end example ] [ Note: overloading of the assignment operator (13.5.3) has no effect on initialization. — end note ]

2 When an aggregate (whether class or array) contains members of class type and is initialized by a brace-enclosed
    initializer-list (8.5.1), each such member is copy-initialized (see 8.5) by the corresponding assignment-expression. If
there are fewer initializer s in the initializer-list than members of the aggregate, each member not explicitly initialized
shall be value-initialized (8.5). [ Note: 8.5.1 describes how assignment-expression s in an initializer-list are paired with
the aggregate members they initialize. — end note ] [ Example:

complex v[6] = { 1, complex(1,2), complex(), 2 };

Here, complex::complex(double) is called for the initialization of v[0] and v[3]. complex::complex(double,
    double) is called for the initialization of v[1]. complex::complex() is called for the initialization v[2], v[4], and
    v[5]. For another example,

class X {
public:
    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 ini-
    tialization 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 ]

3 [ Note: if T is a class type with no default constructor, any declaration of an object of type T (or array thereof) is ill-formed
if no initializer is explicitly specified (see 12.6 and 8.5). — end note ]

4 [ Note: the order in which objects with static storage duration are initialized is described in 3.6.2 and 6.7. — end note ]

12.6.2 Initializing bases and members [class.base.init]

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

ctor-initializer:
    : mem-initializer-list

Draft
mem-initializer-list:
   mem-initializer ...opt
mem-initializer, mem-initializer-list ...opt

mem-initializer:
   mem-initializer-id ( expression-listopt )

mem-initializer-id:
   ::opt nested-name-specifieropt class-name
   identifier

2 Names in a mem-initializer-id are looked up in the scope of the constructor’s class and, if not found in that scope, are 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 name 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]

A mem-initializer-list can delegate to another constructor of the constructor’s class using any name 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 {
   C( int ) { }  // I: 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] 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 the member of an anonymous union that is a member of the constructor’s class. If a ctor-initializer specifies more than one mem-initializer for the same member, for the same base class or for multiple members of the same union (including members of anonymous unions), the ctor-initializer is ill-formed.
The expression-list in a mem-initializer is used to initialize the base class or non-static data member subobject denoted by the mem-initializer-id. The semantics of a mem-initializer are as follows:

- if the expression-list of the mem-initializer is omitted, the base class or member subobject is value-initialized (see 8.5);
- otherwise, the subobject indicated by mem-initializer-id is direct-initialized using expression-list as the initializer (see 8.5).

[Example:

```cpp
struct B1 { B1(int); /* ... */;};
struct B2 { B2(int); /* ... */;};
struct D : B1, B2 {
    D(int);
    B1 b;
    const int c;
};

D::D(int a) : B2(a+1), B1(a+2), c(a+3), b(a+4)
{ /* ... */
}
D d(10);
```

— end example] The initialization of each base and member constitutes a full-expression. Any expression in a mem-initializer is evaluated as part of the full-expression that performs the initialization.

If a given non-static data member or base class is not named by a mem-initializer-id (including the case where there is no mem-initializer-list because the constructor has no ctor-initializer), then

- If the entity is a non-static data member of (possibly cv-qualified) class type (or array thereof) or a base class, and the entity class is a non-POD trivial class, the entity is default-initialized (8.5). If the entity is a non-static data member of a const-qualified type, the entity class shall have a user-declared provided default constructor.
- Otherwise, the entity is not initialized. If the entity is of const-qualified type or reference type, or of a (possibly cv-qualified) POD trivial class type (or array thereof) containing (directly or indirectly) a member of a const-qualified type, the program is ill-formed.

After the call to a constructor for class X has completed, if a member of X is neither specified in the constructor’s mem-initializers, nor default-initialized, nor value-initialized, nor given a value during execution of the compound-statement of the body of the constructor, the member has indeterminate value.

Initialization shall proceed in the following order:

- First, and only for the constructor of the most derived class as described below, virtual base classes shall be 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 class names in the derived class base-specifier-list.
- Then, direct base classes shall be initialized in declaration order as they appear in the base-specifier-list (regardless of the order of the mem-initializers).
- Then, non-static data members shall be initialized in the order they were declared in the class definition (again regardless of the order of the mem-initializers).
— Finally, the `body 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]

6 All subobjects representing virtual base classes are initialized by the constructor of the most derived class (1.8). If the constructor of the most derived class does not specify a `mem-initializer` for a virtual base class V, then V’s default constructor is called to initialize the virtual base class subobject. If V does not have an accessible default constructor, the initialization is ill-formed. A `mem-initializer` naming a virtual base class shall be ignored during execution of the constructor of any class that is not the most derived class. [Example:

```cpp
class V {
  public:
    V();
    V(int);
    // ...
};
class A : public virtual V {
  public:
    A();
    A(int);
    // ...
};
class B : public virtual V {
  public:
    B();
    B(int);
    // ...
};
class C : public A, public B, private virtual V {
  public:
    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]

7 Names in the `expression-list` of a `mem-initializer` are evaluated in the scope of the constructor for which the `mem-`
**12.6 Initialization**

Special member functions

Initializer is specified. [Example:

```cpp
class X {
    int a;
    int b;
    int i;
    int j;
public:
    const int& r;
    X(int i): r(a), b(i), i(i), j(this->i) {} }
```

initializes `X::r` to refer to `X::a`, initializes `X::b` with the value of the constructor parameter `i`, initializes `X::i` with the value of the constructor parameter `i`, and initializes `X::j` with the value of `X::i`; this takes place each time an object of class `X` is created. — end example] [Note: because the mem-initializer are evaluated in the scope of the constructor, the this pointer can be used in the expression-list of a mem-initializer to refer to the object being initialized. — end note]

Member functions (including virtual member functions, 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-initializer s 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();
    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]
[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]

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:

```c++
template<class... Mixins>
class X : public Mixins...
{
public:
X(const Mixins&... mixins) : Mixins(mixins)... { }
};
```

— end example]

### 12.7 Construction and destruction

For an object of non-POD-trivial class type (clause 9), before the constructor begins execution and after the destructor finishes execution, referring to any non-static member or base class of the object results in undefined behavior. [Example:

```c++
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 POD-trivial class
X xobj;
```

For another example,

```c++
struct W { int j; }; // POD-trivial
struct X : public virtual W { }; // pod-trivial
struct Y {
    int *p;
    X x;
    Y() : p(&x.j) // undefined, x is not yet constructed
    { }
};
```

— end example]
3 To explicitly or implicitly convert a pointer (an lvalue) referring to an object of class \(X\) to a pointer (reference) to a direct or indirect base class \(B\) of \(X\), the construction of \(X\) and the construction of all of its direct or indirect bases that directly or indirectly derive from \(B\) shall have started and the destruction of these classes shall not have completed, otherwise the conversion results in undefined behavior. To form a pointer to (or access the value of) a direct non-static member of an object \(obj\), the construction of \(obj\) shall have started and its destruction shall not have completed, otherwise the computation of the pointer value (or accessing the member value) results in undefined behavior. [Example:

```cpp
struct A { }
struct B : virtual A { }
struct C : B { }
struct D : virtual A { D(A*); }
struct X { X(A*); }

struct E : C, D, X {
    E() : D(this), // undefined: upcast from E* to A*
        // might use path E* -> D* -> A*
        // but D is not constructed
        // D((C*)this), // defined:
        // E* -> C* defined because E() has started
        // and C* -> A* defined because
        // C fully constructed
    X(this) // defined: upon construction of X,
        // C/B/D/A sublattice is fully constructed
    { }
};
```

—end example]

4 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 (including from the mem-initializer for a data member) or from a destructor, and the object to which the call applies is the object under construction or destruction, the function called is the one defined in the constructor or destructor’s own class or in one of its bases, but not a function overriding it in a class derived from the constructor or destructor’s class, or overriding it in one of the other base classes of the most derived object (1.8). If the virtual function call uses an explicit class member access (5.2.5) and the object-expression refers to the object under construction or destruction but its type is neither the constructor or destructor’s own class or one of its bases, the result of the call is undefined. [Example:

```cpp
class V {
public:
    virtual void f();
    virtual void g();
};

class A : public virtual V {
public:
    virtual void f();
};

class B : public virtual V {
public:
```

Draft
virtual void g();
B(V*, A*);
}

class D : public A, B {
public:
    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 ]

5 The typeid operator (5.2.8) can be used during construction or destruction (12.6.2). When typeid is used in a constructor (including from the mem-initializer for a data member) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the operand of typeid refers to the object under construction or destruction, typeid yields the std::type_info object representing the constructor or destructor’s class. If the operand of typeid refers to the object under construction or destruction and the static type of the operand is neither the constructor or destructor’s class nor one of its bases, the result of typeid is undefined.

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 from the mem-initializer for a 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:

    class V {
    public:
        virtual void f();
    };

class A : public virtual V { };

class B : public virtual V {
public:
    B(V*, A*);
};

class D : public A, B {

Draft
public:
    D() : B((A*)this, this) { }
};

B::B(V* v, A* a) {
    typeid(*this);
    // type_info for B
    typeid(*v);
    // well-defined: *v has type V, a base of B
    // yields type_info for B
    typeid(*a);
    // undefined behavior: type A not a base of B
dynamic_cast<B*>(v);
    // well-defined: v of type V*, V base of B
    // results in B*
dynamic_cast<B*>(a);
    // undefined behavior;
    // a has type A*, A not a base of B
}

— end example ]

12.8 Copying class objects

A class object can be copied 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 (5.17). Conceptually, these two operations are implemented by a copy constructor (12.1) and copy 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).\[109\] [Example: X::X(const X&) and X::X(X&, int=1) are copy constructors.

    class X {
    //...
    public:
    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 ] [Note: all forms of copy constructor may be declared for a class. [Example:]

    class X {
    //...
    public:
    X(const X&);
    X(X&);  // OK
    };

— end example ] — end note ] [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:

\[109\] Because a template constructor or a constructor whose first parameter is an rvalue reference is never a copy constructor, the presence of such a constructor does not suppress the implicit declaration of a copy constructor. Such constructors participate in overload resolution with other constructors, including copy constructors, and, if selected, will be used to copy an object.

Draft
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 ]

3 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 perform the copy of a class object to an object of its class type. [Example:

```cpp
struct S {
    template<typename T> S(T);
};
S f();
void g() {
    S a( f() ); // does not instantiate member template
}
```

— end example ]

4 If the class definition does not explicitly declare a copy constructor, one is declared implicitly. Thus, for the class definition

```cpp
struct X {
    X(const X&, int);
};
```
a copy constructor is implicitly-declared. If the user-declared constructor is later defined as

```cpp
X::X(const X& x, int i =0) { /* ... */ }
```

then any use of X's copy constructor is ill-formed because of the ambiguity; no diagnostic is required.

5 The implicitly-declared copy constructor for a class X will have the form

```cpp
X::X(const X&)
```

if

   — each direct or virtual base class B of X has a copy constructor whose first parameter is of type const B& or const volatile B&, and

   — for all the non-static data members of X that are of a class type M (or array thereof), each such class type has a copy constructor whose first parameter is of type const M& or const volatile M&.  \[110\\]

Otherwise, the implicitly declared copy constructor will have the form

\[110\\] This implies that the reference parameter of the implicitly-declared copy constructor cannot bind to a volatile lvalue; see C.1.8.

Draft
12.8 Copying class objects

\[ X \cdot X(\&x) \]

An implicitly-declared copy constructor is an inline public member of its class.

6 A copy constructor for class \( X \) is trivial if it is implicitly declared not user-provided (8.4) and if
   - class \( X \) has no virtual functions (10.3) and no virtual base classes (10.1), and
   - each direct base class of \( X \) has a trivial copy constructor, and
   - for all the non-static data members of \( X \) that are of class type (or array thereof), each such class type has a trivial copy constructor;

otherwise the copy constructor is non-trivial.

7 An implicitly-declared non-user-provided copy constructor is implicitly defined if it is used to initialize an object of its class type from a copy of an object of its class type or of a class type derived from its class type\(^{111}\). [Note: the copy constructor is implicitly defined even if the implementation elided its use (12.2). — end note] A program is ill-formed if the class for which a copy constructor is implicitly defined or explicitly defaulted has:
   - a non-static data member of class type (or array thereof) with an inaccessible or ambiguous copy constructor, or
   - a base class with an inaccessible or ambiguous copy constructor.

Before the implicitly-declared non-user-provided copy constructor for a class is implicitly defined, all implicitly-declared non-user-provided copy constructors for its direct and virtual base classes and its non-static data members shall have been implicitly defined. [Note: an implicitly-declared copy constructor has an exception-specification (15.4). An explicitly-defaulted definition has no implicit exception-specification. — end note]

8 The implicitly-defined or explicitly-defaulted copy constructor for class \( X \) performs a memberwise copy of its subobjects. The order of copying is the same as the order of initialization of bases and members in a user-defined constructor (see 12.6.2). Each subobject is copied in the manner appropriate to its type:
   - if the subobject is of class type, the copy constructor for the class is used;
   - if the subobject is an array, each element is copied, in the manner appropriate to the element type;
   - if the subobject is of scalar type, the built-in assignment operator is used.

Virtual base class subobjects shall be copied only once by the implicitly-defined copy constructor (see 12.6.2).

9 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 \( X, \&x, \text{const} \ &x, \text{volatile} \ &x \) or \( \text{const volatile} \ &x \).\(^{112}\) [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 \( \text{const} \ X \) cannot be assigned to an object of type \( X \). [Example:
\[
\text{struct } X \{
\text{X() \};}
\]
\]

\(^{111}\) See 8.5 for more details on direct and copy initialization.

\(^{112}\) 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& operator=(X&);
};
const X cx;
X x;
void f() {
    x = cx;  // error:
    // X::operator=(X&) cannot assign cx into x
}
— end example ] — end note ]

If the class definition does not explicitly declare a copy assignment operator, one is declared implicitly. The implicitly-declared copy assignment operator for a class X will have the form

X& X::operator=(const X&)
if

— 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
— 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. 113)

Otherwise, the implicitly declared copy assignment operator will have the form

X& X::operator=(X&)

The implicitly-declared copy 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 assignment operator is an inline public member of its class. Because a copy assignment operator is implicitly declared for a class if not declared by the user, a base class copy assignment operator is always hidden by the copy 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-assignment operator for the derived class is not considered an explicit declaration of a copy-assignment operator and does not suppress the implicit declaration of the derived class copy-assignment operator; the operator introduced by the using-declaration is hidden by the implicitly-declared copy-assignment operator in the derived class.

A copy assignment operator for class X is trivial if it is implicitly declared not user-provided and if

— class X has no virtual functions (10.3) and no virtual base classes (10.1), and
— each direct base class of X has a trivial copy assignment operator, and
— for all the non-static data members of X that are of class type (or array thereof), each such class type has a trivial copy assignment operator;

otherwise the copy assignment operator is non-trivial.

An implicitly-declared non-user-provided copy assignment operator is implicitly defined when an object of its class type is assigned a value of its class type or a value of a class type derived from its class type. A program is ill-formed if the class for which a copy assignment operator is implicitly defined or explicitly defaulted has:

113) This implies that the reference parameter of the implicitly-declared copy assignment operator cannot bind to a volatile lvalue; see C.1.8.
12.8 Copying class objects

Special member functions

— a non-static data member of `const` type, or
— a non-static data member of reference type, or
— a non-static data member of class type (or array thereof) with an inaccessible copy assignment operator, or
— a base class with an inaccessible copy assignment operator.

Before the implicitly-declared non-user-provided copy assignment operator for a class is implicitly defined, all implicitly-declared non-user-provided copy assignment operators for its direct base classes and its non-static data members shall have been implicitly defined. [Note: an implicitly-declared copy assignment operator has an exception-specification (15.4). An explicitly defaulted definition has no implicit exception-specification. — end note]

The implicitly-defined or explicitly-defaulted copy assignment operator for class X performs memberwise 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. Each subobject is assigned in the manner appropriate to its type:

— if the subobject is of class type, the copy assignment operator for the class is used (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 or explicitly-defaulted copy assignment operator. [Example:

```c
struct V { }
struct A : virtual V { }
struct B : virtual V { }
struct C : B, A { }
```

It is unspecified whether the virtual base class subobject V is assigned twice by the implicitly-defined copy assignment operator for C. — end example]

A program is ill-formed if the copy constructor or the copy assignment operator for an object is implicitly used and the special member function is not accessible (clause 11). [Note: Copying one object into another using the copy constructor or the copy assignment operator does not change the layout or size of either object. — end note]

When certain criteria are met, an implementation is allowed to omit the copy construction of a class object, even if the copy constructor and/or destructor for the object have side effects. In such cases, the implementation treats the source and target of the omitted copy 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 operations is permitted in the following circumstances (which may be combined to eliminate multiple copies):

— in a return statement in a function with a class return type, when the expression is the name of a non-volatile automatic object with the same cv-unqualified type as the function return type, the copy operation can be omitted by constructing the automatic object directly into the function’s return value

---

114) Because only one object is destroyed instead of two, and one copy constructor is not executed, there is still one object destroyed for each one constructed.
— in a throw-expression, when the operand is the name of a non-volatile automatic object, the copy operation from
the operand to the exception object (15.1) can be omitted by constructing the automatic object directly into the
exception object

— when a temporary class object that has not been bound to a reference (12.2) would be copied to a class object with
the same cv-unqualified type, the copy operation can be omitted by constructing the temporary object directly into
the target of the omitted copy

— 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.

[Example:

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. — end example ]

16 When the criteria for elision of a copy operation are met and the object to be copied is designated by an lvalue, overload
resolution to select the constructor for the copy is first performed as if the object were designated by an rvalue. If
overload resolution fails, 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:

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 of t2

— end example ]
Chapter 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 declarations can be overloaded; object 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:

```c
double abs(double);
int abs(int);

abs(1);  // call abs(int);
abs(1.0); // call 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-directive s) or overload resolution (e.g., for operator functions). — end note ]

Certain function declarations cannot be overloaded:

— Function declarations that differ only in the return type cannot be overloaded.

— 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:
class X {
    static void f();
    void f(); // ill-formed
    void f() const; // ill-formed
    void f() const volatile; // ill-formed
    void g();
    void g() const; // OK: no static g
    void g() const volatile; // OK: no static g
};

— end example }

[Note: as specified in 8.3.5, function declarations that have equivalent parameter declarations declare the same function and therefore cannot be overloaded:]

— 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:
    typedef int Int;
    void f(int i);
    void f(Int i); // OK: redeclaration of f(int)
    void f(int i) { /* ... */ } // error: redefinition of f(int)

— end example]

Enumerations, on the other hand, are distinct types and can be used to distinguish overloaded function declarations. [Example:
    enum E { a };
    void f(int i) { /* ... */ }
    void f(E i) { /* ... */ }

— end example]

— 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:
    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:

```
void h(int());
void h(int (*)(()));  // redeclaration of h(int())
void h(int x());     // definition of h(int())
void h(int (*)(x)()); // ill-formed: redefinition of h(int())
```
— end example]

— Parameter declarations that differ only in the presence or absence of const and/or volatile are equivalent. That is, the const and volatile type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called. [Example:

```
typedef const int cInt;
int f (int);
int f (const int);      // redeclaration of f(int)
int f (int) { ... }     // definition of f(int)
int f (cInt) { ... }    // error: redefinition of f(int)
```
— end example]

Only the const and volatile type-specifiers at the outermost level of the parameter type specification are ignored in this fashion; const and volatile type-specifiers buried within a parameter type specification are significant and can be used to distinguish overloaded function declarations. In particular, for any type T, “pointer to T,” “pointer to const T,” and “pointer to volatile T” are considered distinct parameter types, as are “reference to T,” “reference to const T,” and “reference to volatile T.”

— Two parameter declarations that differ only in their default arguments are equivalent. [Example: consider the following:

```
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]

---

(115) 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.
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:

```cpp
class B {
public:
    int f(int);
};
class D : public B {
public:
    int f(char*);
};
```

Here D::f(char*) hides B::f(int) rather than overloading it.

```cpp
void h(D* pd)
{
    pd->f(1); // error: D::f(char*) hides B::f(int)
    pd->B::f(1); // OK
    pd->f("Ben"); // OK, calls D::f
}
```

— end example ]

2 A locally declared function is not in the same scope as a function in a containing scope. [Example:

```cpp
int f(char*);
void g()
{
    extern f(int);
    f("asdf"); // error: f(int) hides f(char*)
    // so there is no f(char*) in this scope
}
```

```cpp
void caller ()
{
    extern void callee(int, int);
    {
        extern void callee(int); // hides callee(int, int)
        callee(88, 99); // error: only callee(int) in scope
    }
}
```

— end example ]

3 Different versions of an overloaded member function can be given different access rules. [Example:
Overloading 13.3 Overload resolution

```cpp
class buffer {
    private:
        char* p;
        int size;

    protected:
        buffer(int s, char* store) { size = s; p = store; }
        // ...

    public:
        buffer(int s) { p = new char[size = s]; }
        // ...
};

— end example ]

13.3 Overload resolution  [over.match]

1 Overload resolution is a mechanism for selecting the best function to call given a list of expressions that are to be the arguments of the call and a set of candidate functions that can be called based on the context of the call. The selection criteria for the best function are the number of arguments, how well the arguments match the parameter-type-list of the candidate function, how well (for non-static member functions) the object matches the implied object parameter, and certain other properties of the candidate function. [Note: the function selected by overload resolution is not guaranteed to be appropriate for the context. Other restrictions, such as the accessibility of the function, can make its use in the calling context ill-formed. — end note ]

2 Overload resolution selects the function to call in seven distinct contexts within the language:

- invocation of a function named in the function call syntax (13.3.1.1.1);
- invocation of a function call operator, a pointer-to-function conversion function, a reference-to-pointer-to-function conversion function, or a reference-to-function conversion function on a class object named in the function call syntax (13.3.1.1.2);
- invocation of the operator referenced in an expression (13.3.1.2);
- invocation of a constructor for direct-initialization (8.5) of a class object (13.3.1.3);
- invocation of a user-defined conversion for copy-initialization (8.5) of a class object (13.3.1.4);
- 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
- invocation of a conversion function for conversion to an lvalue 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:

- 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).
13.3 Overload resolution

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.

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

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.

The set of candidate functions can contain both member and non-member functions to be resolved against the same argument list. So that argument and parameter lists are comparable within this heterogeneous set, a member function is considered to have an extra parameter, called the implicit object parameter, which represents the object for which the member function has been called. For the purposes of overload resolution, both static and non-static member functions have an implicit object parameter, but constructors do not.

Similarly, when appropriate, the context can construct an argument list that contains an implied object argument to denote the object to be operated on. Since arguments and parameters are associated by position within their respective lists, the convention is that the implicit object parameter, if present, is always the first parameter and the implied object argument, if present, is always the first argument.

For non-static member functions, the type of the implicit object parameter is “reference to cv X” 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 implicit 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; and
— even if the implicit object parameter is not const-qualified, an rvalue temporary can be bound to the parameter as long as in all other respects the temporary can be converted to the type of the implicit object parameter. [Note: The fact that such a temporary is an rvalue does not affect the ranking of implicit conversion sequences (13.3.3.2). — end note]
Because 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:

```cpp
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. 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.

**13.3.1.1 Function call syntax** [over.match.call]

Recall from 5.2.2, that a function call is a postfix-expression, possibly nested arbitrarily deep in parentheses, followed by an optional expression-list enclosed in parentheses:

```
( ... (opt postfix-expression ) ... )opt ( expression-listopt )
```

Overload resolution is required if the postfix-expression is the name of a function, a function template (14.5.6), an object of class type, or a set of pointers-to-function.

13.3.1.1.1 describes how overload resolution is used in the first two of the above cases to determine the function to call. 13.3.1.1.2 describes how overload resolution is used in the third of the above cases to determine the function to call.

The fourth case arises from a postfix-expression of the form &F, where F names a set of overloaded functions. In the context of a function call, &F is treated the same as the name F by itself. Thus, (&F) ( expression-listopt ) is simply (F) ( expression-listopt ), which is discussed in 13.3.1.1.1. If the function selected by overload resolution according to 13.3.1.1.1 is a non-static member function, the program is ill-formed. (The resolution of &F in other contexts is described in 13.4.)

**13.3.1.1.1 Call to named function** [over.call.func]

Of interest in 13.3.1.1.1 are only those function calls in which the postfix-expression ultimately contains a name that...

\[116\] 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 the function template specializations can be treated as normal (non-template) functions for the remainder of overload resolution.

\[117\] When F is a non-static member function, a reference of the form &A::f is a pointer-to-member, which cannot be used with the function-call syntax, and a reference of the form &F is an invalid use of the "&" operator on a non-static member function.
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.

In qualified function calls, the name to be resolved is an `id-expression` and is preceded by a `->` 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.[118] Under this assumption, the `id-expression` in the call is looked up as a member function of `T` following the rules for looking up names in classes (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).

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.[119] If the argument list is augmented by a contrived object and overload resolution selects one of the non-static member functions of `T`, the call is ill-formed.

### 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`”, then the set of candidate functions includes at least the function call operators of `T`. The function call operators of `T` are obtained by ordinary lookup of the name `operator()` in the context of `(E).operator()`.

In addition, for each conversion function declared in `T` of the form

```
operator conversion-type-id () cv-qualifier;
```

where `cv-qualifier` is the same `cv`-qualification as, or a greater `cv`-qualification than, `cv`, and where `conversion-type-id` denotes the type “pointer to function of `(P1,...,Pn)` returning `R`”, or the type “reference to pointer to function of `(P1,...,Pn)` returning `R`”, or the type “reference to function of `(P1,...,Pn)` returning `R`”, a surrogate call function with the unique name `call-function` and having the form

```
R call-function ( conversion-type-id F, P1 a1, ... ,Pn an) { return F (a1, ..., an); }
```

[118] Note that `cv`-qualifiers on the type of objects are significant in overload resolution for both lvalue and class rvalue objects.

[119] An implied object argument must be contrived to correspond to the implicit object parameter attributed to member functions during overload resolution. It is not used in the call to the selected function. Since the member functions all have the same implicit object parameter, the contrived object will not be the cause to select or reject a function.
is also considered as a candidate function. Similarly, surrogate call functions are added to the set of candidate functions for each conversion function declared in a base class of T provided the function is not hidden within T by another intervening declaration\(^{120}\).

3 If such a surrogate call function is selected by overload resolution, the corresponding conversion function will be called to convert E to the appropriate function pointer or reference, and the function will then be invoked with the arguments of the call. If the conversion function cannot be called (e.g., because of an ambiguity), the program is ill-formed.

4 The argument list submitted to overload resolution consists of the argument expressions present in the function call syntax preceded by the implied object argument (E). [Note: when comparing the call against the function call operators, the implied object argument is compared against the implicit object parameter of the function call operator. When comparing the call against a surrogate call function, the implied object argument is compared against the first parameter of the surrogate call function. The conversion function from which the surrogate call function was derived will be used in the conversion sequence for that parameter since it converts the implied object argument to the appropriate function pointer or reference required by that first parameter. — end note] [Example:

```c
int f1(int);
int f2(float);
typedef int (*fp1)(int);
typedef int (*fp2)(float);
struct A {
    operator fp1() { return f1; }
    operator fp2() { return f2; }
} a;
int i = a(1);  // calls f1 via pointer returned from
              // conversion function
```

— end example]

### 13.3.1.2 Operators in expressions

[over.match.oper]

1 If no operand of an operator in an expression has a type that is a class or an enumeration, the operator is assumed to be a built-in operator and interpreted according to clause 5. [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] [Example:

```c
class String {
public:
    String (const String&);
    String (char*);
    operator char* () ;
};
String operator + (const String&, const String&);
```

void f(void)
{  
\(^{120}\) Note that this construction can yield candidate call functions that cannot be differentiated one from the other by overload resolution because they have identical declarations or differ only in their return type. The call will be ambiguous if overload resolution cannot select a match to the call that is uniquely better than such undifferentiable functions.
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>

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:

- If T1 is a complete class type, 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.
- 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.
- 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,
  - have the same operator name, and
  - accept the same number of operands, and
  - accept operand types to which the given operand or operands can be converted according to 13.3.3.1, and
— do not have the same parameter-type-list as any non-template non-member candidate.

4 For the built-in assignment operators, conversions of the left operand are restricted as follows:
— no temporaries are introduced to hold the left operand, and
— no user-defined conversions are applied to the left operand to achieve a type match with the left-most parameter of a built-in candidate.

5 For all other operators, no such restrictions apply.

6 The set of candidate functions for overload resolution 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. [Example:

    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 are converted to the types of the corresponding parameters of the selected operation function. Then the operator is treated as the corresponding built-in operator and interpreted according to clause 5.

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. [222]

9 If the operator is the operator ,, the unary operator &, or the operator ->, and there are no viable functions, then the operator is assumed to be the built-in operator and interpreted according to 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:

    struct A { };  
    void operator + (A, A);

    struct B {
      void operator + (B);
      void f ();
    };

    A a;

  [221] If the set of candidate functions is empty, overload resolution is unsuccessful.
  [222] 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.

Draft
13.3 Overload resolution

```cpp
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 [over.match.ctor]

1 When objects of class type are direct-initialized (8.5), or copy-initialized from an expression of the same or a derived class type (8.5), overload resolution selects the constructor. For direct-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 within the parentheses of the initializer.

13.3.1.4 Copy-initialization of class by user-defined conversion [over.match.copy]

1 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. Assuming that “cv1 T” is the type of the object being initialized, with T a class type, the candidate functions are selected as follows:

— The converting constructors (12.3.1) of T are candidate functions.

— When the type of the initializer expression is a class type “cv S”, the conversion functions of S and its base classes are considered. Those that are not hidden within S and yield a type whose cv-unqualified version is the same type as T or is a derived class thereof are candidate functions. Conversion functions that return “reference to X” return lvalues or rvalues, depending on the type of reference, of type X and are therefore considered to yield X for this process of selecting candidate functions.

2 In both cases, the argument list has one argument, which is the initializer expression. [Note: this argument will be compared against the first parameter of the constructors and against the implicit object parameter of the conversion functions. — end note ]

13.3.1.5 Initialization by conversion function [over.match.conv]

1 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:

— The conversion functions of S and its base classes are considered. Those 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. 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 rvalues, depending on the type of reference, of type “cv2 X” and are therefore considered to yield X for this process of selecting candidate functions.

Draft
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 an lvalue 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:

— The conversion functions of S and its base classes are considered. Those that are not hidden within S and yield type “lvalue reference to cv2 T2”, where “cv1 T” is reference-compatible (8.5.3) with “cv2 T2”, are candidate functions.

The argument list has one argument, which is the initializer expression. [Note: this argument will be compared against the implicit object parameter of the conversion functions. — end note]

---

13.3.2 Viable functions

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.

First, to be a viable function, a candidate function shall have enough parameters to agree in number with the arguments in the list.

— If there are m arguments in the list, all candidate functions having exactly m parameters are viable.

— 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).

— 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.

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 a reference to non-const cannot be bound to an rvalue can affect the viability of the function (see 13.3.3.1.4).

---

13.3 Best Viable Function

Define ICS(F) as follows:

---

\(^{(123)}\) According to 8.3.6, parameters following the (m+1)-st parameter must also have default arguments.
13.3 Overload resolution

— if \( F \) is a static member function, \( \text{ICS}_1(F) \) is defined such that \( \text{ICS}_1(F) \) is neither better nor worse than \( \text{ICS}_1(G) \) for any function \( G \), and, symmetrically, \( \text{ICS}_1(G) \) is neither better nor worse than \( \text{ICS}_1(F) \); otherwise,

— 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

— for some argument \( j \), \( \text{ICS}_j(F_1) \) is a better conversion sequence than \( \text{ICS}_j(F_2) \), or, if not that,

— \( F_1 \) is a non-template function and \( F_2 \) is a function template specialization, or, if not that,

— \( F_1 \) and \( F_2 \) are function template specializations, and the function template for \( F_1 \) is more specialized than the template for \( F_2 \) according to the partial ordering rules described in 14.5.6.2, or, if not that,

— 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;
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 ]

If there is exactly one viable function that is a better function than all other viable functions, then it is the one selected by overload resolution; otherwise the call is ill-formed.

[Example:

```c
void Fcn(const int*, short);
void Fcn(int*, int);

int i;
short s = 0;
```

[24] If a function is a static member function, this definition means that the first argument, the implied object parameter, has no effect in the determination of whether the function is better or worse than any other function.

[25] 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.
void f() {
    Fcn(&i, s);  // is ambiguous because
    // &i → int* is better than &i → const int*
    // but s → short is also better than s → int

    Fcn(&i, 1L);  // calls Fcn(int*, int), because
    // &i → int* is better than &i → const int*
    // and 1L → short and 1L → int are indistinguishable

    Fcn(&i,'c');  // calls Fcn(int*, int), because
    // &i → int* is better than &i → const int*
    // and c → int is better than c → short
}

— end example }

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-declaration s — specify a default argument that made the function viable, the program is ill-formed. [ Example:

```c
namespace A {
    extern "C" void f(int = 5);
}

namespace B {
    extern "C" void f(int = 5);
}

using A::f;
using B::f;

void use() {
    f(3);  // OK, default argument was not used for viability
    f();   // Error: found default argument twice
}

— end example }

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 lvalue-ness 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, or accessibility of the argument and whether or not the argument is a bit-field 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:

---
- a standard conversion sequence (13.3.3.1.1),
- a user-defined conversion sequence (13.3.3.1.2), or
- an ellipsis conversion sequence (13.3.3.1.3).

However, when considering the argument of a user-defined conversion function that is a candidate by 13.3.1.3 when invoked for the copying of the temporary in the second step of a class copy-initialization, or by 13.3.1.4, 13.3.1.5, or 13.3.1.6 in all cases, only standard conversion sequences and ellipsis conversion sequences are allowed.

For the case where the parameter type is a reference, see 13.3.3.1.4.

When the parameter type is not a reference, the implicit conversion sequence models a copy-initialization of the parameter from the argument expression. The implicit conversion sequence is the one required to convert the argument expression to an rvalue 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. — 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).

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.

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).

If no sequence of conversions can be found to convert an argument to a parameter type or the conversion is otherwise ill-formed, an implicit conversion sequence cannot be formed.

If several different sequences of conversions exist that each convert the argument to the parameter type, the implicit conversion sequence associated with the parameter is defined to be the unique conversion sequence designated the ambiguous conversion sequence. For the purpose of ranking implicit conversion sequences as described in 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. [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).

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 { A (B&);};
```

Draft
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 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 lvalue-ness, 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 lvalue-ness or data representation of the type; and the Promotions and Conversions do not change the lvalue-ness or cv-qualification of the type. — 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.

### 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 must have exact match rank.

```cpp
class B { operator A () { }; }
class C { C (B&) { }; }
void f(A) { }
void f(C) { }
B b;
f(b); // ambiguous because b → C via constructor and
// b → 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).
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 copy 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.
A standard conversion sequence cannot be formed if it requires binding an lvalue reference to non-const to an rvalue (except when binding an implicit object parameter; see the special rules for that case in 13.3.1). [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-const lvalue reference to a bit-field (8.5.3). — end example]

The binding of a reference to an expression that is reference-compatible with added qualification influences the rank of a standard conversion; see 13.3.3.2 and 8.5.3.

13.3.3.2 Ranking implicit conversion sequences

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.

When comparing the basic forms of implicit conversion sequences (as defined in 13.3.3.1)

— 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

— 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).

Two implicit conversion sequences of the same form are indistinguishable conversion sequences unless one of the following rules applies:

— Standard conversion sequence S1 is a better conversion sequence than standard conversion sequence S2 if

  — S1 is a proper subsequence of S2 (comparing the conversion sequences in the canonical form defined by 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 S1 is better than the rank of S2, or S1 and S2 have the same rank and are distinguishable by the rules in the paragraph below; or, if not that,

  — S1 and S2 differ only in their qualification conversion and yield similar types T1 and T2 (4.4), respectively, and the cv-qualification signature of type T1 is a proper subset of the cv-qualification signature of type T2, and S1 is not the deprecated string literal array-to-pointer conversion (4.2). [Example:

```c
int f(const int *);
int f(int *);
int i;
int j = f(&i); // calls f(int*)
```

Draft
— end example ] or, if not that,

— S1 and S2 are reference bindings (8.5.3) and neither refers to an implicit object parameter, and either S1
binds an lvalue reference to an lvalue and S2 binds an rvalue reference or S1 binds an rvalue reference to an
rvalue and S2 binds an lvalue reference.

[ Example:

```cpp
int i;
int f();
int g(const int &);
int g(const int &&);
int j = g(i);  // calls g(const int &)
int k = g(f()); // calls g(const int &&)
```

```cpp
struct A { A& operator<< (int); };
A& operator<<(A&&, char);
A() << 1;  // calls A::operator<<(int)
A() << 'c';  // calls operator<<(A&&, char)
A a;
a << 1;  // calls A::operator<<(int)
a << 'c';  // calls operator<<(A&&, char)
```

— end example ] or, if not that,

— 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:

```cpp
int f(const int &);
int f(int &);
int g(const int &);
int g(int);
```

```cpp
int i;
int j = f(i);  // calls f(int &)
int k = g(i);  // ambiguous
```

```cpp
class X {
public:
    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 ]
— 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 and if the second standard conversion sequence of U1 is better than the second standard conversion sequence of U2. \[\text{Example:}\]

```cpp
struct A {
    operator short();
} a;
int f(int);
int f(float);
int i = f(a); // calls f(int), because short \rightarrow int is // better than short \rightarrow float.
```

— end example ]

4 Standard conversion sequences are ordered by their ranks: an Exact Match is a better conversion than a Promotion, which is a better conversion than a Conversion. Two conversion sequences with the same rank are indistinguishable unless one of the following rules applies:

— A conversion that is not a conversion of a pointer, or pointer to member, to bool is better than another conversion that is such a conversion.

— 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\*.

— If class B is derived directly or indirectly from class A and class C is derived directly or indirectly from B,

— conversion of C\* to B\* is better than conversion of C\* to A\*, [Example:

```cpp
struct A {}
struct B : public A {}
struct C : public B {}
C *pc;
int f(A *); // calls f(B*)
int f(B *);
int i = f(pc); // calls f(B*)
```

— end example ]

— binding of an expression of type C to a reference of type B\& is better than binding an expression of type C to a reference of type A\&,

— conversion of A::* to B::* is better than conversion of A::* to C::*,

— conversion of C to B is better than conversion of C to A,

— conversion of B to A is better than conversion of C to A.

— conversion of B::* to C::* is better than conversion of A::* to C::*, and

— conversion of B to A is better than conversion of C to A.

Draft
[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

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 matches the target type required in the context. The target can be

- an object or reference being initialized (8.5, 8.5.3),
- the left side of an assignment (5.17),
- a parameter of a function (5.2.2),
- a parameter of a user-defined operator (13.5),
- the return value of a function, operator function, or conversion (6.6.3),
- an explicit type conversion (5.2.3, 5.2.9, 5.4), or
- 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]

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;” the function type of the pointer to member is used to select the member function from the set of overloaded member functions. If a non-static member function is selected, the reference to the overloaded function name is required to have the form of a pointer to member as described in 5.3.1.

If more than one function is selected, any function template specializations in the set are eliminated if the set also contains a non-template function, and any given function template specialization F1 is eliminated if the set contains a second function template specialization whose function template is more specialized than the function template of F1 according to the partial ordering rules of 14.5.6.2. After such eliminations, if any, there shall remain exactly one selected function.

### Example:

```c
int f(double);
int f(int);
```
The initialization of `pfe` 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);
};
```

```c
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
```

```c
void g(D*);
void (*p2)(B*) = &g; // error
```

— end note ]

### 13.5 Overloaded operators

A function declaration having one of the following `operator-function-id` s as its name declares an `operator function`. A function template declaration having one of the following `operator-function-id` s 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`.

```c
operator-function-id:
    operator operator
```

Draft
13.5 Overloaded operators

**operator:** one of

\[
\begin{array}{cccccccccc}
  + & - & * & / & \% & ^ & & & & \\
  ! & = & < & > & += & -= & *= & /= & %= & \\
  \sim & \&= & \|= & \&< & \&> & \&>> & \&<< & \&== & \&!= \\
  \langle & \rangle & \&\& & \&\& & & & & & \\
\end{array}
\]

[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]

2 Both the unary and binary forms of

\[
  + \quad - \quad * \quad &
\]

can be overloaded.

3 The following operators cannot be overloaded:

\[
  . \quad .* \quad :: \quad ?:
\]

nor can the preprocessing symbols `#` and `##` (clause 16).

4 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:

```cpp
complex z = a.operator+(b); //complex z = a+b;
void* p = operator new(sizeof(int)*n);
```

— end example]

5 The allocation and deallocation functions, `operator new`, `operator new[]`, `operator delete` and `operator delete[]`, are described completely in 3.7.3. The attributes and restrictions found in the rest of this subclause do not apply to them unless explicitly stated in 3.7.3.

6 An operator function shall either be a non-static member function or be a non-member function and have at least one parameter whose type is a class, a reference to a class, an enumeration, or a reference to an enumeration. It is not possible to change the precedence, grouping, or number of operands of operators. The meaning of the operators `=`, `(unary) &`, and `, (comma), predefined for each type, can be changed for specific class and enumeration types by defining operator functions that implement these operators. Operator functions are inherited in the same manner as other base class functions.

7 The identities among certain predefined operators applied to basic types (for example, `++a \equiv a+=1`) need not hold for operator functions. Some predefined operators, such as `+=`, require an operand to be an lvalue when applied to basic types; this is not required by operator functions.

8 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.

Draft
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 assignment operator, can be virtual. [Note: for a derived class D with a base class B for which a virtual copy assignment has been declared, the copy assignment operator in D does not override B’s virtual copy assignment operator. [Example:]

```cpp
struct B {
    virtual int operator=(int);
    virtual B& operator=(const B&);
};
struct D : B {
    virtual int operator=(int);
    virtual D& operator=(const B&);
};

D dobj1;
D dobj2;
B* bptr = &dobj1;
void f() {
    bptr->operator=(99); // calls D::operator=(int)
    *bptr = 99;          // ditto
    bptr->operator=(dobj2); // calls D::operator=(const B&)
    *bptr = dobj2;       // ditto
    dobj1 = dobj2;       // calls implicitly-declared
                       // D::operator=(const D&)
}
```

Draft
13.5 Overloaded operators

— 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

\[
\text{postfix-expression ( expression-list_{opt} )}
\]

where the \text{postfix-expression} evaluates to a class object and the possibly empty \text{expression-list} matches the parameter list of an \text{operator()} member function of the class. Thus, a call \text{x(arg1,...)} is interpreted as \text{x.operator() (arg1, ...)} for a class object x of type T if T::operator() (T1, T2, T3) 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

\[
\text{postfix-expression [ expression ]}
\]

Thus, a subscripting expression \text{x[y]} is interpreted as \text{x.operator[](y)} for a class object x of type T if T::operator[](T1) exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3.3).

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 ->.

\[
\text{postfix-expression -> template_{opt} id-expression}
\]

\[
\text{postfix-expression -> pseudo-destructor-name}
\]

An expression \text{x->m} is interpreted as \text{(x.operator->())->m} for a class object x of type T if T::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 \text{operator++} implements the prefix and postfix ++ operator. If this function is a member function with no parameters, or a non-member function with one parameter of class or enumeration type, it defines the prefix increment operator ++ for objects of that type. If the function is a member function with one parameter (which shall be of type \text{int}) or a non-member function with two parameters (the second of which shall be of type \text{int}), it defines the postfix increment operator ++ for objects of that type. When the postfix increment is called as a result of using the ++ operator, the \text{int} argument will have value zero.\footnote{Calling \text{operator++} explicitly, as in expressions like \text{a.operator++(2)}, has no special properties: The argument to \text{operator++} is 2.} [ Example:

```cpp
class X {
public:
    X& operator++(); // prefix ++
    X operator++(int); // postfix a++
};

class Y { };
Y& operator++(Y&); // prefix ++
```

Draft
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.

13.6 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

\[
\begin{align*}
VQ &\ T k\ &\ operator++(VQ\ Tk); \\
T &\ operator++(VQ\ Tk, \ int);
\end{align*}
\]

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

\[
\begin{align*}
VQ &\ T k\ &\ operator--(VQ\ Tk); \\
T &\ operator--(VQ\ Tk, \ int);
\end{align*}
\]
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

\[
\begin{align*}
&T\*VQ\& \text{ operator++(}T\*VQ\&); \\
&T\*VQ\& \text{ operator--(}T\*VQ\&); \\
&T\* \text{ operator++(}T\*VQ\&, \text{ int}); \\
&T\* \text{ operator--(}T\*VQ\&, \text{ int});
\end{align*}
\]

For every cv-qualified or cv-unqualified object type \(T\), there exist candidate operator functions of the form

\[
\begin{align*}
&T\& \text{ operator*}(T\&); \\
&T\& \text{ operator*(}T\&); \\
&T\* \text{ operator+}(T\*); \\
&T \text{ operator*}(T); \\
&T \text{ operator-}(T); \\
&T \text{ operator~}(T);
\end{align*}
\]

For every promoted arithmetic type \(T\), there exist candidate operator functions of the form

\[
\begin{align*}
&T \text{ operator+}(T); \\
&T \text{ operator-}(T);
\end{align*}
\]

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 \texttt{cv-qualifier-seq} s, there exist candidate operator functions of the form

\[
\text{CV12} \ T\& \text{ operator->*}(CV1 C1\&, CV2 T C2::*);
\]

where \(CV12\) is the union of \(CV1\) and \(CV2\).

For every pair of promoted arithmetic types \(L\) and \(R\), there exist candidate operator functions of the form

\[
\begin{align*}
&LR \text{ operator*(}L, R); \\
&LR \text{ operator/}(L, R); \\
&LR \text{ operator+}(L, R); \\
&LR \text{ operator-}(L, R); \\
&bool \text{ operator<}(L, R); \\
&bool \text{ operator<=}(L, R); \\
&bool \text{ operator>}(L, R); \\
&bool \text{ operator>=}(L, R); \\
&bool \text{ operator==}(L, R); \\
&bool \text{ operator!=}(L, R);
\end{align*}
\]

where \(LR\) is the result of the usual arithmetic conversions between types \(L\) and \(R\).

For every cv-qualified or cv-unqualified object type \(T\) there exist candidate operator functions of the form

Draft
For every $T$, where $T$ is a pointer to object type, there exist candidate operator functions of the form

\[
\text{std::ptrdiff_t operator-}(T, T);
\]

For every pointer or enumeration type $T$, there exist candidate operator functions of the form

\[
\text{bool operator<}(T, T);
\]
\[
\text{bool operator>}(T, T);
\]
\[
\text{bool operator<=(T, T)};
\]
\[
\text{bool operator>=(T, T)};
\]
\[
\text{bool operator==}(T, T);
\]
\[
\text{bool operator!=}(T, T);
\]

For every pointer to member type $T$, there exist candidate operator functions of the form

\[
\text{bool operator==}(T, T);
\]
\[
\text{bool operator!=}(T, T);
\]

For every pair of promoted integral types $L$ and $R$, there exist candidate operator functions of the form

\[
\text{LR operator%}(L, R);
\]
\[
\text{LR operator&}(L, R);
\]
\[
\text{LR operator^}(L, R);
\]
\[
\text{LR operator|}(L, R);
\]
\[
\text{L operator<<}(L, R);
\]
\[
\text{L operator>>(L, R)};
\]

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 volatile or empty, and $R$ is a promoted arithmetic type, there exist candidate operator functions of the form

\[
\text{VQ Lk operator=(VQ Lk, R)};
\]
\[
\text{VQ Lk operator**=(VQ Lk, R)};
\]
\[
\text{VQ Lk operator/=(VQ Lk, R)};
\]
\[
\text{VQ Lk operator+=(VQ Lk, R)};
\]
\[
\text{VQ Lk operator-=(VQ Lk, R)};
\]

For every pair $(T, VQ)$, where $T$ is any type and $VQ$ is either volatile or empty, there exist candidate operator functions of the form

\[
T*VQk operator=(T*VQk, T*);
\]

For every pair $(T, VQ)$, where $T$ is an enumeration or pointer to member type and $VQ$ is either volatile or empty, there exist candidate operator functions of the form
13.6 Built-in operators

\[ VQ\ Tk \ \text{operator\(=(VQ\ Tk,\ T); \}\]

21 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*}
VQ\ Tk & \ \text{operator\(+(VQ\ Tk,\ T);\)} \\
VQ\ Tk & \ \text{operator\(-(VQ\ Tk,\ T);\)} \\
T*VQk & \ \text{operator\(+=(T*VQ&\,\,\,\,\text{std::ptrdiff_t});\)} \\
T*VQk & \ \text{operator\(-=(T*VQ&\,\,\,\text{std::ptrdiff_t});\)}
\end{align*}
\]

22 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\ Lk & \ \text{operator\%=\((VQ\ Lk,\ R);\)} \\
VQ\ Lk & \ \text{operator\<<\((VQ\ Lk,\ R);\)} \\
VQ\ Lk & \ \text{operator\>>\((VQ\ Lk,\ R);\)} \\
VQ\ Lk & \ \text{operator\&\=((VQ\ Lk,\ R);\)} \\
VQ\ Lk & \ \text{operator\^\=((VQ\ Lk,\ R);\)} \\
VQ\ Lk & \ \text{operator\|\=((VQ\ Lk,\ R);\)}
\end{align*}
\]

23 There also exist candidate operator functions of the form

\[
\begin{align*}
\text{bool} & \ \text{operator\!(\text{bool);\)} \\
\text{bool} & \ \text{operator\&\&\((\text{bool,\ bool);\)} \\
\text{bool} & \ \text{operator\|\((\text{bool,\ bool);\)}
\end{align*}
\]

24 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);\)}
\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]

25 For every type \(T\), where \(T\) is a pointer or pointer-to-member type, there exist candidate operator functions of the form

\[
\begin{align*}
T & \ \text{operator\?(bool,\ T,\ T);\)}
\end{align*}
\]

Draft
Chapter 14  Templates

1 A template defines a family of classes or functions, or an alias for a family of types.

   template-declaration:
       exportopt template < template-parameter-list > declaration

   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
   — declare or define a function or a class, or
   — define a member function, a member class or a static data member of a class template or of a class nested within a class template, or
   — define a member template of a class or class template, or
   — be an alias-declaration.

A template-declaration is a declaration. A template-declaration is also a definition if its declaration defines a function, a class, or a static data member.

2 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 be a template-name or operator-function-id (i.e., not a template-id).

[ Note: 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 ]

3 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.

4 A template name has linkage (3.5). A non-member function template can have internal linkage; any other template name shall have external linkage. Entities generated from a template with internal linkage are distinct from all entities generated 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, object, 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 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 template-declaration may be preceded by the `export` keyword. Such a template is said to be exported. Declaring exported a class template is equivalent to declaring exported all of its non-inline member functions, static data members, member classes, member class templates, and non-inline member function templates.

If a template is exported in one translation unit, it shall be exported in all translation units in which it appears; no diagnostic is required. A declaration of an exported template shall appear with the `export` keyword before any point of instantiation (14.6.4.1) of that template in that translation unit. In addition, the first declaration of an exported template containing the `export` keyword must not follow the definition of that template. The `export` keyword shall not be used in a friend declaration.

Templates defined in an unnamed namespace, inline functions, and inline function templates shall not be exported. An exported non-class template shall be defined only once in a program; no diagnostic is required. An exported non-class template need only be declared (and not necessarily defined) in a translation unit in which it is instantiated.

A non-exported non-class template must 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.

[Note: an implementation may require that a translation unit containing the definition of an exported template be compiled before any translation unit containing an instantiation of that template. — end note]

A template-declaration that declares a template alias (14.5.7) shall not be exported.

### 14.1 Template parameters

The syntax for `template-parameter`s is:

```
template-parameter:
  type-parameter
  parameter-declaration
type-parameter:
  class . . . opt identifier opt
  class identifier opt = type-id
  typename . . . opt identifier opt
  typename identifier opt = type-id
template < template-parameter-list > class . . . opt identifier opt
template < template-parameter-list > class identifier opt = id-expression
```

[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 `template-parameter`. `typename` followed by an `unqualified-id` names a template type parameter. `typename` followed by a `qualified-id` denotes the type in a non-type parameter-declaration. A storage class shall not be specified in a `template-parameter` declaration. [Note: a template...

128) Since template `template-parameter`s and template `template-argument`s 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.
parameter may be a class template. For example,

```cpp
template<class T> class myarray { /* ... */ };
```

template<class K, class V, template<class T> class C = myarray>
class Map {
    C<K> key;
    C<V> value;
    // ...
};
```

— end note ]

A type-parameter whose identifier does not follow an ellipsis defines its identifier to be a typedef-name (if declared with class or typename) or template-name (if declared with template) in the scope of the template declaration. [ Note: because of the name lookup rules, a template-parameter that could be interpreted as either a non-type template-parameter or a type-parameter (because its identifier is the name of an already existing class) is taken as a type-parameter. For example,

```cpp
class T { /* ... */ };
int i;
```

template<class T, T i> void f(T t)
{
    T t1 = i;  // template-parameters T and i
    ::T t2 = ::i;  // global namespace members T and i
}
```

Here, the template f has a type-parameter called T, rather than an unnamed non-type template-parameter of class T. — end note ]

A non-type template-parameter shall have one of the following (optionally cv-qualified) types:

- integral or enumeration type,
- pointer to object or pointer to function,
- reference to object or reference to function,
- pointer to member.

[ Note: other types are disallowed either explicitly below or implicitly by the rules governing the form of template-argument s (14.3). — end note ] The top-level cv-qualifiers on the template-parameter are ignored when determining its type.

A non-type non-reference template-parameter is not an lvalue. 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
}
```

Draft


### 14.1 Template parameters

```
&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:

```
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. A default template-argument may be specified in a template declaration. A default template-argument shall not be specified in the template-parameter-list 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-argument s 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 ]
If a template-parameter of a class 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 class template is a template parameter pack, it must be the last template-parameter. [Note: These are not requirements for function templates because template arguments might be deduced (14.8.2).] [Example:

```cpp
template<class T1 = int, class T2> class B; // error
```

— end example ] — end note ]

A template-parameter shall not be given default arguments by two different declarations in the same scope. [Example:

```cpp
template<class T = int> class X;
template<class T = int> class X { /* ... */ }; // error
```

— end example ]

The scope of a template-parameter extends from its point of declaration until the end of its template. In particular, a template-parameter can be used in the declaration of subsequent template-parameters and their default arguments. [Example:

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

— end example ]

A template-parameter shall not be used in its own default argument.

When parsing a default template-argument for a non-type template-parameter, the first non-nested > is taken as the end of the template-parameter-list rather than a greater-than operator. [Example:

```cpp
template<int i = 3 > 4 > class X { /* ... */ }; // syntax error
```

```cpp
template<int i = (3 > 4) > class Y { /* ... */ }; // OK
```

— end example ]

A template-parameter of a template template-parameter is permitted to have a default template-argument. When such default arguments are specified, they apply to the template template-parameter in the scope of the template template-parameter. [Example:

```cpp
template <class T = float> struct B {};    // error - TT has no default template argument
```
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). [Example:

```cpp
template <class... Types> class Tuple; // Types is a template type parameter pack
template <class T, int... Dims> struct multi_array; // Dims is a non-type template parameter pack
```
—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 >

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]

For a template-name to be explicitly qualified by the template arguments, the name must be known to refer to a template.

After name lookup (3.4) finds that a name is a template-name, or that an operator-function-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 >> is taken as the ending delimiter rather than a greater-than operator. Similarly, the first non-nested >> is treated as two consecutive but distinct > tokens, the first of which is taken as the end of the template-argument-list and completes the template-id. [Note: The second > token produced by this replacement rule may terminate an enclosing template-id construct or it may be part of a different construct (e.g. a cast). —end note] [Example:

```cpp
template<int i> class X { /* ... */

template<>() class X { /* ... */

X< 1>2 > x1; // syntax error
X<(1>2)> x2; // OK
```

(290) A > that encloses the type-id of a dynamic_cast, static_cast, reinterpret_cast or const_cast, or which encloses the template-argument of a subsequent template-id, is considered nested for the purpose of this description.
template<class T> class Y { /* ... */
};
Y<X<1>> x3;  // OK, same as Y<X<1> > x3;
Y<X<6>>1>> x4;  // syntax error
Y<X<(6>>1)>> x5;  // OK

— end example ]

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 postfix-expression or qualified-id explicitly depends on a template-parameter (14.6.2) but does not refer to 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:

    class X {
    public:
    template<std::size_t> X* alloc();
    template<std::size_t> static X* adjust();
    
    template<class T> void f(T* p)
    {
    T* p1 = p->alloc<200>();
    // ill-formed: < means less than
    
    T* p2 = p->template alloc<200>();
    // OK: < starts template argument list
    
    T::adjust<100>();
    // ill-formed: < means less than
    
    T::template adjust<100>();
    // OK: < starts template argument list
    
    }
    — end example ]

5 If a name prefixed by the keyword template is not the name of a template, the program is ill-formed. [ Note: the keyword template may not be applied to non-template members of class templates. — end note ] [ Note: as is the case with the typename prefix, the template prefix is allowed in cases where it is not strictly necessary; i.e., when the nested-name-specifier or the expression on the left of the -> or . is dependent on a template-parameter, or the use does not appear in the scope of a template. — end note ]

6 A simple-template-id that names a class template specialization is a class-name (clause 9).

7 A template-id that names a template alias specialization is a type-name.

14.3 Template arguments

There are three forms of template-argument, corresponding to the three forms of template-parameter: type, non-type and template. The type and form of each template-argument specified in a template-id shall match the type and form specified for the corresponding parameter declared by the template in its template-parameter-list. When the parameter declared by the template is a template parameter pack, it will correspond to zero or more template-argument s. [ Example:

    Draft
template<class T> class Array {
    T* v;
    int sz;
public:
    explicit Array(int);
    T& operator[](int);
    T& elem(int i) { return v[i]; }
    // ...
};

Array<int> v1(20);
typedef std::complex<double> dcomplex; // std::complex is a standard library template
Array<dcomplex> v2(30);
Array<dcomplex> v3(40);

void bar() {
    v1[3] = 7;
    v2[3] = v3.elem(4) = dcomplex(7,8);
}

— end example }

In a template-argument, an ambiguity between a type-id and an expression is resolved to a type-id, regardless of the form of the corresponding template-parameter.\[130\] [Example:

    template<class T> void f();
    template<int I> void f();

    void g()
    {
        f<int>(); // int() is a type-id: call the first f()
    }

— end example ]

The name of a template-argument shall be accessible at the point where it is used as a template-argument. [Note: if the name of the template-argument is accessible at the point where it is used as a template-argument, there is no further access restriction in the resulting instantiation where the corresponding template-parameter name is used. —end note ]

[Example:

    template<class T> class X {
        static T t;
    };

class Y {
    private:
        struct S { /* ... */};

\[130\] 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.
X<S> x;          // OK: S is accessible
    // X<S> has a static member of type Y::S:
    // OK: even though Y::S is private
};

X<Y::S> y;       // error: S not accessible

— end example —

For a template-argument that is a class type or a class template, the template definition has no special access rights to the members of the template-argument. [ Example:

template <template <class TT> class T> class A {
    typename T<int>::S s;
};

template <class U> class B {
private:
   struct S { /* ... */ };   
};

A<B> b;           // ill-formed: A has no access to B::S

— end example —

When template argument packs or default template-argument s are used, a template-argument list can be empty. In that case the empty <> brackets shall still be used as the template-argument-list. [ Example:

    template<class T = char> class String;
    String<> p;          // OK: String<char>
    String* q;           // syntax error

    template<class ... Elements> class Tuple;
    Tuple<> t;           // OK: Elements is empty
    Tuple* u;            // syntax error

— end example —

An explicit destructor call (12.4) for an object that has a type that is a class template specialization may explicitly specify the template-argument s. [ Example:

    template<class T> struct A {
        ~A();
    };
    void f(A<int>* p, A<int>* q) {
        p->A<int>::~A();            // OK: destructor call
        q->A<int>::~A<int>();       // OK: destructor call
    }

— end example —

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.
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-argument s do not match the template-parameter s are ignored. If none of the function templates have matching template-parameter s, the program is ill-formed.

A template-argument followed by an ellipsis is a pack expansion (14.5.3). A template-argument pack expansion shall not occur in a simple-template-id whose template-name refers to a class template unless the template-parameter-list of that class template declares a template parameter pack.

### 14.3.1 Template type arguments

1. A template-argument for a template-parameter which is a type shall be a type-id.

2. A type without linkage (3.5) shall not be used as a template-argument for a template type-parameter.

   [Example:
   ```
   template <class T> class X { /* */ }; 
   void f()
   {
       struct S { /* */ }; 
       X<S> x3; // error: local type used as template-argument 
       X<S*> x4; // error: pointer to local type used as template-argument 
   }
   ```
   — end example ] [ Note: a template type argument may be an incomplete type (3.9). — end note ]

3. If a declaration acquires a function type through a type dependent on a template-parameter and this causes a declaration that does not use the syntactic form of a function declarator to have function type, the program is ill-formed. [ Example:
   ```
   template<class T> struct A {
       static T t;
   }
   typedef int function();
   A<function> a; // ill-formed: would declare A<function>::t 
                  // as a static member function
   ```
   — end example ]

4. If a template-argument for a template-parameter T names a type that is a reference to a type A, an attempt to create the type “lvalue reference to cv T” creates the type “lvalue reference to A,” while an attempt to create the type type “rvalue reference to cv T” creates the type T [ Example:
   ```
   template <class T> class X {
       void f(const T&);
       void g(T&&);
       /* */
   };
   X<int&> x1; //X<int&>::f has the parameter type int& 
              //X<int&>::g has the parameter type int&
   X<const int&&> x2; //X<const int&&>::f has the parameter type const int& 
                    //X<const int&&>::g has the parameter type const int&
   ```
   Draft
14.3.2 Template non-type arguments

A template-argument for a non-type, non-template template-parameter shall be one of:

- an integral constant expression of integral or enumeration type; or
- the name of a non-type template-parameter; or
- the address of an object or function with external linkage, including function templates and function template-ids but excluding non-static class members, expressed as &id-expression where the & is optional if the name refers to a function or array, or if the corresponding template-parameter is a reference; or
- a constant expression that evaluates to a null pointer value (4.10); or
- a constant expression that evaluates to a null member pointer value (4.11); or
- a pointer to member expressed as described in 5.3.1.

Note: A string literal (2.13.4) does not satisfy the requirements of any of these categories and thus is not an acceptable template-argument. [Example:

template<class T, char* p> class X {
// ...
X();
X(const char* q) { /* ... */ }
};

X<int, "Studebaker"> x1;  // error: string literal as template-argument

c char p[] = "Vivisectionist";
X<int, p> x2;              // OK

— end example — end note ]

Note: Addresses of array elements and names or addresses of non-static class members are not acceptable template-arguments. [Example:

template<int* p> class X { }; int a[10];
struct S { int m; static int s; } s;

X<&a[2]> x3;            // error: address of array element
X<&a.m> x4;             // error: address of non-static member
X<&a.s> x5;             // error: &S::s must be used
X<&S::s> x6;            // OK: address of static member

— end example — end note ]

Note: Temporaries, unnamed lvalues, and named lvalues that do not have external linkage are not acceptable template-arguments when the corresponding template-parameter has reference type. [Example:
14.3 Template arguments

The following conversions are performed on each expression used as a non-type template-argument. If a non-type template-argument cannot be converted to the type of the corresponding template-parameter then the program is ill-formed.

— for a non-type template-parameter of integral or enumeration type, integral promotions (4.5) and integral conversions (4.7) are applied.

— for a non-type template-parameter of type pointer to object, qualification conversions (4.4) and the array-to-pointer conversion (4.2) are applied. [Note: In particular, neither the null pointer conversion (4.10) nor the derived-to-base conversion (4.10) are applied. Although 0 is a valid template-argument for a non-type template-parameter of integral type, it is not a valid template-argument for a non-type template-parameter of pointer type. However, (int*)0 is a valid template-argument for a non-type template-parameter of type “pointer to int.” — end note]

— For a non-type template-parameter of type reference to object, no conversions apply. The type referred to by the reference may be more cv-qualified than the (otherwise identical) type of the template-argument. The template-parameter is bound directly to the template-argument, which must be an lvalue.

— For a non-type template-parameter of type pointer to function, only the function-to-pointer conversion (4.3) is applied. If the template-argument represents a set of overloaded functions (or a pointer to such), the matching function is selected from the set (13.4).

— For a non-type template-parameter of type reference to function, no conversions apply. If the template-argument represents a set of overloaded functions, the matching function is selected from the set (13.4).

— For a non-type template-parameter of type pointer to member function, no conversions apply. If the template-argument represents a set of overloaded member functions, the matching member function is selected from the set (13.4).

— For a non-type template-parameter of type pointer to data member, qualification conversions (4.4) are applied.

[Example:

```cpp
template<const int& CRI> struct B { /* ... */ };
B<1> b2; // error: temporary would be required for template argument
int c = 1;
B<c> b1; // OK
```

```cpp
int ai[10];
X<ai> xi; // array to pointer and qualification conversions
```

```cpp
struct Y { /* ... */ };
template<const Y& b> struct Z { /* ... */ };
Y y;
Z<y> z; // no conversion, but note extra cv-qualification
```

```cpp
template<int (&pa)[5]> struct W { /* ... */ };
```

Draft
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 ]

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 a template alias, 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 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<typename class U> class V> class C {
    V<int> y;
    V<int*> z;
};
C<A> c;
    // V<int> within C<A> uses the primary template,
    // so c.y.x has type int
    // V<int*> within C<A> uses the partial specialization,
    // so c.z.x has type long
— end example ]

[ Example:

template<class T> class A { /* */ };  // primary template
template<class T, class U = T> class B { /* */ };  // partial specialization
template<class ... Types> class C { /* */ };  // so c.y.x has type int

C<A> xa;      // OK

Draft
14.4 Type equivalence

Two template-id s refer to the same class or function if

— their template-name s refer to the same template, and
— their corresponding type template-argument s 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-argument s of pointer type refer to the same external object or function or are both the null pointer value, and
— their corresponding non-type template-argument s 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-argument s of reference type refer to the same external object or function, and
— their corresponding template template-argument s 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<template<class> class TT> struct X \{
```
```cpp
template<class> struct Y { 
```
```cpp
template<class T> using Z = Y<T>;
X<Y> y;
X<Z> z;
```

declares y and z to be of the same type. — end example ]
14.5 Template declarations

A template-id, that is, the template-name followed by a template-argument-list shall not be specified in the declaration of a primary template declaration. [Example:

```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]

For purposes of name lookup and instantiation, default arguments of function templates and default arguments of member functions of class templates are considered definitions; each default argument is a separate definition which is unrelated to the function template definition or to any other default arguments.

Because an alias-declaration cannot declare a template-id, it is not possible to partially or explicitly specialize a template alias.

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:

```cpp
template<class T> class Array {
  T* v;
  int sz;
public:
  explicit Array(int);
  T& operator[](int);
  T& elem(int i) { return v[i]; }
  // ...
};
```
— end example] 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]

When a member function, a member class, 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 s are those of the class template. The names of the template parameters used in the definition of the member may be different from the template parameter names used in the class template definition. The template argument list following the class template name in the member definition shall name the parameters in the same order as the one used in the template parameter list of the member. Each template parameter pack shall be expanded with an ellipsis in the template argument list. [Example:

```cpp
template<class T1, class T2> struct A {
  void f1();
  void f2();
};
template<class T2, class T1> void A<T2,T1>::f1() { } // OK
```
— end example]
14.5 Template declarations

Templates

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]

2 The template-argument s for a member function of a class template are determined by the template-argument s 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);
```

```cpp
v1[3] = 7;  // Array<int>::operator[]()
v2[3] = dcomplex(7,8);  // Array<dcomplex>::operator[]()
```
14.5.1.2 Member classes of class templates

A class member of a class template may be defined outside the class template definition in which it is declared. [Note: the class member 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 { };  // 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 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;
```

— end example —

14.5.2 Member templates

A template can be declared within a class or class template; such a template is called a member template. A member template can be defined within or outside its class definition or class template definition. A member template of a class template that is defined outside of its class template definition shall be specified with the `template-parameter`s of the class template followed by the `template-parameter`s of the member template. [Example:

```cpp
template<class T> class string {
    public:
        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 shall not have member templates. Access control rules (clause 11) apply to member template names. A destructor shall not be a member template. A normal (non-template) member function with a given name and type and a member function template of the same name, which could be used to generate a specialization of the same type, can
both be declared in a class. When both exist, a use of that name and type refers to the non-template member unless an explicit template argument list is supplied. [Example:

```cpp
template <class T> struct A {
    void f(int);
    template <class T2> void f(T2);
};

template <> void A<int>::f(int) { } // non-template member
template <> template <> void A<int>::f<>(int) { } // template member

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:

```cpp
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:

```cpp
class B {
    virtual void f(int);
};

class D : public B {
    template <class T> void f(T);    // does not override B::f(int)
    void f(int i) { f<>(i); }        // overriding function that calls
                                        // the template instantiation
};
```
— end example ]

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:

```cpp
struct A {
    template <class T> operator T*();
};

template <class T> A::operator T*(){ return 0; }
template <> A::operator char*(){ return 0; } // specialization
```
template A::operator void*();  // explicit instantiation

int main()
{
    A a;
    int* ip;

    ip = a.operator int*();     // explicit call to template operator
    // A::operator int*()
}

— end example ] [ Note: because the explicit template argument list follows the function template name, and because
conversion member function templates and constructor member function templates are called without using a function
name, there is no way to provide an explicit template argument list for these function templates. — end note ]

A specialization of a conversion function template is not found by name lookup. Instead, any conversion function
templates visible in the context of the use are considered. For each such operator, if argument deduction succeeds
(14.8.2.3), the resulting specialization is used as if found by name lookup.

A using-declaration in a derived class cannot refer to a specialization of a conversion function template in a base class.

Overload resolution (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 [temp.variadic]

A template parameter pack is a template parameter that accepts zero or more template arguments. [ Example:

    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

    — end example ]

A function parameter pack is a function parameter that accepts zero or more function arguments. [ Example:

    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

    — end example ]

A parameter pack is either a template parameter pack or a function parameter pack.

A pack expansion is a sequence of tokens that names one or more parameter packs, followed by an ellipsis. The sequence
of tokens is called the pattern of the expansion; its syntax depends on the context in which the expansion occurs. Pack
expansions can occur in the following contexts:
— In an expression-list (5.2); the pattern is an assignment-expression.
— In an initializer-list (8.5); the pattern is an initializer-clause.
— In a base-specifier-list (10); the pattern is a base-specifier.
— In a mem-initializer-list (12.6.2); the pattern is a mem-initializer.
— In a template-argument-list (14.3); the pattern is a template-argument.
— In an exception-specification (15.4); the pattern is a type-id.

[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. 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) {
    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 an expansion produces a comma-separated list $E_1, E_2, \ldots, E_N$, where $N$ is the number of elements in the pack expansion parameters. Each $E_i$ is generated by instantiating the pattern and replacing each pack expansion
parameter with its $i$th element. 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]

### 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 an ordinary (non-template) function or class. For a friend function declaration that is not a template declaration:

- 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
- 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,
- if the name of the friend is a `qualified-id` and a matching specialization of a function template is found in the specified class or namespace, the friend declaration refers to that function template specialization, otherwise,
- the name shall be an `unqualified-id` that declares (or redeclares) an ordinary (non-template) function.

[Example:

```
template<class T> class task;
template<class T> task<T>* preempt(task<T>*);

template<class T> class task {
    // ...
    friend void next_time();
    friend void process(task<T>*);
    friend task<T>* preempt<T>(task<T>*);
    template<class C> friend int func(C);

    friend class task<int>;
    template<class P> friend class frd;
    // ...
};
```

Here, each specialization of the `task` class template has the function `next_time` as a friend; because `process` does not have explicit `template-argument` $s$, 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]

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:
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 used. 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 class template is a friend of the class granting friendship. 

Example:
```cpp
template<class T> struct A {
    struct B { };
    void f();
};

class C {
    template<class T> friend struct A<T>::B;
    template<class T> friend void A<T>::f();
};
```

--- end example ]

Note: a friend declaration may first declare a member of an enclosing namespace scope (14.6.5). — end note ]

A friend template shall not be declared in a local class.

Friend declarations shall not declare partial specializations. 

Example:
```cpp
template<class T> class A { };
class X {
    template<class T> friend class A<T*>;    // error
};
```

--- end example ]
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.

When a partial specialization is used within the instantiation of an exported template, and the unspecialized template name is non-dependent in the exported template, a declaration of the partial specialization must be declared before the definition of the exported template, in the translation unit containing that definition. A similar restriction applies to explicit specialization; see 14.7.

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).

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,

```
template<class T1, class T2, int I> class A { }; // error
```

— end note] A class template partial specialization may be declared or redeclared in any namespace scope in which its definition may be defined (14.5.1 and 14.5.2). [Example:

```
template<class T> struct A { 
 struct C { 
   template<class T2> struct B { }; 
};
```
14.5 Template declarations

7 Partial specialization declarations themselves are not found by name lookup. Rather, when the primary template name is used, any previously declared partial specializations of the primary template are also considered. One consequence is that a using-declaration which refers to a class template does not restrict the set of partial specializations which may be found through the using-declaration. [Example:

    namespace N {
      template<class T1, class T2> class A { }; // primary template
    }

    using N::A; // refers to the primary template

    namespace N {
      template<class T> class A<T, T*> { }; // partial specialization
    }

    A<int,int*> a; // uses the partial specialization, which is found through
    // the using declaration which refers to the primary template

    — end example ]

8 A non-type argument is non-specialized if it is the name of a non-type parameter. All other non-type arguments are specialized.

9 Within the argument list of a class template partial specialization, the following restrictions apply:

   — 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:

        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 ]

   — The type of a template parameter corresponding to a specialized non-type argument shall not be dependent on a parameter of the specialization. [Example:

        template <class T, T t> struct C {};  
        template <class T> struct C<T, T>;    // error

    Draft
template< int X, int (*array_ptr)[X] > class A {}

int array[5];

template< int X > class A<X,&array> { };    // error

— end example }

— The argument list of the specialization shall not be identical to the implicit argument list of the primary template.

— The template parameter list of a specialization shall not contain default template argument values.\(^\text{131}\)

— 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

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.

— If exactly one matching specialization is found, the instantiation is generated from that specialization.

— 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.

— 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). \([\text{Example}:
\begin{align*}
A\langle\text{int, int, 1}\rangle & \ a1; \quad \text{// uses #1} \\
A\langle\text{int, int*}, 1\rangle & \ a2; \quad \text{// uses #2, T is int, I is 1} \\
A\langle\text{int, char*}, 5\rangle & \ a3; \quad \text{// uses #4, T is char} \\
A\langle\text{int, char*}, 1\rangle & \ a4; \quad \text{// uses #5, T1 is int, T2 is char, I is 1} \\
A\langle\text{int*}, \text{int*}, 2\rangle & \ a5; \quad \text{// ambiguous: matches #3 and #5}
\end{align*}
\]

— 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. \([\text{Example: the declaration of a2 above.} \ — \text{end example} ]\]

In a type name that refers to a class template specialization, (e.g., \(A\langle\text{int, int, 1}\rangle\)) the argument list must match the template parameter list of the primary template. The template arguments of a specialization are deduced from the arguments of the primary template.

\(^{131}\) There is no way in which they could be used.
14.5 Template declarations

14.5.5.2 Partial ordering of class template specializations

For two class template partial specializations, the first is at least as specialized as the second if, given the following rewrite to two function templates, the first function template is at least as specialized as the second according to the ordering rules for function templates (14.5.6.2):

— 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

— 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:

```
template<int I, int J, class T> class X { };
template<int I, int J> class X<I, J, int> { } // #1
template<int I> class X<I, I, int> { } // #2

template<int I, int J> void f(X<I, J, int>); // #A
template<int I> void f(X<I, I, int>); // #B
```

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

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();
};

template<class T, int I> void A<T,I>::f() {} 
```

```
// class template partial specialization
template<class T> struct A<T,2> {
    void f();
    void g();
    void h();
};
```
// member of class template partial specialization
// explicit specialization
template<>
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 ]

1 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:

    template<class T> struct A {
        template<class T2> struct B {}; // #1
        template<class T2> struct B<T2*> {}; // #2
    };

    template<> template<class T2> struct A<short>::B {}; // #3

    A<char>::B<int*> abcip; // uses #2
    A<short>::B<int*> absip; // uses #3
    A<char>::B<int> abci; // uses #1

— end example ]

14.5.6 Function templates

A function template defines an unbounded set of related functions. [ Example: a family of sort functions might be declared like this:

    template<class T> class Array { };  
    template<class T> void sort(Array<T>&);
A function template can be overloaded with other function templates and with normal (non-template) functions. A normal function is not related to a function template (i.e., it is never considered to be a specialization), even if it has the same name and type as a potentially generated function template specialization.\(^\text{132}\)

### 14.5.6.1 Function template overloading

It is possible to overload function templates so that two different function template specializations have the same type. [Example:

```c
// file1.c
template<class T>
void f(T*);
void g(int* p) {
    f(p); // call
    //f<int>(int*)
}

// file2.c
template<class T>
void f(T);
void h(int* p) {
    f(p); // call
    //f<int*>(int*)
}
```

Such specializations are distinct functions and do not violate the one definition rule (3.2).

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: two distinct function templates may have identical function return types and function parameter lists, even if overload resolution alone cannot distinguish them.

```c
template<class T> void f();
template<int I> void f(); // OK: overloads the first template
    // distinguishable with an explicit template argument list
```

When an expression that references a template parameter is used in the function parameter list or the return type in the declaration of a function template, the expression that references the template parameter is part of the signature of the function template. This is necessary to permit a declaration of a function template in one translation unit to be linked with another declaration of the function template in another translation unit and, conversely, to ensure that function templates that are intended to be distinct are not linked with one another. [Example:

```c
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
```

[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]

\(^{132}\) 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 used in a program, it must be defined; it will not be implicitly instantiated using the function template definition.
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. [Example:

```
template <int I, int J> void f(A<I+J>); // #1
template <int K, int L> void f(A<K+L>); // same as #1
```

— 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:

```
// Guaranteed to be the same
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+10>);

// Guaranteed to be different
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+11>);

// Ill-formed, no diagnostic required
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+1+2+3+4>);
```

— end note ]

### 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:

— during overload resolution for a call to a function template specialization (13.3.3);
— when the address of a function template specialization is taken;
— when a placement operator delete that is a function template specialization is selected to match a placement operator new (3.7.3.2, 5.3.4);
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 parameter types, or in the case of a conversion function the return 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 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.

Using the transformed function template's function parameter list, or in the case of a conversion function its transformed return type, perform type deduction against the function parameter list (or return type) of the other function. The mechanism for performing these deductions is given in 14.8.2.4.

Example:

```cpp
template<class T> struct A { A(); }

template<class T> void f(T);
template<class T> void f(T*);
template<class T> void f(const T*);

template<class T> void g(T);
template<class T> void g(T&);

template<class T> void h(const T&);
template<class T> void h(A<T>&);

void m() {
    const int *p;
    f(p); // f(const T*) is more specialized than f(T) or f(T*)
    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
}
```

Example:  

```cpp
template<class T> void f(T);  // #1  
template<class T> void f(T*, int=1);  // #2  
template<class T> void g(T);  // #3  
template<class T> void g(T*, ...);  // #4
```
int main() {
    int* ip;
    f(ip);    // calls #2
    g(ip);    // calls #4
}
— end example ]

14.5.7 Template aliases

A template alias declares a name for a family of types. The name of the template alias is a template-name.

When a template-id refers to the specialization of a template alias, it is equivalent to the associated type obtained by substitution of its template-argument s for the template-parameter s in the type-id of the template alias. [ Note: A template alias 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, allocator<int>>);
    g(v);    // OK: TT = vector
— end example ]

14.6 Name resolution

Three kinds of names can be used within a template definition:

— The name of the template itself, and names declared within the template itself.
— Names dependent on a template-parameter (14.6.2).
— 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:
Name resolution

class X;

template<class T> class Y {
    class Z;   // forward declaration of member class

    void f() {
        X* a1;   // declare pointer to X
        T* a2;   // declare pointer to T
        Y* a3;   // declare pointer to Y<T>
        Z* a4;   // declare pointer to Z
        typedef typename T::A TA;
        TA* a5;  // declare pointer to T's A
        typename T::A* a6; // declare pointer to T's A
        T::A* a7; // 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
    }
};

— end example ]

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 depends on a template-parameter (14.6.2), 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.

typename-specifier:
    typename ::opt nested-name-specifier identifier
    typename ::opt nested-name-specifier template::opt simple-template-id

If a specialization of a template is instantiated for a set of template-argument s 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:

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;

    Draft
325 Templates 14.6 Name resolution

f(b);  // OK: T::X refers to B::X
f(a);  // error: T::X refers to the data member A::X not the struct A::X
}

— end example ]

5 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. [ Note: the typename keyword is not permitted by the syntax of these constructs. — end note ]

6 Within the definition of a class template or within the definition of a member of a class template, the keyword typename is not required when referring to the unqualified name of a previously declared member of the class template that declares a type. [ Example:

```cpp
template<class T> struct A {
    typedef int B;
    B b;                  // OK, no typename required
};
```

— end example ]

7 Knowing which names are type names allows the syntax of every template definition to be checked. No diagnostic shall be issued for a template definition for which a valid specialization can be generated. If no valid specialization can be generated for a template definition, and that template is not instantiated, the template definition is ill-formed, no diagnostic required. If 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 done, and if the completeness of that type affects whether or not the program is well-formed or affects the semantics of the program, the program is ill-formed; no diagnostic is required. [ 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:

```cpp
int j;
template<class T> class X {
    // ...
    void f(T t, int i, char* p)
    {
        t = i;                        // diagnosed if X::f is instantiated
        // and the assignment to t is an error
        p = i;                        // may be diagnosed even if X::f is
        // not instantiated
        p = j;                        // may be diagnosed even if X::f is
        // not instantiated
    }
    void g(T t) {                    // may be diagnosed even if X::g is
        +;                          // not instantiated
    }
};

— end example ]

Draft
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:

```
#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-argument s 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:

```
#include <iostream>
using namespace std;

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
}

template<int i> void h()
{
    // ...
    // error: declaration for h not found
}
```

Draft
g(e);  // will cause one call of f(char) followed
        // by two calls of f(E)
g('a');  // will cause three calls of f(char)
}

— end example ]

[ Note: for purposes of name lookup, default arguments of function templates and default arguments 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 with or without a template-argument-list. When it is used without a template-argument-list, it is equivalent to the injected-class-name followed by the template-parameter s of the class template enclosed in <>. When it is used with a template-argument-list, it refers to the specified class template specialization, which could be the current specialization or another specialization.

2 Within the scope of a class template specialization or partial specialization, when the injected-class-name is not followed by a <, it is equivalent to the injected-class-name followed by the template-argument s of the class template specialization or partial specialization enclosed in <>. [ Example:

    template<class T> class Y;
    template<> class Y<int> {
        Y* p;        // meaning Y<int>
        Y<char>* q;  // meaning Y<char>
    };

— end example ]

3 The injected-class-name of a class template or class template specialization can be used either with or without a template-argument-list wherever it is in scope. [ Example:

    template <class T> struct Base {
        Base* p;
    };

    template <class T> struct Derived: public Base<T> {
        typename Derived::Base* p;  // meaning Derived::Base<T>
    };

— 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 followed by a template-argument-list, the reference refers to the class template itself and not a specialization thereof, and is not ambiguous. [ Example:

    template <class T> struct Base { };
    template <class T> struct Derived: Base<int>, Base<char> {

Draft
5 When the normal name of the template (i.e., the name from the enclosing scope, not the injected-class-name) is used without a \textit{template-argument-list}, it refers to the class template itself and not a specialization of the template. [\textit{Example:}]

\begin{verbatim}
template <class T> class X {
    X* p; // meaning X<T>
    X<T>* p2;
    X<int>* p3;
    ::X* p4; // error: missing template argument list
};
\end{verbatim}

— end example ]

6 The scope of a \textit{template-parameter} extends from its point of declaration until the end of its template. A \textit{template-parameter} hides any entity with the same name in the enclosing scope. [\textit{Note:} this implies that a \textit{template-parameter} can be used in the declaration of subsequent \textit{template-parameter}s and their default arguments but cannot be used in preceding \textit{template-parameter}s or their default arguments. For example,

\begin{verbatim}
template<class T, T* p, class U = T> class X { /* */
    template<class T> void f(T* p = new T);
};
\end{verbatim}

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

\begin{verbatim}
template<class T> class X : public Array<T> { /* */
    template<class T> class Y : public T { /* */
};
\end{verbatim}

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

7 A \textit{template-parameter} shall not be redeclared within its scope (including nested scopes). A \textit{template-parameter} shall not have the same name as the template name. [\textit{Example:}]

\begin{verbatim}
template<class T, int i> class Y {
    int T; // error: template-parameter redeclared
    void f() {
        char T; // error: template-parameter redeclared
    }
};
\end{verbatim}

\begin{verbatim}
template<class X> class X; // error: template-parameter redeclared
\end{verbatim}

— end example ]

8 In the definition of a member of a class template that appears outside of the class template definition, the name of a member of this template hides the name of a \textit{template-parameter}. [\textit{Example:}]

\begin{verbatim}

typename Derived::Base b; // error: ambiguous
typename Derived::Base<double> d; // OK

};
\end{verbatim}
template<class T> struct A {
    struct B { /* ... */};
    void f();
};

template<class B> void A<B>::f() {
    B b; // A's B, not the template parameter
}

— end example ]

9 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 ]

10 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 base class which does not depend on a template-parameter (14.6.2), if the name of the base
class or the name of a member of the base class is the same as the name of a template-parameter, the base class name
or member name hides the template-parameter name (3.3.8). [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]

1 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:
where the postfix-expression is an unqualified-id but not a template-id, the unqualified-id denotes a dependent name if and only if any of the expressions in the expression-list is a type-dependent expression (14.6.2.2). 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.

[Example:

template<class T> struct X : B<T> {
    typename T::A* pa;
    void f(B<T>* pb) {
        static int i = B<T>::i;
        pb->j++;
    }
};

the base class name B<T>, the type name T::A, the names B<T>::i and pb->j explicitly depend on the template-parameter. — end example]

In the definition of a class template or a member of a class template, if a base class of the class template depends on a template-parameter, the base class scope 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:

typedef double A;
template<class T> class B {
    typedef int A;
};
template<class T> struct X : B<T> {
    A a; // a has type double
};

The type name A in the definition of X<T> binds to the typedef name defined in the global namespace scope, not to the typedef name defined in the base class B<T>. — end example] [Example:

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;

Draft
The members A::B, A::a, and A::Y of the template argument A do not affect the binding of names in Y<A>. — end example]

14.6.2.1 Dependent types

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, a name refers to the current instantiation if it is

— the injected-class-name (9) of the class template or nested class,

— in the definition 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 <>,

— 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

— in the definition of a partial specialization, the name of the class template followed by the template argument list of the partial specialization enclosed in <>. 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.

The template argument list of a primary template is a template argument list in which the nth template parameter has the value of the nth template parameter of the class template. If the nth template parameter is a template parameter pack, the nth template argument is a pack expansion (14.5.3) whose pattern is the name of the template parameter pack.

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:

template <class T> class A {
  A* p1; // A is the current instantiation
  A<T>* p2; // A<T> is the current instantiation
  A<T*> p3; // A<T*> is not the current instantiation
  ::A<T>* p4; // ::A<T> is the current instantiation
  class B {
    B* p1; // B is the current instantiation
    A<T>::B* p2; // A<T>::B is the current instantiation
    typename A<T>::::B* p3, // A<T>::::B is not the
    // current instantiation
  };
};

template <class T> class A<T*> {
  A<T>* p1; // A<T> is the current instantiation
  A<T>* p2; // A<T> is not the current instantiation
};

template <class T1, class T2, int I> struct B {
  B<T1, T2, I>* b1; // refers to the current instantiation
  B<T2, T1, I>* b2; // not the current instantiation
  typedef T1 my_T1;
  static const int my_I = I;
};
static const int my_I2 = I+0;
static const int my_I3 = my_I;
B<my_T1, T2, my_I2>* b3;    // refers to the current instantiation
B<my_T1, T2, my_I2>* b4;    // not the current instantiation
B<my_T1, T2, my_I3>* b5;    // refers to the current instantiation
}
— end example ]

4 A name is a member of the current instantiation if it is
— An unqualified name that, when looked up, refers to a member of a class template. [ Note: this can only occur when looking up a name in a scope enclosed by the definition of a class template. — end note ]
— A qualified-id in which the nested-name-specifier refers to the current instantiation.
— [ 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 ]

5 A name is a member of an unknown specialization if the name is a qualified-id in which the nested-name-specifier names a dependent type that is not the current instantiation.

6 A type is dependent if it is
— a template parameter,
— a member of an unknown specialization,
— a nested class that is a member of the current instantiation,
— a cv-qualified type where the cv-unqualified type is dependent,
— a compound type constructed from any dependent type,
— an array type constructed from any dependent type or whose size is specified by a constant expression that is value-dependent, or
— 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.
— denoted by decltype(expression), where expression is type-dependent (14.6.2.2).
[Note: because typedefs do not introduce new types, but instead simply refer to other types, a name that refers to a typedef that is a member of the current instantiation is dependent only if the type referred to is dependent. — end note]

### 14.6.2.2 Type-dependent expressions [temp.dep.expr]

Except as described below, an expression is type-dependent if any subexpression is type-dependent.

This is type-dependent if the class type of the enclosing member function is dependent (14.6.2.1).

An id-expression is type-dependent if it contains:

- an identifier that was declared with a dependent type,
- a template-id that is dependent,
- a conversion-function-id that specifies a dependent type,
- a nested-name-specifier or a qualified-id that names a member of an unknown specialization.

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:

```plaintext
simple-type-specifier ( expression-list_opt )
::opt new new-placement_opt new-type-id new-initializer_opt
::opt new new-placement_opt ( type-id ) new-initializer_opt
dynamic_cast < type-id > ( expression )
static_cast < type-id > ( expression )
const_cast < type-id > ( expression )
reinterpret_cast < type-id > ( expression )
(type-id) cast-expression
```

Expressions of the following forms are never type-dependent (because the type of the expression cannot be dependent):

```plaintext
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
```

[Note: For the standard library macro offsetof, see 18.1. — end note]

A class member access expression (5.2.5) is type-dependent if the type of the referenced member is dependent. [Note: in an expression of the form x.y or xp->y the type of the expression is usually the type of the member y of the class of x (or the class pointed to by xp). However, if x or xp refers to a dependent type that is not the current instantiation, the type of y is always dependent. If x or xp refers to a non-dependent type or refers to the current instantiation, the type of y is the type of the class member access expression. — end note]
14.6 Name resolution

14.6.2.3 Value-dependent expressions

Except as described below, a constant expression is value-dependent if any subexpression is value-dependent.

1 An identifier is value-dependent if it is:
   — a name declared with a dependent type,
   — the name of a non-type template parameter,
   — a constant with integral or enumeration literal type and is initialized with an expression that is value-dependent.

Expressions of the following form are value-dependent if the unary-expression is type-dependent or the type-id is dependent (even if sizeof unary-expression and sizeof (type-id) are not type-dependent):

\[
\begin{align*}
&\text{sizeof \ unary-expression} \\
&\text{sizeof \ (type-id)} \\
&\text{alignof \ (type-id)}
\end{align*}
\]

[Note: For the standard library macro offsetof, see 18.1. — end note]

2 Expressions of the following form are value-dependent if either the type-id or simple-type-specifier is dependent or the expression or cast-expression is value-dependent:

\[
\begin{align*}
&\text{simple-type-specifier \ (expression-list\opt)} \\
&\text{static\_cast \ <\ type-id> \ (expression)} \\
&\text{const\_cast \ <\ type-id> \ (expression)} \\
&\text{reinterpret\_cast \ <\ type-id> \ (expression)} \\
&\text{(type-id) \ cast-expression}
\end{align*}
\]

3 Expressions of the following form are value-dependent:

\[
\begin{align*}
&\text{sizeof \ ... \ (identifier)}
\end{align*}
\]

14.6.2.4 Dependent template arguments

1 A type template-argument is dependent if the type it specifies is dependent.

2 An integral non-type template-argument is dependent if the constant expression it specifies is value-dependent.

3 A non-integral non-type template-argument is dependent if its type is dependent or it has either of the following forms

\[
\begin{align*}
&\text{qualified-id} \\
&\&\text{qualified-id}
\end{align*}
\]

and contains a nested-name-specifier which specifies a class-name that names a dependent type.

4 A template template-argument is dependent if it names a template-parameter or is a qualified-id with a nested-name-specifier which contains a class-name that names a dependent type.

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:
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

1 In resolving dependent names, names from the following sources are considered:
   
   — Declarations that are visible at the point of definition of the template.
   
   — 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

1 For a function template specialization, a member function template specialization, or a specialization for a member function or static data member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization and the context from which it is referenced depends on a template parameter, the point of instantiation of the specialization is the point of instantiation of the enclosing specialization. Otherwise, the point of instantiation for such a specialization immediately follows the namespace scope declaration or definition that refers to the specialization.

2 If a function template or member function of a class template is called in a way which uses the definition of a default argument of that function template or member function, the point of instantiation of the default argument is the point of instantiation of the function template or member function specialization.

3 For a 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.

4 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. 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 Candidate functions

For a function call that depends on a template parameter, if the function name is an unqualified-id but not a template-id, or if the function is called using operator notation, the candidate functions are found using the usual lookup rules (3.4.1, 3.4.2) except that:

— For the part of the lookup using unqualified name lookup (3.4.1), only function declarations with external linkage from the template definition context are found.

— For the part of the lookup using associated namespaces (3.4.2), only function declarations with external linkage 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 Friend names declared within a class template

Friend classes or functions can be declared within a class template. When a template is instantiated, the names of its friends are treated as if the specialization had been explicitly declared at its point of instantiation.

As with non-template classes, the names of namespace-scope friend functions of a class template specialization are not visible during an ordinary lookup unless explicitly declared at namespace scope (11.4). Such names may be found under the rules for associated classes (3.4.2).[[133]]

```cpp
template<typename T> class number {
public:
    number(int);
    // ...
    friend number gcd(number x, number y) { return 0; }
private:
    // ...
};

void g()
{
```

---

[[133]] Friend declarations do not introduce new names into any scope, either when the template is declared or when it is instantiated.
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, 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 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:

```
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).
For a given template and a given set of template-argument s,

— 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.

Each class template specialization instantiated from a template has its own copy of any static members. [Example:

```
template<class T> class X {
    static T s;
    // ...
};
template<class T> T X<T>::s = 0;
X<int> aa;
X<char*> bb;
```

X<int> has a static member s of type int and X<char*> has a static member s of type char*. — end example]

### 14.7.1 Implicit instantiation

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. The implicit instantiation of a class template specialization causes the implicit instantiation of the declarations, but not of the definitions or default arguments, of the class member functions, member classes, static data members and member templates; and it causes the implicit instantiation of the definitions of member anonymous unions. 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:

```
template<class T> class Z {
public:
    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]

4 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:
```
template<class T> class B { /* ... */};
template<class T> class D : public B<T> { /* ... */};

void f(void*);
void f(B<int>*);

void g(D<int>* p, D<char>* pp, D<double>* ppp)
{
    f(p); // instantiation of D<int> required: call f(B<int>*)

    B<char>* q = pp; // instantiation of D<char> required:
    // convert D<char>* to B<char>*

    delete ppp; // instantiation of D<double> required
}
```
— end example ]

5 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:
```
template <class T> struct S {
    operator int();
};

void f(int);
void f(S<int>&);
void f(S<float>);
```

void g(S<int>& sr) {
    f(sr); // instantiation of S<int> allowed but not required
    // instantiation of S<float> allowed but not required
};

— end example ]

If an implicit instantiation of a class template specialization is required and the template is declared but not defined, the
program is ill-formed. [ Example:

    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 member template, a non-virtual member func-
tion, 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 and function template specializations are placed in the namespace where the template is
defined. Implicitly instantiated specializations for members of a class template are placed in the namespace where the
enclosing class template is defined. Implicitly instantiated member templates are placed in the namespace where the
enclosing class or class template is defined. [ Example:

    namespace N {
        template<class T> class List {
        public:
            T* get();
            // ...
        };}

    template<class K, class V> class Map {
        N::List<V> lt;
        V get(K);
        // ...
    };

    void g(Map<char*,int>& m)
    {

Draft
int i = m.get("Nicholas");
   // ...
}

A call of \texttt{lt.get()} from \texttt{Map<char*,\text{int}>::get()} would place \texttt{List<int>::get()} in the namespace \texttt{N} rather than in the global namespace. — end example

If a function template \texttt{f} is called in a way that requires a default argument expression 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 expression is done as if the default argument expression had been an expression 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. This analysis is called \textit{default argument instantiation}. The instantiated default argument is then used as the argument of \texttt{f}.

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 = zdef(T())} instantiated
   f(a);       // ill-formed; \texttt{ydef} is not declared
}
\end{verbatim}

— end example]

[\textit{Note: 14.6.4.1 defines the point of instantiation of a template specialization. — end note}]

There is an implementation-defined quantity that specifies the limit on the total depth of recursive instantiations, 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]

\subsection*{14.7.2 Explicit instantiation} [temp.explicit]

A class, a function 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. \textit{An explicit instantiation of a function template shall not use the \texttt{inline} or \texttt{constexpr} specifiers.}

The syntax for explicit instantiation is:

\begin{verbatim}

Draft
\end{verbatim}
There are two forms of explicit instantiation: an explicit instantiation definition and an explicit instantiation declaration. An explicit instantiation declaration begins with the `extern` keyword.

If the explicit instantiation is for a class or member class, the `elaborated-type-specifier` in the `declaration` shall include a `simple-template-id`. 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`. 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. [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 ]

A declaration of a function template shall be in scope at the point of the explicit instantiation of the function template. A definition of the class or class template containing a member function template shall be in scope at the point of the explicit instantiation of the member function template. A definition of a class template or class member template shall be in scope at the point of the explicit instantiation of the class template or class member template. A definition of a class template shall be in scope at the point of an explicit instantiation of a member function or a static data member of the class template. A definition of a member class of a class template shall be in scope at the point of an explicit instantiation of the member class. If the `declaration` of the explicit instantiation names an implicitly-declared special member function (clause 12), the program is ill-formed.

For a given set of template parameters, 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 non-exported function template, a non-exported member function template, or a non-exported member function or static data member of a class template shall be present in every translation unit in which it is explicitly instantiated.

An explicit instantiation of a class or function template specialization is placed in the namespace in which the template is defined. An explicit instantiation for a member of a class template is placed in the namespace where the enclosing class template is defined. An explicit instantiation for a member template is placed in the namespace where the enclosing class or class template is defined. [Example:

```cpp
namespace N {

Draft
template<class T> class Y { void mf() { } ; }

template class Y<int>; // error: class template Y not visible
// in the global namespace

using N::Y;
template class Y<int>; // OK: explicit instantiation in namespace N

template class N::Y<char*>; // OK: explicit instantiation in namespace N
template void N::Y<double>::mf(); // OK: explicit instantiation
// in namespace N

— end example ]

A trailing template-argument can be left unspecified in an explicit instantiation of a function template specialization or of a member function template specialization provided it can be deduced from the type of a function parameter (14.8.2). [ Example:

template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v);

// instantiate sort(Array<int>&) - template-argument deduced
template void sort<>(Array<int>&);

— end example ]

An explicit instantiation that names a class template specialization is an explicit instantiation of the same kind (declaration or definition) of each of its members (not including members inherited from base classes) that has not been previously explicitly specialized in the translation unit containing the explicit instantiation, except as described below.

An explicit instantiation definition that names a class template specialization explicitly instantiates the class template specialization and is only an explicit instantiation definition of members whose definition is visible at the point of instantiation.

An explicit instantiation declaration that names a class template specialization has no effect on the class template specialization itself (except for perhaps resulting in its implicit instantiation). Except for inline functions, other 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 used so that the body can be considered for inlining, but that no out-of-line copy of the inline function would be generated in the translation unit. — end note ]

If an entity is the subject of both an explicit instantiation declaration and an explicit instantiation definition in the same translation unit, the definition shall follow the declaration. An entity that is the subject of an explicit instantiation declaration and that is also used in the translation unit shall be the subject of an explicit instantiation definition somewhere in the program; otherwise the program is ill-formed, no diagnostic required. [ Note: This rule does apply to inline functions even though an explicit instantiation declaration of such an entity has no other normative effect. This is needed to ensure that if the address of an inline function is taken in a translation unit in which the implementation chose to suppress the out-of-line body, another translation unit will supply the body. — end note ] An explicit instantiation declaration shall not name a specialization of a template with internal linkage.

Draft
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]

An explicit instantiation does not constitute a use of a default argument, so default argument instantiation is not done. [Example:

```cpp
char* p = 0;
template<class T> T g(T = &p);
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:

- non-deleted function template
- class template
- non-deleted member function of a class template
- static data member of a class template
- member class of a class template
- member class template of a class template
- non-deleted member function template of a class template

can be declared by a declaration introduced by `template<>`; that is:

```
explicit-specialization:
    template <> declaration
```

[Example:

```cpp
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]

An explicit specialization shall be declared in the namespace of which the template is a member, or, for member templates, in the namespace of which the enclosing class or enclosing class template is a member. An explicit specialization
of a member function, member class or static data member of a class template shall be declared in the namespace of which the class template is a member. 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 or class template being explicitly specialized shall be in scope at the point of declaration of an 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 be in scope at the point of 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 class or a static data member 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 be in scope at the point of declaration of 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. In this case, the definition of the class template explicit specialization shall be in scope at the point of declaration of the explicit specialization of the member. 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. Definitions of members of an explicitly specialized class are defined in the same manner as members of normal classes, and not using the syntax for explicit specialization. [Example:

```cpp
template<class T> struct A {
    void f(T) { /* ... */ }
};

template<> struct A<int> {
    void f(int);
};

void h()
{
    A<int> a;
    a.f(16); // A<int>::f must be defined somewhere
}

// explicit specialization syntax not used for a member of
// explicitly specialized class template specialization
void A<int>::f(int) { /* ... */ }

—end example]
```
If a template, a member template or the 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:

```cpp
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
```
—end example]

The placement of explicit specialization declarations for function templates, class templates, member functions of class templates, static data members of class templates, member classes of class templates, member class templates of class templates, member function templates of class templates, member functions of member templates of non-template classes, member function templates of member classes of class templates, etc., and the placement of partial specialization declarations of class templates, member class 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.

When a specialization for which an explicit specialization exists is used within the instantiation of an exported template, and the unspecialized template name is non-dependent in the exported template, a declaration of the explicit specialization shall be declared before the definition of the exported template, in the translation unit containing that definition. [Example:

```cpp
// file #1
#include <vector>
// Primary class template vector
export template<class T> void f(t) {
  std::vector<T> vec; // should match the specialization
  /* ... */
}

// file #2
#include <vector>
class B { }
// Explicit specialization of vector for vector<B>
namespace std {
```
template<> class vector<B> { /* ... */ };  
}
template<class T> void f(T);
void g(B b) {
f(b);
  // ill-formed:
  // f<B> should refer to vector<B>, but the
  // specialization was not declared with the
  // definition of f in file #l
}

— end example |

9 A template explicit specialization is in the scope of the namespace in which the template was defined. [Example:

namespace N {
template<class T> class X { /* ... */ };  
template<class T> class Y { /* ... */ };  

template<> class X<int> { /* ... */ };  // OK: specialization
  // in same namespace  
template<> class Y<double>;  // forward declare intent to
  // specialize for double
}
template<> class N::Y<double> { /* ... */ };  // OK: specialization
  // in same namespace

— end example ]

10 A simple-template-id that names a class template explicit specialization that has been declared but not defined can be
used exactly like the names of other incompletely-defined classes (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 ]

11 A trailing template-argument can be left unspecified in the template-id naming an explicit function template specialization
provided it can be deduced from the function argument type. [Example:

template<class T> class Array { /* ... */ };  
template<class T> void sort(Array<T>& v);

// explicit specialization for sort(Array<int>&)
// with deduces template-argument of type int
template<> void sort(Array<int>&);

— end example ]
12 [Note: This paragraph is intentionally empty. — end note]

13 A function with the same name as a template and a type that exactly matches that of a template specialization is not an explicit specialization (14.5.6).

14 An explicit specialization of a function template is inline only if it is explicitly declared to be, and independently of whether its function template is. [Example:

```cpp
template<class T> void f(T) { /* ... */ }
template<class T> inline T g(T) { /* ... */ }

template<> inline void f<int>(int) { /* ... */ } // OK: inline
template<> int g<int>(int) { /* ... */ } // OK: not inline
```
— end example]

15 An explicit specialization of a static data member of a template is a definition if the declaration includes an initializer; otherwise, it is a declaration. [Note: there is no syntax for the definition of a static data member of a template that requires default initialization.

```cpp
template<> X Q<int>::x;
```

16 This is a declaration regardless of whether X can be default initialized (8.5). — end note]

17 A member or a member template of a class template may be explicitly specialized for a given implicit instantiation of the class template, even if the member or member template is defined in the class template definition. An explicit specialization of a member or member template is specified using the syntax for explicit specialization. [Example:

```cpp
template<class T> struct A {
    void f(T);
    template<class X1> void g1(T, X1);
    template<class X2> void g2(T, X2);
    void h(T) { }
};

// specialization
template<> void A<int>::f(int);

// out of class member template definition
template<class T> template<class X1> void A<T>:::g1(T, X1) { }

// member template specialization
template<> template<class X1> void A<int>:::g1(int, X1);

//member template specialization

template<> template<>
    void A<int>:::g1(int, char); //X1 deduced as char

template<> template<>
    void A<int>:::g2<char>(int, char); //X2 specified as char

// member specialization even if defined in class definition

template<> void A<int>:::h(int) { }
```

Draft
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() { } // ill-formed; its enclosing class template A is not

— end example ]

A specialization of a member function template or member class 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:

— the explicit specialization of a function template;

Draft
14.8 Function template specializations

— the explicit specialization of a member function template;

— 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:

```cpp
template<class T> void f(T* p) {
    static T s;
    // ...
}

void g(int a, char* b) {
    f(&a); // call f<int>(int*)
    f(&b); // call 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-argument` s in the same way as `template-argument` s are specified in uses of a class template specialization. [Example:

```cpp
template<class T> void sort(Array<T>& v);
void f(Array<dcomplex>& cv, Array<int>& ci) {
    sort<dcomplex>(cv); // sort(Array<dcomplex>&)
    sort<int>(ci);      // sort(Array<int>&)
}
```

and

```cpp
template<class U, class V> U convert(V v);
void g(double d) {
    int i = convert<int,double>(d);    // int convert(double)
```
char c = convert<char,double>(d); // char convert(double)
}

— end example ]

2 A template argument list may be specified when referring to a specialization of a function template
— when a function is called,
— 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,
— in an explicit specialization,
— in an explicit instantiation, or
— in a friend declaration.

3 Trailing template arguments that can be deduced (14.8.2) or obtained from default template-argument s may be omitted
from the list of explicit template-argument s. A trailing template parameter pack 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; Z is deduced to be an empty sequence
    int j = f(5.6); // ill-formed: X cannot be deduced
    f<void>(f<int, bool>); // Y for outer f deduced to be
                          // int (*)(bool); Z is deduced to be an empty sequence
    f<void>(f<int>); // ill-formed: f<int> does not denote a
                    // single function template specialization
    int k = g<int>(5.6); // Y is deduced to be double, Z is deduced to an empty sequence
    f<void>(g<int, bool>); // Y for outer f is deduced to be
                            // int (*)(bool), Z is deduced to an empty sequence
}
— end example ]
```

4 [ Note: An empty template argument list can be used to indicate that a given use refers to a specialization of a function
template even when a normal (i.e., non-template) function 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<int>(1); // uses #1
— end note ]
```
Template arguments that are present shall be specified in the declaration order of their corresponding template-parameter.

```

- Example:

  template<class X, class Y, class Z> X f(Y,Z);
  template<class ... Args> void f2();

  void g()
  {
    f<int,char*,double>("aa",3.0);
    f<int,char*>("aa",3.0); // Z is deduced to be double
    f<int>("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-parameter that participate in template argument deduction. [Note: template parameters do not participate in template argument deduction if they are explicitly specified. For example,]

```

- 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);
```

Draft
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 ]

9 Template argument deduction can extend the sequence of template arguments corresponding to a template parameter pack, even when the sequence contains explicitly specified template arguments. [Example:

```cpp
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 [temp.deduct]

1 When a function template specialization is referenced, all of the template arguments must have values. The values can be explicitly specified or, in some cases, be deduced from the use or obtained from default template-argument s. [Example:

```cpp
void f(Array<dcomplex>& cv, Array<int>& ci)
{
    sort(cv);   // call sort(Array<dcomplex>&)
    sort(ci);   // call sort(Array<int>&)
}
```

and

```cpp
void g(double d)
{
    int i = convert<int>(d);       // call convert<int,double>(double)
    int c = convert<char>(d);     // call 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:

— 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. Otherwise, type deduction fails.

— 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.

— All references in the function type of the function template to the corresponding template parameters are replaced by the specified template argument values. If a substitution in a template parameter or in the function type of the function template results in an invalid type, type deduction fails. [Note: The equivalent substitution in exception specifications is done only when the function is instantiated, at which point a program is ill-formed if the substitution results in an invalid type.] Type deduction may fail for the following reasons:

— Attempting to create an array with an element type that is `void`, a function type, a reference type, or an abstract class type, or attempting to create an array with a size that is zero or negative. [Example:

```
template <class T> int f(T[5]);
int i = f<int>(0);
int j = f<void>(0);  // invalid array
```
— end example]

— Attempting to use a type that is not a class type in a qualified name. [Example:

```
template <class T> int f(typename T::B*);
int i = f<int>(0);
```
— end example]

— Attempting to use a type in a nested-name-specifier of a qualified-id when that type does not contain the specified member, or

— the specified member is not a type where a type is required, or

— the specified member is not a template where a template is required, or

— the specified member is not a non-type where a non-type is required.

[Example:

```
template <int I> struct X { };  
template <template <class T> class> struct Z { };  
template <class T> void f(typename T::Y*){}  
template <class T> void g(X<T::N>*){}  
template <class T> void h(Z<T::template TT>*){}  
struct A {};  
struct B { int Y; };  
struct C {  
typedef int N;
```
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 B member of C is not a non-type
    h<D>(0); // The TT member of D is not a template
}

— end example ]

— Attempting to create a pointer to reference type.
— Attempting to create a reference to void.
— Attempting to create “pointer to member of T” when T is not a class type. [ Example:
    template <class T> int f(int T::*);
    int i = f<int>(0);
    — end example ]

— Attempting to give an invalid type to a non-type template parameter. [ Example:
    template <class T, T> struct S {}
    template <class T> int f(S<T, T>()*);
    struct X {}
    int i0 = f<X>(0);
    — end example ]

— Attempting to perform an invalid conversion in either a template argument expression, or an expression used in the function declaration. [ Example:
    template <class T, T*> int f(int);
    int i2 = f<int, 1>(0); // can’t conv 1 to int*
    — end example ]

— Attempting to create a function type in which a parameter has a type of void, or in which the return type is a function type or array type.

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:

Draft
template <class T> void f(T t);
template <class X> void g(const X x);
template <class Z> void h(Z, Z*);

int main()
{
    // #1: function type is f(int), t is non const
    f<int>(1);

    // #2: function type is f(int), t is const
    f<const int>(1);

    // #3: function type is g(int), x is const
    g<int>(1);

    // #4: function type is g(int), x is const
    g<const int>(1);

    // #5: function type is h(int, const int*)
    h<const int>(1,0);
}

— end example ]

[ Note: f<int>(1) and f<const int>(1) call distinct functions even though both of the functions called have the same function type. — end note ]

The resulting substituted and adjusted function type is used as the type of the function template for template argument deduction. If a template argument has not been deduced, its default template argument, if any, is used. [ Example:

template <class T, class U = double>
void f(T t = 0, U u = 0);

void g()
{
    f(1, 'c'); // f<int, char>(1,'c')
    f(1); // f<int, double>(1,0)
    f(); // error: T cannot be deduced
    f<int>(); // f<int, double>(0,0)
    f<int, char>(); // f<int, char>(0,0)
}

— end example ]

When all template arguments have been deduced or obtained from default template arguments, all uses of template parameters in non-deduced contexts are replaced with the corresponding deduced or default argument values. If the substitution results in an invalid type, as described above, type deduction fails.

Except as described above, the use of an invalid value shall not cause type deduction to fail. [ Example: In the following example 1000 is converted to signed char and results in an implementation-defined value as specified in (4.7). In other
words, both templates are considered even though 1000, when converted to signed char, results in an implementation-defined value.

```cpp
template <int> int f(int);
template <signed char> int f(int);
int i1 = f<1>(0);     // ambiguous
int i2 = f<1000>(0); // ambiguous
```

— 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. For a function parameter pack, the type $A$ of each remaining argument of the call 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. [ Note: A function parameter pack can only occur at the end of a parameter-declaration-list (8.3.5). — end note ] [ Example:

```cpp
template<class ... Types> void f(Types& ...);
template<class T1, class ... Types> void g(T1, Types ...);
void h(int x, float& y)
{
    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
}
```

— end example ]

1 If $P$ is not a reference type:

   — 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,

   — 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,

   — If $A$ is a cv-qualified type, the top level cv-qualifiers of $A$’s type are ignored for type deduction.

2 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. If $P$ is an rvalue reference type and the argument is an lvalue, the type $A$ is used in place of $A$ for type deduction. [ Example:

```cpp
template <typename T> int f(T&&);
int i;
int j = f(i);    // calls f<int&&>(i)
```

— end example ] [ Note: The effect of this rule for lvalue arguments and rvalue reference parameters is that deduction in such cases will fail unless the function parameter is of the form cv T&& (14.8.2.5). — end note ]
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:

- 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 \( A \).
- 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).
- If \( P \) is a class and \( P \) has the form `simple-template-id`, then the transformed \( A \) can be a derived class of the deduced \( A \). Likewise, if \( P \) is a pointer to a class of the form `simple-template-id`, the transformed \( A \) can be a pointer to a derived class pointed to by the deduced \( A \).

These alternatives are considered only if type deduction would otherwise fail. If they yield more than one possible deduced \( A \), the type deduction fails. [Note: if a `template-parameter` is not used in any of the function parameters of a function template, or is used only in a non-deduced context, its corresponding `template-argument` cannot be deduced from a function call and the `template-argument` must be explicitly specified. — end note ]

When \( P \) is a function type, pointer to function type, or pointer to member function type:

- If the argument is an overload set containing one or more function templates, the parameter is treated as a non-deduced context.
- If the argument is an overload set (not containing function templates), trial argument deduction is attempted using each of the members of the set. If deduction succeeds for only one of the overload set members, that member is used as the argument value for the deduction. If deduction succeeds for more than one member of the overload set the parameter is treated as a non-deduced context.

[Example:

```cpp
// Only one function of an overload set matches the call so the function
// parameter is a deduced context.
-template <class T> int f(T (*p)(T));
-int g(int);
-int g(char);
-int i = f(g);  // calls f(int (*)(int))
```

— end example ]

[Example:

```cpp
// Ambiguous deduction causes the second function parameter to be a
// non-deduced context.
-template <class T> int f(T, T (*p)(T));
-int g(int);
-char g(char);
-int i = f(1, g);  // calls f(int, int (*)(int))
```

— end example ]
// The overload set contains a template, causing the second function // parameter to be a non-deduced context.

```cpp
template <class T> int f(T, T (*p)(T));
char g(char);
template <class T> T g(T);
int i = f(1, g);  // calls f(int, int (*)(int))
```

--- end example

## 14.8.2.2 Deducing template arguments taking the address of a function template

Template arguments can be deduced from the type specified when taking the address of an overloaded function (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.

### 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) as described in 14.8.2.5.

**1.** 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.

**2.** 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. If P is a reference type, the type referred to by P is used for type deduction.

**3.** 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)
- The deduced A can be another pointer or pointer to member type that can be converted to A via a qualification conversion.

**4.** These alternatives are considered only if type deduction would otherwise fail. If they yield more than one possible deduced A, the type deduction fails.

**5.** 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

\[ cv_{1,0} \text{ “pointer to ...” } cv_{1,n-1} \text{ “pointer to” } cv_{1,n} T \]
and P is a type

\( \text{cv}_2,0 \) “pointer to ...” \( \text{cv}_{2,n-1} \) “pointer to” \( \text{cv}_{2,n} \) T2

The cv-unqualified T1 and T2 are used as the types of A and P respectively for type deduction. [Example:

```c
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:

- In the context of a function call, the function parameter types are used.
- In the context of a call to a conversion operator, the return types of the conversion function templates are used.
- In other contexts (14.5.6.2) the function template’s function type is used.

Each type from the parameter template and the corresponding type from the argument template are used as the types of P and A.

Before the partial ordering is done, certain transformations are performed on the types used for partial ordering:

- If P is a reference type, P is replaced by the type referred to.
- If A is a reference type, A is replaced by the type referred to.

If both P and A were reference types (before being replaced with the type referred to above), determine which of the two types (if any) is more cv-qualified than the other; otherwise the types are considered to be equally cv-qualified for partial ordering purposes. The result of this determination will be used below.

Remove any top-level cv-qualifiers:

- If P is a cv-qualified type, P is replaced by the cv-unqualified version of P.
- If A is a cv-qualified type, A is replaced by the cv-unqualified version of A.

Draft
Using the resulting types P and A the deduction is then done as described in 14.8.2.5. 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.

If, for a given type, deduction succeeds in both directions (i.e., the types are identical after the transformations above) and if the type from the argument template is more cv-qualified than the type from the parameter template (as described above) that type is considered to be more specialized than the other. If neither type is more cv-qualified than the other then neither type is more specialized than the other.

If for each type being considered a given template is at least as specialized for all types and more specialized for some set of types and the other template is not more specialized for any types or is not at least as specialized for any types, then the given template is more specialized than the other template. Otherwise, neither template is more specialized than the other.

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:

```cpp
template <class T> T f(int); // #1
template <class T, class U> T f(U); // #2
void g() {
    f<int>(1); // Calls #1
}
```

— end example ]

[Note: Partial ordering of function templates containing template parameter packs is independent of the number of deduced arguments for those template parameter packs. — end note] [Example:

```cpp
template<class ...> struct Tuple { };
template<class ... Types> void g(Tuple<Types ...>); // #1
template<class T1, class ... Types> void g(Tuple<T1, Types ...>); // #2
template<class T1, class ... Types> void g(Tuple<T1, Types& ...>); // #3

g(Tuple<>()); // calls #1
g(Tuple<int, float>()); // calls #2
g(Tuple<int, float&>()); // calls #3
g(Tuple<int>()); // calls #3
```

— end example ]

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:

- A function type includes the types of each of the function parameters and the return type.
- A pointer to member type includes the type of the class object pointed to and the type of the member pointed to.
- A type that is a specialization of a class template (e.g., \( A\langle\text{int}\rangle \)) includes the types, templates, and non-type values referenced by the template argument list of the specialization.
- An array type includes the array element type and the value of the array bound.

In most cases, the types, templates, and non-type values that are used to compose \( P \) participate in template argument deduction. That is, they may be used to determine the value of a template argument, and 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:

- The `nested-name-specifier` of a type that was specified using a `qualified-id`.
- A non-type template argument or an array bound in which a subexpression references a template parameter.
- 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.
- 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:
  - more than one function matches the function parameter type (resulting in an ambiguous deduction), or
  - no function matches the function parameter type, or
  - the set of functions supplied as an argument contains one or more function templates.

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 \( A\langle T\rangle::B\langle T2\rangle \), both \( T \) and \( T2 \) are non-deduced. Likewise, if a type is specified as \( A\langle I+J\rangle::X\langle T\rangle \), \( I \), \( J \), and \( T \) are non-deduced. If a type is specified as \( \text{void } f(\text{typename } A\langle T\rangle::B, A\langle T\rangle) \), the \( T \) in \( A\langle T\rangle::B \) is non-deduced but the \( T \) in \( A\langle T\rangle \) is deduced. — end example ]

[Example: Here is an example in which different parameter/argument pairs produce inconsistent template argument deductions:

```c
template<class T> void f(T x, T y) { /* ... */ }
struct A { /* ... */ };
struct B : A { /* ... */ };
void g(A a, B b)
{
    f(a,b); // error: T could be A or B
}
```

Draft
Here is an example where two template arguments are deduced from a single function parameter/argument pair. This can lead to conflicts that cause type deduction to fail:

```cpp
template <class T, class U> void f( T (*)( T, U, U ) );

int g1( int, float, float);
char g2( int, float, float);
int g3( int, char, float);

void r()
{
  f(g1); // OK: T is int and U is float
  f(g2); // error: T could be char or int
  f(g3); // error: U could be char or float
}
```

Here is an example where a qualification conversion applies between the argument type on the function call and the deduced template argument type:

```cpp
template<class T> void f(const T*) {}

int *p;

void s()
{
  f(p); //calls f(const int*)
}
```

Here is an example where the template argument is used to instantiate a derived class type of the corresponding function parameter type:

```cpp
template <class T> struct B { };  
template <class T> struct D : public B&lt;T&gt; { };  
struct D2 : public B&lt;int&gt; { };  
template &lt;class T&gt; void f(B&lt;T&gt;*){}

void t()
{
  D&lt;int&gt; d;
  D2 d2;
  f(d); // calls f(B&lt;int&gt;*)
  f(d2); // calls f(B&lt;int&gt;*)
}
```

8 A template type argument T, a template template argument TT or a template non-type argument i can be deduced if P and A have one of the following forms:
14.8 Function template specializations

Templates

Function template specializations

- cv-list
- T
- Tk
- T&
- T&&
- T[
- T[integer-constant]
- template-name<T> (where template-name refers to a class template)
- type(T)
- T()
- T(T)
- T type::*
- type T::*
- T (type::*())
- type (T::*())
- type (type::*)(T)
- type (T::*)(T)
- T (type::*)(T)
- T (T::*())
- T (T::*)(T)
- type[i]
- template-name<i> (where template-name refers to a class template)
- TT<T>
- TT<i>
- TT<>

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.

9 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.

10 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 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. [Note: A function parameter pack can only occur at the end of a parameter-declaration-list (8.3.5). — end note]

11 These forms can be used in the same way as T is for further composition of types. [Example:

X<int> (*)(char[6])

is of the form

Draft
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:

   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 ]

14 [ 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
      f1<20>(v); // OK
      f2(v); // error: cannot deduce template-argument i
      f2<10>(v); // OK
      f3(v); // OK: i deduced to be 10
   }

— end example ]

15 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
   }

Draft
— end example ] — end note ] [ Note: template parameters do not participate in template argument deduction if they are used only in non-deduced contexts. For example,

```cpp
    template<int i, typename T>
    T deduce(typename A<T>::X x, // T is not deduced here
              T t, // but T is deduced here
typename B<i>::Y y); // i is not deduced here
A<int> a;
B<77> b;
```

```cpp
int x = deduce<77>(a.xm, 62, y.ym);
// T is deduced to be int. a.xm must be convertible to
// A<int>::X
// i is explicitly specified to be 77, y.ym must be convertible
// to B<77>::Y
```

— end note ]

17 If, in the declaration of a function template with a non-type template-parameter, the non-type template-parameter is used in an expression in the function parameter-list and, if the corresponding template-argument is deduced, the template-argument type shall match the type of the template-parameter exactly, except that a template-argument deduced from an array bound may be of any integral type.\(^{134}\) [ Example:

```cpp
    template<int i> class A { /* ... */ };  
    template<short s> void f(A<s>);  
    void k1() {
        A<1> a;  
        f(a); // error: deduction fails for conversion from int to short  
        f<1>(a); // OK  
    }  
    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:

```cpp
    template<class T> void f(void (*)(T,int));  
    template<class T> void foo(T,int);  
    void g(int,int);  
    void g(char,int);  
```

\(^{134}\) 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.
void h(int,int,int);
void h(char,int);
int m()
{
  f(&g); // error: ambiguous
  f(&h); // OK: void h(char,int) is a unique match
  f(&foo); // error: type deduction fails because foo is a template
}

— end example ]

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 { };  
  A<int> ab;  
  f(ab); // calls f(A<int>)

— end example ]

21 [ 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 ... Types> struct Y { };  
  template<class T, class ... Types> struct Y<T, Types ...>{ };  

  template<class ... types> int f(void (*)(Types ...));  
  void g(int, float);

  X<int> x1; // uses primary template
  X<int(int, float, double)> x2; // uses partial specialization; ArgTypes contains float, double
  X<int(float, int)> x3; // uses primary template
  Y<> 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

Draft
If the original function parameter associated with A is a function parameter pack and the function parameter associated with P is not a function parameter pack, then template argument deduction fails. [Example:

```c++
template<class ... Args> void f(Args ... args); // #1
template<class T1, class ... Args> void f(T1 a1, Args ... args); // #2
template<class T1, class T2> void f(T1 a1, T2 a2); // #3
f(); // calls #1
f(1, 2, 3); // calls #2
f(1, 2); // calls #3: non-variadic template #3 is more specialized than the variadic templates #1 and #2
```

--- end example ---

14.8.3 Overload resolution [temp.over]

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-argument s (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 13.3.3.135)

[Example:

```c++
template<class T> T max(T a, T b) { return a>b?a:b; }
void f(int a, int b, char c, char d)
{
    int m1 = max(a,b);    // max(int a, int b)
    char m2 = max(c,d);   // max(char a, char b)
    int m3 = max(a,c);    // error: cannot generate max(int,char)
}
```

2 Adding the non-template function

```c++
int max(int,int);
```

135) 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.
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:

```
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:

```
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:

```
template<class T> void f(T); // declaration

void g()
{
  f("Annemarie"); // call of f<const char*>  
}
```

6 The call of `f` is well-formed even if the template `f` is only declared and not defined at the point of the call. The program will be ill-formed unless a specialization for `f<const char*>`, either implicitly or explicitly generated, is present in some translation unit. — end example ]

Draft
Exception handling provides a way of transferring control and information from a point in the execution of a program to an exception handler associated with a point previously passed by the execution. A handler will be invoked only by a throw-expression invoked in code executed in the handler’s try block or in functions called from the handler’s try block.

A try-block is a statement (clause 6). A throw-expression is of type void. Code that executes a throw-expression is said to “throw an exception;” code that subsequently gets control is called a “handler.” [Note: within this clause “try block” is taken to mean both try-block and function-try-block. — end note]

A goto or switch statement shall not be used to transfer control into a try block or into a handler. [Example:

```c
void f() {
    goto 11;          // Ill-formed
    goto 12;          // Ill-formed
    try {
        goto 11;      // OK
        goto 12;      // Ill-formed
        11: ;
    }
    catch (...) {
        goto 11;      // Ill-formed
        goto 12;      // OK
    }
    } 
```
A goto, break, return, or continue statement can be used to transfer control out of a try block or handler. When this happens, each variable declared in the try block will be destroyed in the context that directly contains its declaration. [Example:

```c
lab: try {
    T1 t1;
    try {
        T2 t2;
        if (condition)
            goto lab;
    } catch(...) { /* handler 2 */ }
} catch(...) { /* handler 1 */ }
```

Here, executing `goto lab;` will destroy first `t2`, then `t1`, assuming the `condition` does not declare a variable. Any exception 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 function-body compound-statement. An exception thrown during the execution of the initializer expressions in the ctor-initializer or during the execution of the function-body compound-statement 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:

```c
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 function-body statements
} catch (...) {
    // handles exceptions thrown from the ctor-initializer
    // and from the constructor function-body statements
}
```

— end example]
throw "Help!";

can be caught by a handler of const char* type:

```c++
try {
    // ... 
} catch(const char* p) {
    // handle character string exceptions here
}
```

and

```c++
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

```c++
try {
    // ...
    f(1.2);
    // ...
} catch(Overflow& oo) {
    // handle exceptions of type Overflow here
}
```

— end example ]

When an exception is thrown, control is transferred to the nearest handler with a matching type (15.3); “nearest” means the handler for which the compound-statement, or ctor-initializer, or function-body following the try keyword was most recently entered by the thread of control and not yet exited.

A throw-expression initializes a temporary object, called the exception object, the type of which is determined by removing any top-level cv-qualifiers from the static type of the operand of throw and adjusting the type from “array of T” or “function returning T” to “pointer to T” or “pointer to function returning T”, respectively. [ Note: the temporary object created for a throw-expression that is a string literal is never of type char*, char16_t*, char32_t*, or wchar_t*; that is, the special conversions for string literals from the types “array of const char”, “array of const char16_t”, “array of const char32_t”, and “array of const wchar_t” to the types “pointer to char”, “pointer to char16_t”, “pointer to char32_t”, and “pointer to wchar_t”, respectively (4.2), are never applied to a throw-expression. — end note ]

The temporary is an lvalue and is used to initialize the variable named in the matching handler (15.3). The type of the
throw-expression shall not be an incomplete type, or a pointer to an incomplete type other than (possibly cv-qualified) void. Except for these restrictions and the restrictions on type matching mentioned in 15.3, the operand of throw is treated exactly as a function argument in a call (5.2.2) or the operand of a return statement.

4 The memory for the temporary copy of the exception being thrown is allocated in an unspecified way, except as noted in 3.7.3.1. The temporary persists as long as there is a handler being executed for that exception. In particular, if a handler exits by executing a throw; statement, that passes control to another handler for the same exception, so the temporary remains. When the last remaining active handler for the exception exits by any means other than throw; the temporary object is destroyed and the implementation may deallocate the memory for the temporary object; any such deallocation is done in an unspecified way. The destruction occurs immediately after the destruction of the object declared in the exception-declaration in the handler.

5 When the thrown object is a class object, the copy constructor and the destructor shall be accessible, even if the copy 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 A throw-expression with no operand rethrows the currently handled exception (15.3). The exception is reactivated with the existing temporary; no new temporary exception object is created. The exception is no longer considered to be caught; therefore, the value of std::uncaught_exception() will again be true. [Example: code that must be executed because of an exception yet cannot completely handle the exception can be written like this:

```cpp
try {
  // ...
}
catch (...) {
  // catch all exceptions
  throw;  // pass the exception to some
          // other handler
}
```
—end example]

8 If no exception is presently being handled, executing a throw-expression with no operand calls std::terminate() (15.5.1).

15.2 Constructors and destructors [except.ctor]

1 As control passes from a throw-expression to a handler, destructors are invoked for all automatic objects constructed since the try block was entered. The automatic objects are destroyed in the reverse order of the completion of their construction.

2 An object that is partially constructed or partially destroyed will have destructors executed for all of its fully constructed subobjects, that is, for subobjects for which the principal constructor (12.6.2) has completed execution and the destructor has not yet begun execution. 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 will be invoked. Should a constructor for an element of an automatic array throw an exception, only the constructed elements of that array will be destroyed. If the object or array was allocated in a new-expression, the matching deallocation function (3.7.3.2, 5.3.4, 12.5), if any, is called to free the storage occupied by the object.
The process of calling destructors for automatic objects constructed on the path from a try block to a `throw-expression` is called “stack unwinding.” [Note: If a destructor called during stack unwinding exits with an exception, `std::terminate` is called (15.5.1). So destructors should generally catch exceptions and not let them propagate out of the destructor. — end note]

### 15.3 Handling an exception [except.handle]

1. The `exception-declaration` in a handler describes the type(s) of exceptions that can cause that handler to be entered. The `exception-declaration` shall not denote an incomplete type. 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*`. Types shall not be defined in an `exception-declaration`.

2. 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.

3. A handler is a match for an exception object of type E if
   - The handler is of type `cv T` or `cv T&` and E and T are the same type (ignoring the top-level `cv-qualifiers`), or
   - the handler is of type `cv T` or `cv T&` and T is an unambiguous public base class of E, or
   - the handler is of type `cv1 T* cv2` and E is a pointer type that can be converted to the type of the handler by either or both of
     - a standard pointer conversion (4.10) not involving conversions to pointers to private or protected or ambiguous classes
     - a qualification conversion

   [Note: a `throw-expression` which is an integral constant expression of integer type that evaluates to zero does not match a handler of pointer type; that is, the null pointer constant conversions (4.10, 4.11) do not apply. — 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) {
    //...
  }
}
```
15.3 Handling an exception

Exception handling 376

Here, the Overflow handler will catch exceptions of type Overflow and the Matherr handler will catch exceptions of type Matherr and of all types publicly derived from Matherr including exceptions of type Underflow and Zerodivide. — end example

The handlers for a try block are tried in order of appearance. 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.

A ... in a handler’s exception-declaration functions similarly to ... in a function parameter declaration; it specifies a match for any exception. If present, a ... handler shall be the last handler for its try block.

If no match is found among the handlers for a try block, the search for a matching handler continues in a dynamically surrounding try block.

A handler is considered active when initialization is complete for the formal 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.

The exception with the most recently activated handler that is still active is called the currently handled exception.

If no matching handler is found in a program, 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).

Referring to any non-static member or base class of an object in the handler for a function-try-block of a constructor or destructor for that object results in undefined behavior.

The fully constructed base classes and members of an object shall be destroyed before entering the handler of a function-try-block of a constructor or destructor for that object. Similarly, if a delegating constructor for an object exits with an exception after the non-delegating constructor for that object has completed execution, the object’s destructor shall be executed before entering the handler of a function-try-block of a constructor for that object.

The scope and lifetime of the parameters of a function or constructor extend into the handlers of a function-try-block.

Exceptions thrown in destructors of objects with static storage duration or in constructors of namespace-scope objects are not caught by a function-try-block on main().

If a return statement appears in a handler of the function-try-block of a constructor, the program is ill-formed.

The currently handled exception is rethrown if control reaches the end of a handler of the function-try-block of a constructor or destructor. Otherwise, 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).

When the exception-declaration specifies a class type, a copy constructor is used to initialize either the object declared in the exception-declaration or, if the exception-declaration does not specify a name, a temporary object of that type. The object shall not have an abstract class type. The object is destroyed when the handler exits, after the destruction of any automatic objects initialized within the handler. The copy constructor and destructor shall be accessible in the context of the handler. If the copy constructor and destructor are implicitly declared (12.8), such a use in the handler causes these functions to be implicitly defined; otherwise, the program shall provide a definition for these functions.

The copy constructor and destructor associated with the object shall be accessible even if the copy operation is elided (12.8).
When the handler declares a non-constant object, any changes to that object will not affect the temporary object that was initialized by execution of the `throw-expression`. When the handler declares a reference to a non-constant object, any changes to the referenced object are changes to the temporary object initialized when the `throw-expression` was executed and will have effect should that object be rethrown.

### 15.4 Exception specifications

A function declaration lists exceptions that its function might directly or indirectly throw by using an `exception-specification` as a suffix of its declarator.

```plaintext
exception-specification:
   throw ( type-id-list_opt )

type-id-list:
   type-id ... opt
   type-id-list , type-id ... opt
```

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.

```plaintext
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 an `exception-specification` shall not denote an incomplete type. A type denoted in an `exception-specification` shall not denote a pointer or reference to an incomplete type, other than `void*`, `const void*`, `volatile void*`, or `const volatile void*`.

If any declaration of a function has an `exception-specification`, all declarations, including the definition and an explicit specialization, of that function shall have an `exception-specification` with the same set of `type-id`s. 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 an `exception-specification` with the same set of `type-id`s. 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 have the same set of `type-id`s as other declarations of that function. A diagnostic is required only if the sets of `type-id`s are different 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.

```plaintext
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. [Example:

```c
class A { /* ... */ };  
void (*pf1)();  // no exception specification
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 match exactly. In other assignments or initializations, exception-specifications shall match exactly.

5 Types shall not be defined in exception-specifications.

7 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. An exception-specification can also include the class std::bad_exception (18.7.2.1).

8 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.

9 Whenever an exception is thrown and the search for a handler (15.3) encounters the outermost block of a function with an exception-specification, the function std::unexpected() is called (15.5.2) if the exception-specification does not allow the exception. [Example:

```c
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]

Draft
The function `std::unexpected()` may throw an exception that will satisfy the exception-specification for which it was invoked, and in this case the search for another handler will continue at the call of the function with this exception-specification (see 15.5.2), or it may call `std::terminate()`.

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]

A function with no exception-specification allows all exceptions. A function with an empty exception-specification, `throw()`, does not allow any exceptions.

An exception-specification is not considered part of a function’s type.

An implicitly declared special member function (clause 12) shall have an exception-specification. If `f` is an implicitly declared default constructor, copy constructor, destructor, or copy assignment operator, its implicit exception-specification specifies the type-id `T` if and only if `T` is allowed by the exception-specification of a function directly invoked by `f`’s implicit definition; `f` shall allow all exceptions if any function it directly invokes allows all exceptions, and `f` shall allow no exceptions if every function it directly invokes allows no exceptions. [Example:

```cpp
struct A {
    A();
    A(const A&) throw();
    ~A() throw(X);
};
struct B {
    B() throw();
    B(const B&) throw();
    ~B() throw(Y);
};
struct D : public A, public B {
    // Implicit declaration of D::D();
    // Implicit declaration of D::D(const D&) throw();
    // Implicit declaration of D::~D() throw(X,Y);
};
```

Furthermore, if `A::~A()` or `B::~B()` were virtual, `D::~D()` would not be as restrictive as that of `A::~A`, and the program would be ill-formed since a function that overrides a virtual function from a base class shall have an exception-specification at least as restrictive as that in the base class. — end example]

In an exception-specification, a type-id followed by an ellipsis is a pack expansion (14.5.3).
15.5 Special functions

Exception handling

with errors related to the exception handling mechanism itself (18.7).

15.5.1 The std::terminate() function

In the following situations exception handling must be abandoned for less subtle error handling techniques:

— when the exception handling mechanism, after completing evaluation of the expression to be thrown but before the exception is caught (15.1), calls a user function that exits via an uncaught exception,\(^{136}\)
— when the exception handling mechanism cannot find a handler for a thrown exception (15.3), or
— when the destruction of an object during stack unwinding (15.2) exits using an exception, or
— when construction or destruction of a non-local object with static storage duration exits using an exception (3.6.2), or
— when execution of a function registered with std::atexit exits using an exception (18.4), or
— when a throw-expression with no operand attempts to rethrow an exception and no exception is being handled (15.1), or
— when std::unexpected throws an exception which is not allowed by the previously violated exception-specification, and std::bad_exception is not included in that exception-specification (15.5.2), or
— when the implementation’s default unexpected exception handler is called (18.7.2.2).

In such cases, std::terminate() is called (18.7.3). In the situation where no matching handler is found, it is implementation-defined whether or not the stack is unwound before std::terminate() is called. In all other situations, the stack shall not be unwound before std::terminate() is called. An implementation is not permitted to finish stack unwinding prematurely based on a determination that the unwind process will eventually cause a call to std::terminate().

15.5.2 The std::unexpected() function

If a function with an exception-specification throws an exception that is not listed in the exception-specification, the function std::unexpected() is called (18.7.2) immediately after completing the stack unwinding for the former function.

The std::unexpected() function shall not return, but it can throw (or re-throw) 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 throws or rethrows an exception that the exception-specification does not allow then the following happens: If the exception-specification does not include the class std::bad_exception (18.7.2.1) then the function std::terminate() is called, otherwise the thrown exception is replaced by an implementation-defined object of the type std::bad_exception and the search for another handler will continue at the call of the function whose exception-specification was violated.

Thus, an exception-specification guarantees that only the listed exceptions will be thrown. If the exception-specification includes the type std::bad_exception then any exception not on the list may be replaced by std::bad_exception.

\(^{136}\) For example, if the object being thrown is of a class with a copy constructor, std::terminate() will be called if that copy constructor exits with an exception during a throw.
within the function `std::unexpected()`.

**15.5.3 The `std::uncaught_exception()` function**

The function `std::uncaught_exception()` returns `true` after completing evaluation of the object to be thrown until completing the initialization of the `exception-declaration` in the matching handler (18.7.4). This includes stack unwinding. If the exception is rethrown (15.1), `std::uncaught_exception()` returns `true` from the point of rethrow until the rethrown exception is caught again.

**15.6 Exceptions and access**

If the `exception-declaration` in a catch clause has class type, and the function in which the catch clause occurs does not have access to the destructor of that class, the program is ill-formed.

An object can be thrown if it can be copied and destroyed in the context of the function in which the `throw-expression` occurs.
A preprocessing directive consists of a sequence of preprocessing tokens. 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.\(^{137}\) A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro.

\[\text{preprocessing-file:}\]
\[\quad \text{group}_{opt}\]
\[\quad \text{group:}\]
\[\quad \quad \text{group-part}\]
\[\quad \quad \quad \text{group}_{group-part}\]
\[\quad \text{group-part:}\]
\[\quad \quad \text{if-section}\]
\[\quad \quad \quad \text{control-line}\]
\[\quad \quad \quad \text{text-line}\]
\[\quad \quad \quad \text{# non-directive}\]
\[\quad \text{if-section:}\]
\[\quad \quad \text{if-group} \quad \text{elif-groups}_{opt} \quad \text{else-group}_{opt} \quad \text{endif-line}\]
\[\text{if-group:}\]
\[\quad \text{# if} \quad \text{constant-expression} \quad \text{new-line} \quad \text{group}_{opt}\]
\[\quad \text{# ifdef} \quad \text{identifier} \quad \text{new-line} \quad \text{group}_{opt}\]
\[\quad \text{# ifndef} \quad \text{identifier} \quad \text{new-line} \quad \text{group}_{opt}\]
\[\text{elif-groups:}\]
\[\quad \text{elif-group}\]
\[\quad \quad \text{elif-groups} \quad \text{elif-group}\]
\[\text{elif-group:}\]
\[\quad \text{# endif} \quad \text{new-line} \quad \text{group}_{opt}\]
\[\text{else-group:}\]
\[\quad \text{# else} \quad \text{new-line} \quad \text{group}_{opt}\]
\[\text{endif-line:}\]
\[\quad \text{# endif} \quad \text{new-line}\]

\(^{137}\) 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).
control-line:
  # include pp-tokens new-line
  # define identifier replacement-list new-line
  # define identifier lparen identifier-list \opt replacement-list new-line
  # define identifier lparen \opt \opt replacement-list new-line
  # define identifier lparen identifier-list \opt replacement-list new-line
  # define identifier lparen \opt replacement-list \opt 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

text-line:
  pp-tokens \opt new-line

non-directive:
  pp-tokens \opt new-line

lparen:
  a ( character not immediately preceded by white-space

identifier-list:
  identifier
  identifier-list , identifier

replacement-list:
  pp-tokens \opt

pp-tokens:
  preprocessing-token
  pp-tokens preprocessing-token

new-line:
  the new-line character

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:
#define EMPTY
EMPTY  # include <file.h>

Draft
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 EMTPY has been replaced. — end example]

16.1 Conditional inclusion

The expression that controls conditional inclusion shall be an integral constant expression except that: it shall not contain a cast; identifiers (including those lexically identical to keywords) are interpreted as described below;\(^\text{138}\) and it may contain unary operator expressions of the form

\[
\text{defined} \ \text{identifier}
\]

or

\[
\text{defined} \ (\ \text{identifier})
\]

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 after all macro replacements have occurred shall be in the lexical form of a token (2.6).

Preprocessing directives of the forms

\[
\text{# if} \quad \text{constant-expression new-line group}^{\text{opt}}
\]

\[
\text{# elif} \quad \text{constant-expression new-line group}^{\text{opt}}
\]

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 \textbf{defined} unary operator), just as in normal text. If the token \textbf{defined} is generated as a result of this replacement process or use of the \textbf{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 \textbf{defined} unary operator have been performed, all remaining identifiers and keywords\(^\text{139}\), except for \text{true} and \text{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 5.19 using arithmetic that has at least the ranges specified in 18.2, except that all signed and unsigned integer types act as if they have the same representation as, respectively, \text{intmax}_t or \text{uintmax}_t (18.3.2). 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.\(^\text{140}\) Also, whether a single-character character literal may have a negative value is implementation-defined. Each subexpression with type \text{bool} is subjected to integral promotion before processing continues.

\(^{138}\)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, and so on.

\(^{139}\)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.

\(^{140}\)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.

\[
\text{#if } 'z' - 'a' == 25
\]

\[
\text{if } ( 'z' - 'a' == 25)
\]
Preprocessing directives of the forms

```
#include <h-char-sequence> new-line
```

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the " delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

```
#include <h-char-sequence> new-line
```

with the identical contained sequence (including > characters, if any) from the original directive.

A preprocessing directive of the form

```
#include "q-char-sequence" new-line
```

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the " delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

```
#include <h-char-sequence> new-line
```

with the identical contained sequence (including > characters, if any) from the original directive.

A preprocessing directive of the form

```
#include pp-tokens new-line
```

that does not match one of the two previous forms) is permitted. The preprocessing tokens after include in the directive are processed just as in normal text (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.\(^{142}\) 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.

---

\(^{141}\) 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.

\(^{142}\) Note that adjacent string literals are not concatenated into a single string literal (see the translation phases in 2.1); thus, an expansion that results in two string literals is an invalid directive.

Draft
The implementation provides unique mappings for sequences consisting of one or more nondigit s or digit s (2.10) followed by a period (.) and a single nondigit. The first character shall not be a digit. The implementation may ignore the distinctions of alphabetical case.

A #include preprocessing directive may appear in a source file that has been read because of a #include directive in another file, up to an implementation-defined nesting limit.

[ Note: Although an implementation may provide a mechanism for making arbitrary source files available to the < > search, in general programmers should use the < > form for headers provided with the implementation, and the " " form for sources outside the control of the implementation. For instance:

```c
#include <stdio.h>
#include <unistd.h>
#include "usefullib.h"
#include "myprog.h"
```

— end note ]

[ Example: Here is a macro-replaced #include directive:

```c
#if VERSION == 1
    #define INCFILE "vers1.h"
#elif VERSION == 2
    #define INCFILE "vers2.h"  // and so on
#else
    #define INCFILE "versN.h"
#endif
#include INCFILE
```

— end example ]

16.3 Macro replacement [cpp.replace]

Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

An identifier currently defined as an object-like macro 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\(^{143}\) to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive.\(^{144}\) The replacement list is then rescanned for more macro names as specified below.

A preprocessing directive of the form

```
#define identifier lparen identifier-list opt ) replacement-list new-line
#define identifier lparen ... ) replacement-list new-line
#define identifier lparen identifier-list , ... ) replacement-list new-line
```

defines a function-like macro with parameters, similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the `#define` preprocessing directive. Each subsequent instance of the function-like macro name followed by a `(` as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is terminated by the matching `)` preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If (before argument substitution) any argument consists of no preprocessing tokens, the behavior is undefined. 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 `...` in the identifier-list in the 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.

\(^{143}\) Since, by macro-replacement time, all character literals and string literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see 2.1.1.2, translation phases), they are never scanned for macro names or parameters.

\(^{144}\) 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.
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.

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 sequence; however, if an argument consists of no preprocessing tokens, the parameter is replaced by a placemarker preprocessing token instead.\(^\text{145}\)

For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemarker preprocessing tokens are handled specially; concatenation of two placemarkers results in a single placemarker preprocessing token, and concatenation of a placemarker with a non-placemarker preprocessing token results in the non-placemarker preprocessing token. If the result is not a valid preprocessing token, the behavior is undefined. The resulting token is available for further macro replacement. The order of evaluation of ## operators is unspecified.

**Example:** In the following fragment:

```c
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)
char p[] = join(x, y); // equivalent to
```

\(^{145}\)Placemaker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.
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 place-marker 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

```c
#define max(a, b) ((a) > (b) ? (a) : (b))
```

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.

[Note: The simplest use of this facility is to define a “manifest constant,” as in

```c
#define TABSIZE 100
int table[TABSIZE];
```]

The following defines a function-like macro whose value is the maximum of its arguments. It has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and generating more code than a function if invoked several times. It also cannot have its address taken, as it has none.

```c
#define max(a, b) ((a) > (b) ? (a) : (b))
```
The parentheses ensure that the arguments and the resulting expression are bound properly.

To illustrate the rules for redefinition and reexamination, the sequence

```c
#define x 3
#define f(a) f(x * (a))
#undef x
#define x 2
#define g f
#define z[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
#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

```c
f(2 * (y+1)) + f(2 * (f(2 * (z[0])))) % f(2 * (0)) + t(1);
f(2 * (2+(3,4)-0,1)) | f(2 * (~5)) & f(2 * (0,1))"m(0,1);
```

To illustrate the rules for creating character string literals and concatenating tokens, the sequence

```c
#define str(s) # s
#define xstr(s) str(s)
#define debug(s, t) printf("x" # s " = %d, x" # t " = %s", \
 x ## s, x ## t)
#define INCFILE(n) vers ## n /∗ from previous #include example ∗/
#define glue(a, b) a ## b
#define xglue(a, b) glue(a, b)
#define HIGHLOW "hello"
#define LOW LOW "world"
```

```c
dump(1, 2);
#define t(a) a
#define INCFILE(n) INCFILE(2)
```

results in

```c
Draft"
printf("x" "1" "= \%d, x" "2" "= \%s", x1, x2);
fgets("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);
fgets("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.

7 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(10,11), t(12), t(13) };

results in

  int j[] = { 123, 45, 67, 89,
             10, 11, 12, };

8 To demonstrate the redefinition rules, the following sequence is valid.

  #define OBJ_LIKE (1-1)
  #define OBJ_LIKE  /* white space */(1-1)/* other */
  #define FTN_LIKE(a) ( a )
  #define FTN_LIKE( a ) ( /* note the white space */
                        a /* 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 FTN_LIKE(b) ( a ) // different parameter usage
  #define FTN_LIKE(b) ( b ) // different parameter spelling

— end note ]

9 Finally, to show the variable argument list macro facilities:

  #define debug(...) fprintf(stderr, __VA_ARGS__)
  #define showlist(...) puts(#__VA_ARGS__)
  #define report(test, ...) ((test) ? puts(#test) : printf(__VA_ARGS__))
  debug("Flag");
  debug("X = \%d\textbackslash{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\n");
 fprintf(stderr, "X = %d\textbackslash{}n", x );
 puts( "The first, second, and third items." );
 ((x>y) ? puts("x>y") : printf("x is %d but y is %d", x, y));
```

16.4 Line control

1. The string literal of a `#line` directive, if present, shall be a character string literal.

2. The line number of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 (2.1) while processing the source file to the current token.

3. A preprocessing directive of the form

   ```
   # line digit-sequence new-line
   ```

   causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer). If the digit sequence specifies zero or a number greater than 2147483647, the behavior is undefined.

4. A preprocessing directive of the form

   ```
   # line digit-sequence "s-char-sequence_opt" new-line
   ```

   sets the line number similarly and changes the presumed name of the source file to be the contents of the character string literal.

5. A preprocessing directive of the form

   ```
   # line pp-tokens new-line
   ```

   (that does not match one of the two previous forms) is permitted. The preprocessing tokens after `line` on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined; otherwise, the result is processed as appropriate.

16.5 Error directive

1. A preprocessing directive of the form

   ```
   # error pp-tokens_opt new-line
   ```

   causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens, and renders the program ill-formed.

16.6Pragma directive

1. A preprocessing directive of the form

   ```
   # pragma pp-tokens_opt new-line
   ```
causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any pragma that is not recognized by the implementation is ignored.

### 16.7 Null directive

A preprocessing directive of the form

```cpp
# new-line
```

has no effect.

### 16.8 Predefined macro names

The following macro names shall be defined by the implementation:

- **__cplusplus**
  - The name __cplusplus is defined to the value [tbd] when compiling a C++ translation unit.\(^{146}\)

- **__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 is supplied.

- **__FILE__**
  - The presumed name of the source file (a character string literal).

- **__LINE__**
  - The line number of the current source line (a decimal constant).

- **__STDC_HOSTED__**
  - The integer constant 1 if the implementation is a hosted implementation or the integer constant 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 is 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_VERSION__**
  - Whether __STDC_VERSION__ is predefined and if so, what its value is, are implementation-defined.

- **__STDC_ISO_10646__**
  - An integer constant of the form yyyymmL (for example, 199712L), intended to indicate that values of type wchar_t are the coded representations of the characters defined by ISO/IEC 10646, along with all amendments and technical corrigenda as of the specified year and month.

\(^{146}\) 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.
The values of the predefined macros (except for __LINE__ and __FILE__) remain constant throughout the translation unit.

If any of the pre-defined macro names in this subclause, or the identifier defined, is the subject of a #define or a #undef preprocessing directive, the behavior is undefined. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

16.9 **Pragma operator**

A unary operator expression of the form:

```c
Pragma ( string-literal )
```

is processed as follows: The string literal is *destringized* by deleting the L prefix, if present, deleting the leading and trailing double-quotes, replacing each escape sequence " by a double-quote, and replacing each escape sequence \ by a single backslash. The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the *pp-tokens* in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

[Example:

```c
#pragma listing on ..\listing.dir
```

can also be expressed as:

```c
Pragma ( listing on "..\listing.dir"
```

The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:

```c
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x) _Pragma(#x)

LISTING( ..\listing.dir )
```

— end example ]
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 C++ Standard Library provides an extensible framework, and contains components for: language support, diagnostics, general utilities, strings, locales, containers, iterators, algorithms, numerics, and input/output. The language support components 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 general utilities include components used by other library elements, such as a predefined storage allocator for dynamic storage management (3.7.3). The diagnostics components provide a consistent framework for reporting errors in a C++ program, including predefined exception classes.

The strings components provide 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, or sequences of any other “character-like” type. The localization components extend internationalization support for such text processing.

The containers, iterators, and algorithms provide a C++ program with access to a subset of the most widely used algorithms and data structures.

Numeric algorithms and the complex number components extend support for numeric processing. The valarray components provide support for n-at-a-time processing, potentially implemented as parallel operations on platforms that support such processing.

The iostreams components are the primary mechanism for C++ program input/output. They can be used with other elements of the library, particularly strings, locales, and iterators.

This library also makes available the facilities of the Standard C library, suitably adjusted to ensure static type safety.

The descriptions of many library functions rely on the Standard C99 Library for the signatures and semantics of those functions. In all such cases, any use of the restrict qualifier shall be omitted.

The following subclauses describe the definitions (17.1), and method of description (17.3) for the library. Clause 17.4, and clauses 18 through 27, and Annex D specify the contents of the library, and library requirements and constraints on both well-formed C++ programs and conforming implementations.

### 17.1 Definitions

#### 17.1.1 arbitrary-positional stream

[definitions]

defns.arbitrary.stream]
17.1 Definitions

17.1.2 [defs.character]
character
in clauses 21, 22, and 27, means any object which, when treated sequentially, can represent text. The term does not only mean 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.

17.1.3 [defs.character.container]
character container type
a class or a type used to represent a character (). It is used for one of the template parameters of the string and iostream class templates. A character container type shall be a POD (3.9) type.

17.1.4 [defs.comparison]
comparison function
an operator function (13.5) for any of the equality (5.10) or relational (5.9) operators.

17.1.5 [defs.component]
component
a group of library entities directly related as members, parameters, or return types. For example, the class template basic_string and the non-member function templates that operate on strings are referred to as the string component.

17.1.6 [defs.default.behavior]
default behavior
a description of replacement function and handler function semantics. Any specific behavior provided by the implementation, within the scope of the required behavior.

17.1.7 [defs.handler]
handler function
a non-reserved function whose definition may be provided by a C++ program. 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).

17.1.8 [defs.iostream.templates]
iostream class templates
templates, defined in clause 27, that take two template arguments: 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.

17.1.9 [defs.modifier]

Draft
modifier function
a class member function (9.3), other than constructors, assignment, or destructor, that alters the state of an object of the class.

17.1.10 [defns.obj.state]
object state
the current value of all non-static class members of an object (9.2). The state of an object can be obtained by using one or more observer functions.

17.1.11 [defns.ntcts]
NTCTS
a sequence of values that have character type, that precede the terminating null character type value charT().

17.1.12 [defns.narrow.iostream]
narrow-oriented iostream classes
the instantiations of the iostream class templates on the character container class char and the default value of the traits parameter. The traditional iostream classes are regarded as the narrow-oriented iostream classes (27.3.1).

17.1.13 [defns.observer]
observer function
a class member function (9.3) that accesses the state of an object of the class, but does not alter that state. Observer functions are specified as const member functions (9.3.2).

17.1.14 [defns.replacement]
replacement function
a non-reserved function whose definition is provided by a C++ program. 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 (2.1) and resolving the definitions of all translation units (3.5).

17.1.15 [defns.repositional.stream]
repositional stream
a stream (described in clause 27) that can seek only to a position that was previously encountered.

17.1.16 [defns.required.behavior]
required behavior
a description of replacement function and handler function semantics, applicable to both the behavior provided by the implementation and the behavior that shall be provided by any function definition in the program. If a function defined in a C++ program fails to meet the required behavior when it executes, the behavior is undefined.

17.1.17 [defns.reserved.function]
reserved function

Draft
a function, specified as part of the C++ Standard Library, that must be defined by the implementation. If a C++ program provides a definition for any reserved function, the results are undefined.

17.1.18 stable algorithm
an algorithm that preserves, as appropriate to the particular algorithm, the order of elements.

— For the sort algorithms the relative order of equivalent elements is preserved.
— For the remove algorithms the relative order of the elements that are not removed is preserved.
— For the merge algorithms, for equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range.

17.1.19 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. 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.

17.1.20 wide-oriented iostream classes
the instantiations of the iostream class templates on the character container class wchar_t and the default value of the traits parameter (27.3.2).

17.2 Additional definitions
1.3 defines additional terms used elsewhere in this International Standard.

17.3 Method of description (Informative)
17.3 describes the conventions used to describe the C++ Standard Library. It describes the structures of the normative clauses 18 through 27 (17.3.1), Annex D and other editorial conventions (17.3.2).

17.3.1 Structure of each subclause
17.4.1 provides a summary of the C++ Standard library’s contents. Other Library clauses provide detailed specifications for each of the components in the library, as shown in Table 12.

Each Library clause contains the following elements, as applicable:¹⁴⁷)

— Summary
— Requirements
— Detailed specifications

¹⁴⁷) To save space, items that do not apply to a clause are omitted. For example, if a clause does not specify any requirements, there will be no “Requirements” subclause.
17.3 Method of description (Informative)

<table>
<thead>
<tr>
<th>Clause</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>Language support</td>
</tr>
<tr>
<td>19</td>
<td>Diagnostics</td>
</tr>
<tr>
<td>20</td>
<td>General utilities</td>
</tr>
<tr>
<td>21</td>
<td>Strings</td>
</tr>
<tr>
<td>22</td>
<td>Localization</td>
</tr>
<tr>
<td>23</td>
<td>Containers</td>
</tr>
<tr>
<td>24</td>
<td>Iterators</td>
</tr>
<tr>
<td>25</td>
<td>Algorithms</td>
</tr>
<tr>
<td>26</td>
<td>Numerics</td>
</tr>
<tr>
<td>27</td>
<td>Input/output</td>
</tr>
<tr>
<td>28</td>
<td>Regular expressions</td>
</tr>
</tbody>
</table>

— References to the Standard C library

17.3.1 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 labelled “Note(s):” or “Example(s):” are informative, other paragraphs are normative.

3 The summary and the detailed specifications are presented in the order:
   — Macros
   — Values
   — Types
   — Classes
   — Functions
   — Objects

17.3.1.2 Requirements

1 The library can be extended by a C++ program. Each clause, as applicable, describes the requirements that such extensions must meet. Such extensions are generally one of the following:
   — Template arguments
   — Derived classes
   — Containers, iterators, and/or algorithms that meet an interface convention

Draft
2 The string and iostreams components use an explicit representation of operations required of template arguments. They use a class template char_traits to define these constraints.

3 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.

4 Requirements are stated in terms of well-defined expressions, which define valid terms of the types that satisfy the requirements. For every set of requirements there is a table that specifies an initial set of the valid expressions and their semantics (20.1.2, 23.1, 24.1). Any generic algorithm (clause 25) that uses the requirements is described in terms of the valid expressions for its formal type parameters.

5 Template argument requirements are sometimes referenced by name. See 17.3.2.1.

6 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.\(^{148}\)

### 17.3.1.3 Specifications

1 The detailed specifications each contain the following elements:\(^{149}\)

- Name and brief description
- Synopsis (class definition or function prototype, as appropriate)
- Restrictions on template arguments, if any
- Description of class invariants
- Description of function semantics

2 Descriptions of class member functions follow the order (as appropriate):\(^{150}\)

- Constructor(s) and destructor
- Copying & assignment functions
- Comparison functions
- Modifier functions
- Observer functions
- Operators and other non-member functions

3 Descriptions of function semantics contain the following elements (as appropriate):\(^{151}\)

- Requires: the preconditions for calling the function

---

\(^{148}\) Although in some cases the code given is unambiguously the optimum implementation.

\(^{149}\) The form of these specifications was designed to follow the conventions established by existing C++ library vendors.

\(^{150}\) To save space, items that do not apply to a class are omitted. For example, if a class does not specify any comparison functions, there will be no “Comparison functions” subclause.

\(^{151}\) 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.
— *Effects:* the actions performed by 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

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.

Complexity requirements specified in the library clauses are upper bounds, and implementations that provide better complexity guarantees satisfy the requirements.

### 17.3.1.4 C Library

Paragraphs labelled “SEE ALSO:” contain cross-references to the relevant portions of this Standard and the ISO C standard, which is incorporated into this Standard by reference.

### 17.3.2 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 (17.3.2.1), and member functions (17.3.2.2).

### 17.3.2.1 Type descriptions

The Requirements subclauses may describe names that are used to specify constraints on template arguments. These names are used in library clauses to describe the types that may be supplied as arguments by a C++ program when instantiating template components from the library.

Certain types defined in clause 27 are used to describe implementation-defined types. They are based on other types, but with added constraints.

### 17.3.2.1.1 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
enum enumerated { V0, V1, V2, V3, .......};
static const enumerated C0 (V0);
static const enumerated C1 (V1);
static const enumerated C2 (V2);
static const enumerated C3 (V3);
```

---

152) Examples from 20.1 include: EqualityComparable, LessThanComparable, CopyConstructable, etc. Examples from 24.1 include: InputIterator, ForwardIterator, Function, Predicate, etc.
153) Such as an integer type, with constant integer values (3.9.1).
Here, the names \( C_0 \), \( C_1 \), etc. represent enumerated elements for this particular enumerated type. All such elements have distinct values.

### 17.3.2.1.2 Bitmask types

Several types defined in clauses 18 through 27 and Annex D are bitmask types. Each bitmask type can be implemented as an enumerated type that overloads certain operators, as an integer type, or as a bitset (23.3.5).

The bitmask type `bitmask` can be written:

```cpp
enum bitmask {
    V0 = 1 << 0, V1 = 1 << 1, V2 = 1 << 2, V3 = 1 << 3, ...
};

static const bitmask C0(V0);
static const bitmask C1(V1);
static const bitmask C2(V2);
static const bitmask C3(V3);
...
```

Here, the names \( C_0 \), \( C_1 \), etc. represent bitmask elements for this particular bitmask type. All such elements have distinct values such that, for any pair \( C_i \) and \( C_j \), \( C_i \& C_j \) is nonzero and \( C_i \& C_j \) is zero.
The following terms apply to objects and values of bitmask types:

- To set a value \( Y \) in an object \( X \) is to evaluate the expression \( X \mid= Y \).
- To clear a value \( Y \) in an object \( X \) is to evaluate the expression \( X \land= \sim Y \).
- The value \( Y \) is set in the object \( X \) if the expression \( X \land Y \) is nonzero.

### 17.3.2.1.3 Character sequences

The Standard C library makes widespread use of characters and character sequences that follow a few uniform conventions:

- A letter is any of the 26 lowercase or 26 uppercase letters in the basic execution character set.\(^{154}\)
- 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 27 and Annex D by a period, \( \cdot \), which is also its value in the "C" locale, but may change during program execution by a call to `setlocale(int, const char*)`,\(^{155}\) or by a change to a locale object, as described in clauses 22.1 and 27.
- A character sequence is an array object (8.3.4) \( A \) that can be declared as \( \text{T} \ A [N] \), where \( \text{T} \) is any of the types \( \text{char}, \text{unsigned char}, \text{or signed char} \) (3.9.1), optionally qualified by any combination of \texttt{const} or \texttt{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.

### 17.3.2.1.3.1 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).\(^{156}\)

The length of an NTBS is the number of elements that precede the terminating null character. An empty NTBS has a length of zero.

The value of an NTBS is the sequence of values of the elements up to and including the terminating null character.

A static NTBS is an NTBS with static storage duration.\(^{157}\)

### 17.3.2.1.3.2 Multibyte strings

A null-terminated multibyte string, or NTMBS, is an NTBS that constitutes a sequence of valid multibyte characters, beginning and ending in the initial shift state.\(^{158}\)

---

\(^{154}\) Note that this definition differs from the definition in ISO C subclause 7.1.1.

\(^{155}\) declared in `<locale>` (22.3).

\(^{156}\) Many of the objects manipulated by function signatures declared in `<cstring>` (21.4) are character sequences or NTBSs. The size of some of these character sequences is limited by a length value, maintained separately from the character sequence.

\(^{157}\) A string literal, such as "abc", is a static NTBS.

\(^{158}\) 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.
17.3 Method of description (Informative) Library introduction 406

2 A static NTMBS is an NTMBS with static storage duration.

17.3.2.1.3.3 char16_t sequences

A char16-character sequence is an array object (8.3.4) $A$ that can be declared as $T \ [N]$, where $T$ is type char16_t (3.9.1), optionally qualified by any combination of const or volatile. The initial elements of the array have defined contents up to and including an element determined by some predicate. A char16-character sequence can be designated by a pointer value $S$ that designates its first element.

2 A null-terminated char16-character string, or NTC16S, is a char16-character sequence whose highest-addressed element with defined content has the value zero.

3 The length of an NTC16S is the number of elements that precede the terminating null char16_t character. An empty NTC16S has a length of zero.

4 The value of an NTC16S is the sequence of values of the elements up to and including the terminating null character.

5 A static NTC16S is an NTC16S with static storage duration.

17.3.2.1.3.4 char32_t sequences

A char32-character sequence is an array object (8.3.4) $A$ that can be declared as $T \ [N]$, where $T$ is type char32_t (3.9.1), optionally qualified by any combination of const or volatile. The initial elements of the array have defined contents up to and including an element determined by some predicate. A char32-character sequence can be designated by a pointer value $S$ that designates its first element.

2 A null-terminated char32-character string, or NTC32S, is a char32-character sequence whose highest-addressed element with defined content has the value zero.

3 The length of an NTC32S is the number of elements that precede the terminating null char32_t character. An empty NTC32S has a length of zero.

4 The value of an NTC32S is the sequence of values of the elements up to and including the terminating null character.

5 A static NTC32S is an NTC32S with static storage duration.

17.3.2.1.3.5 Wide-character sequences

A wide-character sequence is an array object (8.3.4) $A$ that can be declared as $T \ [N]$, where $T$ is type wchar_t (3.9.1), optionally qualified by any combination of const or volatile. The initial elements of the array have defined contents up to and including an element determined by some predicate. A wide-character sequence can be designated by a pointer value $S$ that designates its first element.

2 A null-terminated wide-character string, or NTWCS, is a wide-character sequence whose highest-addressed element with defined content has the value zero.

3 The length of an NTWCS is the number of elements that precede the terminating null wide character. An empty NTWCS has a length of zero.

4 The value of an NTWCS is the sequence of values of the elements up to and including the terminating null character.

5 A static NTWCS is an NTWCS with static storage duration.

159) Many of the objects manipulated by function signatures declared in <cuchar> are char16-character sequences or NTC16Ss.

160) A char16_t string literal, such as u"abc", is a static NTC16S.

161) Many of the objects manipulated by function signatures declared in <cuchar> are char32-character sequences or NTC32Ss.

162) A char32_t string literal, such as U"abc", is a static NTC32S.

163) Many of the objects manipulated by function signatures declared in <cwchar> are wide-character sequences or NTWCSs.
The value of an NTWCS is the sequence of values of the elements up to and including the terminating null character.

A static NTWCS is an NTWCS with static storage duration.\(^4\)

### 17.3.2.2 Functions within classes

For the sake of exposition, clauses 18 through 27 and Annex D do not describe copy 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).

It is unspecified whether the implementation provides explicit definitions for such member function signatures, or for virtual destructors that can be generated by default.

### 17.3.2.3 Private members

Clauses 18 through 27 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 27 and Annex D.

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 enclosed in a comment that ends with exposition only, as in:

```cpp
// streambuf* sb;
```

Any alternate implementation that provides equivalent external behavior is equally acceptable.

### 17.4 Library-wide requirements

This subclause specifies requirements that apply to the entire C++ Standard library. Clauses 18 through 27 and Annex D specify the requirements of individual entities within the library.

The following subclauses describe the library’s contents and organization (17.4.1), how well-formed C++ programs gain access to library entities (17.4.2), constraints on such programs (17.4.3), and constraints on conforming implementations (17.4.4).

#### 17.4.1 Library contents and organization

This subclause provides a summary of the entities defined in the C++ Standard Library. In general, these entities are defined in library headers, which subclause 17.4.1.2 lists alphabetically.

#### 17.4.1.1 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.

\(^4\) A wide string literal, such as L"abc" is a static NTWCS.
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.

### 17.4.1.2 Headers

The elements of the C++ Standard Library are declared or defined (as appropriate) in a header.\[165]\[166]\[167]

The C++ Standard Library provides 40 C++ headers, as shown in Table 13.

#### Table 13: C++ Library Headers

| `<algorithm>` | `<ios>` | `<new>` | `<stack>` | `<unordered_map>` |
| `<array>` | `<iosfwd>` | `<numeric>` | `<stdexcept>` | `<unordered_set>` |
| `<bitset>` | `<iostream>` | `<ostream>` | `<streambuf>` | `<utility>` |
| `<complex>` | `<istream>` | `<queue>` | `<string>` | `<valarray>` |
| `<deque>` | `<iterator>` | `<random>` | `<sstream>` | `<vector>` |
| `<exception>` | `<limits>` | `<regex>` | `<system_error>` |
| `<fstream>` | `<list>` | `<set>` | `<tuple>` |
| `<functional>` | `<locale>` | `<map>` | `<type_traits>` |
| `<iomanip>` | `<memory>` | `<sstream>` | `<typeinfo>` |

The facilities of the Standard C Library are provided in 24 additional headers, as shown in Table 14.

#### Table 14: C++ Headers for C Library Facilities

| `<cassert>` | `<cfloat>` | `<cmath>` | `<cstddef>` | `<ctgmath>` |
| `<ccomplex>` | `<cinttypes>` | `<csetjmp>` | `<cstdio>` | `<ctime>` |
| `<cctype>` | `<ciso646>` | `<csignal>` | `<cstdlib>` | `<cwchar>` |
| `<cerrno>` | `<climits>` | `<cstdarg>` | `<cwchar.h>` | `<cwctype>` |

Except as noted in clauses 18 through 27 and Annex D, the contents of each header `cname` shall be the same as that of the corresponding header `cname.h`, as specified in the Standard C99 Library (1.2) or the C Unicode 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 (3.3.5) 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]

Names that are defined as functions in C shall be defined as functions in the C++ Standard Library.\[166]\[167]

Identifiers that are keywords or operators in C shall not be defined as macros in C++ standard library headers.\[167\]

---

\[165\] 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).

\[166\] This disallows the practice, allowed in C, of providing a “masking macro” in addition to the function prototype. The only way to achieve equivalent “inline” behavior in C++ is to provide a definition as an extern inline function.

\[167\] In particular, including the standard header `<iso646.h>` or `<ciso646>` has no effect.
17.4 Library-wide requirements

17.4.1.3 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 following headers, as shown in Table 15.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.1 Types</td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>18.2 Implementation properties</td>
<td>&lt;limits&gt;</td>
</tr>
<tr>
<td>18.4 Start and termination</td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>18.5 Dynamic memory management</td>
<td>&lt;new&gt;</td>
</tr>
<tr>
<td>18.6 Type identification</td>
<td>&lt;typeinfo&gt;</td>
</tr>
<tr>
<td>18.7 Exception handling</td>
<td>&lt;exception&gt;</td>
</tr>
<tr>
<td>18.8 Other runtime support</td>
<td>&lt;cstdarg&gt;</td>
</tr>
</tbody>
</table>

The supplied version of the header <cstdlib> shall declare at least the functions abort(), atexit(), and exit() (18.4).

17.4.2 Using the library

This subclause describes how a C++ program gains access to the facilities of the C++ Standard Library. 17.4.2.1 describes effects during translation phase 4, while 17.4.2.2 describes effects during phase 8 (2.1).

17.4.2.1 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).

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.169

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 to any of the entities it declares in that translation unit.

17.4.2.2 Linkage

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).

---

168) The ".*.h" headers dump all their names into the global namespace, whereas the newer forms keep their names in namespace std. Therefore, the newer forms are the preferred forms for all uses except for C++ programs which are intended to be strictly compatible with C.

169) This is the same as the Standard C library.
17.4 Library-wide requirements

2 Whether a name from the Standard C 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.\(^{170}\)

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 (17.4.3.4), run-time changes (17.4.3.5).

17.4.3 Constraints on programs

1 This subclause describes restrictions on C++ programs that use the facilities of the C++ Standard Library. The following subclauses specify constraints on the program’s namespace (17.4.3.1), its use of headers (17.4.3.2), classes derived from standard library classes (17.4.3.3), definitions of replacement functions (17.4.3.4), and installation of handler functions during execution (17.4.3.5).

17.4.3.1 Reserved names

1 It is undefined for a C++ program to add declarations or definitions to namespace std or namespaces within namespace std unless otherwise specified. A program may add template specializations for any standard library template to namespace std. Such a specialization (complete or partial) of a standard library template results in undefined behavior unless the declaration depends on a user-defined type of external linkage and unless the specialization meets the standard library requirements for the original template.\(^{171}\) It is undefined for a C++ program to declare

— an explicit specialization of any member function of a standard library class template, or
— an explicit specialization of any member function template of a standard library class or class template, or
— an explicit or partial specialization of any member class template of a standard library class or class template.

A program may explicitly instantiate any templates in the standard library only if the declaration depends on the name of a user-defined type of external linkage and the instantiation meets the standard library requirements for the original template.

2 The C++ Standard Library reserves the following kinds of names:

— Macros
— Global names
— Names with external linkage

3 If the program declares or defines a name in a context where it is reserved, other than as explicitly allowed by this clause, the behavior is undefined.

17.4.3.1.1 Macro names

1 A translation unit that includes a standard library header shall not \#define or \#undef names declared in any standard library header.

\(^{170}\) 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 subclause 7.1.7 of the C Standard.

\(^{171}\) 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.


A translation unit shall not \#define or \#undef names lexically identical to keywords.

**17.4.3.1.2 Global names**

Certain sets of names and function signatures are always reserved to the implementation:

- Each name that contains a double underscore \_\_ or begins with an underscore followed by an uppercase letter (2.11) is reserved to the implementation for any use.
- Each name that begins with an underscore is reserved to the implementation for use as a name in the global namespace.\(^{172}\)

**17.4.3.1.3 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,\(^{173}\) 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.\(^{174}\)

Each name having two consecutive underscores (2.11) is reserved to the implementation for use as a name with both extern "C" and extern "C++" linkage.

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,\(^{175}\) or as a name of namespace scope in the global namespace.

**17.4.3.1.4 Types**

For each type T from the Standard C library,\(^{176}\) the types ::T and std::T are reserved to the implementation and, when defined, ::T shall be identical to std::T.

**17.4.3.2 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.

**17.4.3.3 Derived classes**

Virtual member function signatures defined for a base class in the C++ Standard Library may be overridden in a derived

---

\(^{172}\) Such names are also reserved in namespace ::std (17.4.3.1).

\(^{173}\) The list of such reserved names includes errno, declared or defined in <cerrno>.

\(^{174}\) 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 <cstdlib>.

\(^{175}\) 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.

\(^{176}\) 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.
class defined in the program (10.3).

17.4.3.4 Replacement functions

Clauses 18 through 27 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 (17.1).

A C++ program may provide the definition for any of eight dynamic memory allocation function signatures declared in header `<new>` (3.7.3, clause 18):

- operator new(std::size_t)
- operator new(std::size_t, const std::nothrow_t&)
- operator new[](std::size_t)
- operator new[](std::size_t, const std::nothrow_t&)
- operator delete(void*)
- operator delete(void*, const std::nothrow_t&)
- operator delete[](void*)
- operator delete[](void*, const std::nothrow_t&)

The program’s definitions are used instead of the default versions supplied by the implementation (18.5). Such replacement occurs prior to program startup (3.2, 3.6). The program’s definitions shall not be specified as `inline`. No diagnostic is required.

17.4.3.5 Handler functions

The C++ Standard Library provides default versions of the following handler functions (clause 18):

- unexpected_handler
- 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):

- set_new_handler
- set_unexpected
- set_terminate

See also: subclauses 18.5.2, Storage allocation errors, and 18.7, Exception handling.

Draft
17.4.3.6 Other functions

In certain cases (replacement functions, handler functions, operations on types used to instantiate standard library template components), the C++ Standard Library depends on components supplied by a C++ program. If these components do not meet their requirements, the Standard places no requirements on the implementation.

In particular, the effects are undefined in the following cases:

— for replacement functions (18.5.1), if the installed replacement function does not implement the semantics of the applicable Required behavior: paragraph.

— for handler functions (18.5.2.2, 18.7.3.1, 18.7.2.2), if the installed handler function does not implement the semantics of the applicable Required behavior: paragraph

— for types used as template arguments when instantiating a template component, if the operations on the type do not implement the semantics of the applicable Requirements subclause (20.1.2, 23.1, 24.1, 26.1). Operations on such types can report a failure by throwing an exception unless otherwise specified.

— if any replacement function or handler function or destructor operation throws an exception, unless specifically allowed in the applicable Required behavior: paragraph.

— if an incomplete type (3.9) is used as a template argument when instantiating a template component, unless specifically allowed for that component.

17.4.3.7 Function arguments

Each of the following statements applies to all arguments to functions defined in the C++ Standard Library, unless explicitly stated otherwise.

— If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer invalid for its intended use), the behavior is undefined.

— If a function argument is described as being an array, the pointer actually passed to the function shall have a value such that all address computations and accesses to objects (that would be valid if the pointer did point to the first element of such an array) are in fact valid.

17.4.3.8 Required paragraph

Violation of the preconditions specified in a function’s Required behavior: paragraph results in undefined behavior unless the function’s Throws: paragraph specifies throwing an exception when the precondition is violated.

17.4.4 Conforming implementations

This subclause describes the constraints upon, and latitude of, implementations of the C++ Standard library. The following subclauses describe an implementation’s use of headers (17.4.4.1), macros (17.4.4.2), global functions (17.4.4.3),
member functions (17.4.4.4), reentrancy (17.4.5), access specifiers (17.4.6), class derivation (17.4.7), and exceptions (17.4.8).

17.4.4.1 Headers [res.on.headers]

1 A C++ header may include other C++ headers.\(^{(177)}\)

2 Certain types and macros are defined in more than one header. For such an entity, a second or subsequent header that also defines it may be included after the header that provides its initial definition (3.2).

3 Header inclusion is limited as follows:
   — The C headers (.h form, described in Annex D, D.5) shall include only their corresponding C++ header, as described above (17.4.1.2).

17.4.4.2 Restrictions on macro definitions [res.on.macro.definitions]

1 The names or global function signatures described in 17.4.1.1 are reserved to the implementation.

2 All object-like macros defined by the Standard C 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.

17.4.4.3 Global or non-member functions [global.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 27 and Annex D behaves the same as if the implementation declares no additional global or non-member function signatures.\(^{(178)}\)

3 A global or non-member function cannot be declared by the implementation as taking additional default arguments.

4 Unless otherwise specified, no global or non-member function in the standard library shall use a function from another namespace which is 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\_iterators: Effects:

```c
*out_stream << value;
if (delim != 0)
   *out_stream << delim;
return (*this);
```

— end note ]

17.4.4.4 Member functions [member.functions]

1 It is unspecified whether any member functions in the C++ Standard Library are defined as inline (7.1.2).

2 An implementation can declare additional non-virtual member function signatures within a class:
   — by adding arguments with default values to a member function signature;\(^{(179)}\) The same latitude does not extend to the implementation of virtual or global or non-member functions, however.

\(^{(177)}\) C++ headers must include a C++ header that contains any needed definition (3.2).

\(^{(178)}\) 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.

\(^{(179)}\) Hence, taking the address of a member function has an unspecified type.
— by replacing a member function signature with default values by two or more member function signatures with equivalent behavior;
— 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 the same as if the implementation declares no additional member function signatures. \(^{180}\)

17.4.4.5 Reentrancy [reentrancy]

Which of the functions in the C++ Standard Library are not reentrant subroutines is implementation-defined.

17.4.4.6 Protection within classes [protection.within.classes]

It is unspecified whether a function signature or class described in clauses 18 through 27 and Annex D is a friend of another class in the C++ Standard Library.

17.4.4.7 Derived classes [derivation]

It is unspecified whether a class in the C++ Standard Library is itself derived from other classes (with names reserved to the implementation).

Certain classes defined in the C++ Standard Library are derived from other classes in the C++ Standard Library:

— It is unspecified whether a class described in the C++ Standard Library as derived from another class is derived from that class directly, or through other classes (with names reserved to the implementation) that are derived from the specified base class.

In any case:

— A base class described as virtual is always virtual;
— A base class described as non-virtual is never virtual;
— Unless explicitly stated otherwise, types with distinct names are distinct types. \(^{181}\)

17.4.4.8 Restrictions on exception handling [res.on.exception.handling]

Any of the functions defined in the C++ Standard Library can report a failure by throwing an exception of the type(s) described in their Throws: paragraph and/or their exception-specification (15.4). An implementation may strengthen the exception-specification for a non-virtual function by removing listed exceptions. \(^{182}\)

None of the functions from the Standard C library shall report an error by throwing an exception, \(^{183}\) unless it calls a program-supplied function that throws an exception. \(^{184}\)

\(^{180}\) 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.

\(^{181}\) An implicit exception to this rule are types described as synonyms for basic integral types, such as size_t (18.1) and streamoff (27.4.1).

\(^{182}\) That is, an implementation of the function will have an explicit exception-specification that lists fewer exceptions than those specified in this International Standard. It may not, however, change the types of exceptions listed in the exception-specification from those specified, nor add others.

\(^{183}\) That is, the C library functions all have a throw() exception-specification. This allows implementations to make performance optimizations based on the absence of exceptions at runtime.

\(^{184}\) The functions qsort() and bsearch() (25.4) meet this condition.
No destructor operation defined in the C++ Standard Library will throw an exception. Any other functions defined in the C++ Standard Library that do not have an exception-specification may throw implementation-defined exceptions unless otherwise specified.\(^{185}\) An implementation may strengthen this implicit exception-specification by adding an explicit one.\(^{186}\)

\(^{185}\) In particular, they can report a failure to allocate storage by throwing an exception of type \texttt{bad\_alloc}, or a class derived from \texttt{bad\_alloc} (18.5.2.1). Library implementations are encouraged (but not required) to report errors by throwing exceptions from (or derived from) the standard exception classes (18.5.2.1, 18.7, 19.1).

\(^{186}\) 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.
Chapter 18  Language support library
[language.support]

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, and other runtime support, as summarized in Table 16.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.1 Types</td>
<td><code>&lt;cstdint&gt;</code></td>
</tr>
<tr>
<td>18.2 Implementation properties</td>
<td><code>&lt;limits&gt;</code> <code>&lt;climits&gt;</code> <code>&lt;cfloat&gt;</code></td>
</tr>
<tr>
<td>18.3 Integer types</td>
<td><code>&lt;cstddef&gt;</code></td>
</tr>
<tr>
<td>18.4 Start and termination</td>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
<tr>
<td>18.5 Dynamic memory management</td>
<td><code>&lt;new&gt;</code></td>
</tr>
<tr>
<td>18.6 Type identification</td>
<td><code>&lt;typeinfo&gt;</code></td>
</tr>
<tr>
<td>18.7 Exception handling</td>
<td><code>&lt;exception&gt;</code></td>
</tr>
<tr>
<td>18.8 Other runtime support</td>
<td><code>&lt;cstddef&gt;</code> <code>&lt;csetjmp&gt;</code> <code>&lt;ctime&gt;</code> <code>&lt;csignal&gt;</code> <code>&lt;cstdlib&gt;</code> <code>&lt;cstdbool&gt;</code></td>
</tr>
</tbody>
</table>

18.1 Types [support.types]

1. Common definitions.

   Header `<cstdint>` (Table 17).

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).\(^\text{187}\)

The macro `offsetof(type, member-designator)` accepts a restricted set of `type` arguments in this International Standard. If `type` is not a POD standard-layout class (clause 9), the results are undefined.\(^\text{188}\) 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.

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.

**SEE ALSO:** subclause 5.3.3, Sizeof, subclause 5.7, Additive operators, subclause 12.5, Free store, and ISO C sub-clause 7.1.6.

### 18.2 Implementation properties

The headers `<limits>`, `<climits>`, `<cfloat>`, and `<cinttypes>` supply characteristics of implementation-dependent arithmetic types (3.9.1).

#### 18.2.1 Numeric limits

The `numeric_limits` component 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 `constexpr` in the `numeric_limits` template, specializations shall define these values in such a way that they are usable as integral constant expressions.

Non-arithmetic standard types, such as `complex<T>` (26.3.2), shall not have specializations.

#### Header `<limits>` synopsis

```cpp
class numeric_limits;
enum float_round_style;
enum float_denorm_style;

class numeric_limits<bool>;

class numeric_limits<char>;
```

\(^{187}\) Possible definitions include 0 and 0L, but not `(void*)0`.
\(^{188}\) Note that `offsetof` is required to work as specified even if unary `operator&` is overloaded for any of the types involved.
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>;

}[]

18.2.1.1 Class template numeric_limits

namespace std {
    template<class T> class numeric_limits {
        public:
            static constexpr bool is_specialized = false;
            static constexpr T min() throw();
            static constexpr T max() throw();
            static constexpr T lowest() throw();

            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() throw();
            static constexpr T round_error() throw();

            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;
        }
    }
}
18.2 Implementation properties

static constexpr T infinity() throw();
static constexpr T quiet_NaN() throw();
static constexpr T signaling_NaN() throw();
static constexpr T denorm_min() throw();

static constexpr bool is_iec559 = false;
static constexpr bool is_bounded = false;
static constexpr bool is_modulo = false;
static constexpr bool traps = false;
static constexpr bool tinyness_before = false;

static constexpr float_round_style round_style = round_toward_zero;

};

template<class T> class numeric_limits<const T>;

template<class T> class numeric_limits<volatile T>;

template<class T> class numeric_limits<const volatile T>;

}

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.

18.2.1.2 numeric_limits members

static constexpr T min() throw();
1 Minimum finite value.\(^{189}\)
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() throw();
4 Maximum finite value.\(^{190}\)
5 Meaningful for all specializations in which is_bounded != false.

static constexpr T lowest() throw();
6 A finite value x such that there is no other finite value y where y < x.\(^{191}\)
7 Meaningful for all specializations in which is_bounded != false.

static constexpr int digits;

\(^{189}\) Equivalent to CHAR_MIN, SHRT_MIN, FLT_MIN, DBL_MIN, etc.
\(^{190}\) Equivalent to CHAR_MAX, SHRT_MAX, FLT_MAX, DBL_MAX, etc.
\(^{191}\) lowest () is necessary because not all floating-point representations have a smallest (most negative) value that is the negative of the largest (most positive) finite value.
Number of radix digits that can be represented without change.
For integer types, the number of non-sign bits in the representation.
For floating point types, the number of radix digits in the mantissa.\(^{192}\)

\[
\text{static constexpr int digits10;}
\]

Number of base 10 digits that can be represented without change.\(^{193}\)

\[
\text{static constexpr int max_digits10;}
\]

Number of base 10 digits required to ensure that values which differ are always differentiated.

\[
\text{static constexpr bool is_signed;}
\]

True if the type is signed.

\[
\text{static constexpr bool is_integer;}
\]

True if the type is integer.

\[
\text{static constexpr bool is_exact;}
\]

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.

\[
\text{static constexpr int radix;}
\]

For floating types, specifies the base or radix of the exponent representation (often 2).\(^{194}\)
For integer types, specifies the base of the representation.\(^{195}\)

\[
\text{static constexpr T epsilon() throw();}
\]

Machine epsilon: the difference between 1 and the least value greater than 1 that is representable.\(^{196}\)

\[
\text{static constexpr T round_error() throw();}
\]

\(^{192}\) Equivalent to FLT\_MANT\_DIG, DBL\_MANT\_DIG, LDBL\_MANT\_DIG.
\(^{193}\) Equivalent to FLT\_DIG, DBL\_DIG, LDBL\_DIG.
\(^{194}\) Equivalent to FLT\_RADIX.
\(^{195}\) Distinguishes types with bases other than 2 (e.g. BCD).
\(^{196}\) Equivalent to FLT\_EPSILON, DBL\_EPSILON, LDBL\_EPSILON.
Measure of the maximum rounding error.\textsuperscript{(197)}

\texttt{static \texttt{const\texttt{constexpr}}} \texttt{int} \texttt{min\_exponent;}

Minimum negative integer such that \texttt{radix} raised to the power of one less than that integer is a normalized floating point number.\textsuperscript{(198)}

Meaningful for all floating point types.

\texttt{static \texttt{const\texttt{constexpr}}} \texttt{int} \texttt{min\_exponent10;}

Minimum negative integer such that 10 raised to that power is in the range of normalized floating point numbers.\textsuperscript{(199)}

Meaningful for all floating point types.

\texttt{static \texttt{const\texttt{constexpr}}} \texttt{int} \texttt{max\_exponent;}

Maximum positive integer such that \texttt{radix} raised to the power one less than that integer is a representable finite floating point number.\textsuperscript{(200)}

Meaningful for all floating point types.

\texttt{static \texttt{const\texttt{constexpr}}} \texttt{int} \texttt{max\_exponent10;}

Maximum positive integer such that 10 raised to that power is in the range of representable finite floating point numbers.\textsuperscript{(201)}

Meaningful for all floating point types.

\texttt{static \texttt{const\texttt{constexpr}}} \texttt{bool} \texttt{has\_infinity;}

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} \texttt{!=} \texttt{false}.

\texttt{static \texttt{const\texttt{constexpr}}} \texttt{bool} \texttt{has\_quiet\_NaN;}

True if the type has a representation for a quiet (non-signaling) “Not a Number.”\textsuperscript{(202)}

Meaningful for all floating point types.

 Shall be \texttt{true} for all specializations in which \texttt{is\_iec559} \texttt{!=} \texttt{false}.

\texttt{static \texttt{const\texttt{constexpr}}} \texttt{bool} \texttt{has\_signaling\_NaN;}

\textsuperscript{(197)} 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.

\textsuperscript{(198)} Equivalent to \texttt{FLT\_MIN\_EXP}, \texttt{DBL\_MIN\_EXP}, \texttt{LDBL\_MIN\_EXP}.

\textsuperscript{(199)} Equivalent to \texttt{FLT\_MIN\_10\_EXP}, \texttt{DBL\_MIN\_10\_EXP}, \texttt{LDBL\_MIN\_10\_EXP}.

\textsuperscript{(200)} Equivalent to \texttt{FLT\_MAX\_EXP}, \texttt{DBL\_MAX\_EXP}, \texttt{LDBL\_MAX\_EXP}.

\textsuperscript{(201)} Equivalent to \texttt{FLT\_MAX\_10\_EXP}, \texttt{DBL\_MAX\_10\_EXP}, \texttt{LDBL\_MAX\_10\_EXP}.

\textsuperscript{(202)} Required by LIA-1.
423 Language support library

18.2 Implementation properties

True if the type has a representation for a signaling "Not a Number."\(^{(203)}\)
Meaningful for all floating point types.
Shall be true for all specializations in which \texttt{is_iec559} != false.

\begin{verbatim}
static constexpr float_denorm_style has_denorm;
\end{verbatim}
\begin{itemize}
\item \texttt{denorm\_present} if the type allows denormalized values (variable number of exponent bits)\(^{(204)}\),
\item \texttt{denorm\_absent} if the type does not allow denormalized values, and
\item \texttt{denorm\_indeterminate} if it is indeterminate at compile time whether the type allows denormalized values.
\end{itemize}
Meaningful for all floating point types.

\begin{verbatim}
static constexpr bool has_denorm_loss;
\end{verbatim}
\begin{itemize}
\item True if loss of accuracy is detected as a denormalization loss, rather than as an inexact result.\(^{(205)}\)
\end{itemize}

\begin{verbatim}
static constexpr T infinity() throw();
\end{verbatim}
\begin{itemize}
\item Representation of positive infinity, if available.\(^{(206)}\)
\item Meaningful for all specializations for which \texttt{has\_infinity} != false. Required in specializations for which \texttt{is_iec559} != false.
\end{itemize}

\begin{verbatim}
static constexpr T quiet_NaN() throw();
\end{verbatim}
\begin{itemize}
\item Representation of a quiet "Not a Number," if available.\(^{(207)}\)
\item Meaningful for all specializations for which \texttt{has\_quiet\_NaN} != false. Required in specializations for which \texttt{is_iec559} != false.
\end{itemize}

\begin{verbatim}
static constexpr T signaling_NaN() throw();
\end{verbatim}
\begin{itemize}
\item Representation of a signaling "Not a Number," if available.\(^{(208)}\)
\item Meaningful for all specializations for which \texttt{has\_signaling\_NaN} != false. Required in specializations for which \texttt{is_iec559} != false.
\end{itemize}

\begin{verbatim}
static constexpr T denorm_min() throw();
\end{verbatim}
\begin{itemize}
\item Minimum positive denormalized value.\(^{(209)}\)
\item Meaningful for all floating point types.
\item In specializations for which \texttt{has\_denorm} == false, returns the minimum positive normalized value.
\end{itemize}

\begin{verbatim}
static constexpr bool is_iec559;
\end{verbatim}
\(^{(203)}\) Required by LIA-1.
\(^{(204)}\) Required by LIA-1.
\(^{(205)}\) See IEC 559.
\(^{(206)}\) See IEC 559.
\(^{(207)}\) Required by LIA-1.
\(^{(208)}\) Required by LIA-1.
\(^{(209)}\) Required by LIA-1.

Draft
18.2 Implementation properties

56 True if and only if the type adheres to IEC 559 standard.\textsuperscript{210}
57 Meaningful for all floating point types.

\texttt{static const\_constexpr bool is\_bounded;}
58 True if the set of values representable by the type is finite.\textsuperscript{211} [\textit{Note: All built-in types are bounded. This member would be false for arbitrary precision types. — end note}]
59 Meaningful for all specializations.

\texttt{static const\_constexpr bool is\_modulo;}
60 True if the type is modulo.\textsuperscript{212} A type is modulo if it is possible to add two positive numbers and have a result that wraps around to a third number that is less.
61 Generally, this is \texttt{false} for floating types, \texttt{true} for unsigned integers, and \texttt{true} for signed integers on most machines.
62 Meaningful for all specializations.

\texttt{static const\_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{213}
64 Meaningful for all specializations.

\texttt{static const\_constexpr bool tinyness\_before;}
65 \texttt{true} if tinyness is detected before rounding.\textsuperscript{214}
66 Meaningful for all floating point types.

\texttt{static const\_constexpr float\_round\_style round\_style;}
67 The rounding style for the type.\textsuperscript{215}
68 Meaningful for all floating point types. Specializations for integer types shall return \texttt{round\_toward\_zero}.

18.2.1.3 Type \texttt{float\_round\_style}

\begin{verbatim}
namespace std {
  enum float_round_style {
    round_indeterminate = -1,
    round_toward_zero   = 0,
    round_to_nearest    = 1,
    round_toward_infinity = 2,
  }
\end{verbatim}

\textsuperscript{210} International Electrotechnical Commission standard 559 is the same as IEEE 754.
\textsuperscript{211} Required by LIA-1.
\textsuperscript{212} Required by LIA-1.
\textsuperscript{213} Required by LIA-1.
\textsuperscript{214} Refer to IEC 559. Required by LIA-1.
\textsuperscript{215} Equivalent to \texttt{FLT\_ROUNDS}. Required by LIA-1.
The rounding mode for floating point arithmetic is characterized by the values:

- `round_indeterminate` if the rounding style is indeterminable
- `round_toward_zero` if the rounding style is toward zero
- `round_to_nearest` if the rounding style is to the nearest representable value
- `round_toward_infinity` if the rounding style is toward infinity
- `round_toward_neg_infinity` if the rounding style is toward negative infinity

18.2.1.4 Type `float_denorm_style`  

```cpp
namespace std {
    enum float_denorm_style {
        denorm_indeterminate = -1,
        denorm_absent = 0,
        denorm_present = 1
    };
}
```

The presence or absence of denormalization (variable number of exponent bits) is characterized by the values:

- `denorm_indeterminate` if it cannot be determined whether or not the type allows denormalized values
- `denorm_absent` if the type does not allow denormalized values
- `denorm_present` if the type does allow denormalized values

18.2.1.5 `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.

```cpp
namespace std {
    template<> class numeric_limits<float> {
        public:
            static constexpr bool is_specialized = true;

            inline static constexpr float min() throw() { return 1.17549435E-38F; }
            inline static constexpr float max() throw() { return 3.40282347E+38F; }
            inline static constexpr float lowest() throw() { return -3.40282347E+38F; }

            static constexpr int digits = 24;
    }
}
```

Draft
Implementation properties

Language support library

```cpp
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() throw() { return 1.19209290E-07F; }
inline static constexpr float round_error() throw() { 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 bool has_denorm = denorm_absent;
static constexpr bool has_denorm_loss = false;
inline static constexpr float infinity() throw() { return ...; }
inline static constexpr float quiet_NaN() throw() { return ...; }
inline static constexpr float signaling_NaN() throw() { return ...; }
inline static constexpr float denorm_min() throw() { 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 `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() throw() { return false; }
      static constexpr bool max() throw() { return true; }
      static constexpr bool lowest() throw() { return false; }

      static constexpr int digits = 1;
      static constexpr int digits10 = 0;
  }
```

Draft
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() throw() { return 0; }
static constexpr bool round_error() throw() { 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() throw() { return 0; }
static constexpr bool quiet_NaN() throw() { return 0; }
static constexpr bool signaling_NaN() throw() { return 0; }
static constexpr bool denorm_min() throw() { 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;
};
}

18.2.2 C Library

Header <climits> (Table 18)\textsuperscript{216}

Table 18: Header <climits> synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values:</td>
<td></td>
</tr>
<tr>
<td>CHAR_BIT</td>
<td>INT_MAX          LONG_MAX    SCHAR_MIN   SHRT_MIN    ULLONG_MAX</td>
</tr>
<tr>
<td>CHAR_MAX</td>
<td>LONGLONG_MAX     LONG_MIN    SCHAR_MAX   UCHAR_MAX   ULONG_MAX</td>
</tr>
<tr>
<td>CHAR_MIN</td>
<td>LONGLONG_MIN     MB_LEN_MAX   SHRT_MAX    UINT_MAX    USHRT_MAX</td>
</tr>
</tbody>
</table>

\textsuperscript{216} LLONG_MAX, LLONG_MIN, and ULLONG_MAX added by TR1.
The contents are the same as the Standard C library header `<limits.h>`. [Note: The types of the constants defined by macros in `<climits>` are not required to match the types to which the macros refer. — end note]

Header `<cfloat>` (Table 19)<sup>217</sup>

<table>
<thead>
<tr>
<th>Values</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBL_DIG</td>
<td>DBL_MIN_EXP</td>
</tr>
<tr>
<td>DBL_EPSILON</td>
<td>DECIMAL_DIG</td>
</tr>
<tr>
<td>DBL_MANT_DIG</td>
<td>FLT_DIG</td>
</tr>
<tr>
<td>DBL_MAX</td>
<td>FLT_EPSILON</td>
</tr>
<tr>
<td>DBL_MAX_10_EXP</td>
<td>FLT_EVAL_METHOD</td>
</tr>
<tr>
<td>DBL_MAX_EXP</td>
<td>FLT_MANT_DIG</td>
</tr>
<tr>
<td>DBL_MIN</td>
<td>FLT_MAX</td>
</tr>
<tr>
<td>DBL_MIN_10_EXP</td>
<td>FLT_MAX_10_EXP</td>
</tr>
</tbody>
</table>

The contents are the same as the Standard C library header `<float.h>`.

SEE ALSO: ISO C subclause 7.1.5, 5.2.4.2.2, 5.2.4.2.1.

18.3 Integer types

18.3.1 Header `<cstdint>` synopsis

```cpp
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;
    typedef unsigned integer type uint64_t;
}
```

<sup>217</sup>DECIMAL_DIG and FLT_EVAL_METHOD added by TR1.
1 The header also defines numerous macros of the form:

    INT_[FAST LEAST]{8 16 32 64}_MIN
    [U]INT_[FAST LEAST]{8 16 32 64}_MAX
    INT{MAX PTR}_MIN
    [U]INT{MAX PTR}_MAX
    {PTRDIFF SIG_ATOMIC WCHAR WINT}{_MAX _MIN}
    SIZE_MAX

plus function macros of the form:

    [U]INT{8 16 32 64 MAX}_C

2 The header defines all functions, types, and macros the same as C99 subclause 7.18. [Note: The macros defined by <stdint> are provided unconditionally. In particular, the symbols __STDC_LIMIT_MACROS and __STDC_CONSTANT_MACROS (mentioned in C99 footnotes 219, 220, and 222) play no role in C++. — end note ]

18.3.2 The header <stdint.h> [stdinth]

1 The header behaves as if it includes the header <cstdint>, and provides sufficient using declarations to declare in the global namespace all type names defined in the header <cstdint>.

18.4 Start and termination [support.start.term]

1 Header <stdlib> (partial), Table 20:

Table 20: Header <stdlib> 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></td>
<td>EXIT_SUCCESS</td>
</tr>
<tr>
<td>Functions:</td>
<td>abort</td>
</tr>
<tr>
<td></td>
<td>atexit</td>
</tr>
<tr>
<td></td>
<td>exit</td>
</tr>
</tbody>
</table>

2 The contents are the same as the Standard C library header <stdlib.h>, with the following changes:

Draft
The function `abort()` has additional behavior in this International Standard:

- The program is terminated without executing destructors for objects of automatic or static storage duration and without calling the functions passed to `atexit()` (3.6.3).

```c
extern "C" int atexit(void (*)(void))
extern "C++" int atexit(void (*)(void))
```

**Effects:** The `atexit()` functions register the function pointed to by `f`, to be called without arguments at normal program termination.

For the execution of a function registered with `atexit()`, if control leaves the function because it provides no handler for a thrown exception, `terminate()` is called (18.7.3.3).

**Implementation limits:** The implementation shall support the registration of at least 32 functions.

**Returns:** The `atexit()` function returns zero if the registration succeeds, nonzero if it fails.

The function `exit()` has additional behavior in this International Standard:

- First, objects with static storage duration are destroyed and functions registered by calling `atexit` are called. Non-local objects with static storage duration are destroyed in the reverse order of the completion of their constructor. (Automatic objects are not destroyed as a result of calling `exit()`.)

  Functions registered with `atexit` are called in the reverse order of their registration, except that a function is called after any previously registered functions that had already been called at the time it was registered. A function registered with `atexit` before a non-local object `obj1` of static storage duration is initialized will not be called until `obj1`'s destruction has completed. A function registered with `atexit` after a non-local object `obj2` of static storage duration is initialized will be called before `obj2`'s destruction starts. A local static object `obj3` is destroyed at the same time it would be if a function calling the `obj3` destructor were registered with `atexit` at the completion of the `obj3` constructor.

- 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.

- 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.

The function `exit()` never returns to its caller.

**See also:** subclauses 3.6, 3.6.3, ISO C subclause 7.10.4.

---

218) 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()`.

219) A function is called for every time it is registered.

220) Any C streams associated with `cin`, `cout`, etc (27.3) are flushed and closed when static objects are destroyed in the previous phase. The function `tmpfile()` is declared in `<cstdio>`.

221) The macros `EXIT_FAILURE` and `EXIT_SUCCESS` are defined in `<cstdlib>`.
18.5 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;
    struct nothrow_t {}
    extern const nothrow_t nothrow;
    typedef void (*new_handler)();
    new_handler set_new_handler(new_handler new_p) throw();
}

void* operator new(std::size_t size) throw(std::bad_alloc);
void* operator new(std::size_t size, const std::nothrow_t&) throw();
void operator delete(void* ptr) throw();
void operator delete(void* ptr, const std::nothrow_t&) throw();
void* operator new[](std::size_t size) throw(std::bad_alloc);
void* operator new[](std::size_t size, const std::nothrow_t&) throw();
void operator delete[](void* ptr) throw();
void operator delete[](void* ptr, const std::nothrow_t&) throw();
void* operator new (std::size_t size, void* ptr) throw();
void* operator new[](std::size_t size, void* ptr) throw();
void operator delete (void* ptr, void*) throw();
void operator delete[](void* ptr, void*) throw();
```

See also: 1.7, 3.7.3, 5.3.4, 5.3.5, 12.5, 20.6.

18.5.1 Storage allocation and deallocation

Except where otherwise specified, the provisions of (3.7.3) apply to the library versions of `operator new` and `operator delete`.

18.5.1.1 Single-object forms

```cpp
void* operator new(std::size_t size) throw(std::bad_alloc);
```

**Effects:** The allocation function (3.7.3.1) called by a `new-expression` (5.3.4) to allocate `size` bytes of storage suitably aligned to represent any object of that size.

**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 (3.7.3), or else throw a `bad_alloc` exception. This requirement is binding on a replacement version of this function.

**Default behavior:**

— Executes a loop: Within the loop, the function first attempts to allocate the requested storage. Whether the attempt involves a call to the Standard C library function `malloc` is unspecified.
18.5 Dynamic memory management

— Returns a pointer to the allocated storage if the attempt is successful. Otherwise, if the last argument to
set_new_handler() was a null pointer, throw bad_alloc.

— Otherwise, the function calls the current new_handler (18.5.2.2). If the called function returns, the loop
repeats.

— The loop terminates when an attempt to allocate the requested storage is successful or when a called new-
handler function does not return.

```cpp
void* operator new(std::size_t size, const std::nothrow_t&) throw();
```

**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 (3.7.3), 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. Other-
wise, returns a null pointer.

[Example:

```cpp
T* p1 = new T; // throws bad_alloc if it fails
T* p2 = new(nothrow) T; // returns 0 if it fails
```

— end example]

```cpp
void operator delete(void* ptr) throw();
void operator delete(void* ptr, const std::nothrow_t&) throw();
```

**Effects:** The deallocation function (3.7.3.2) called by a delete-expression to render the value of ptr invalid.

**Replaceable:** a C++ program may define a function with this function signature that displaces the default version
defined by the C++ Standard library.

**Requires:** the value of ptr shall be null or it 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*).

**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 subse-
quent calls to operator new or any of calloc, malloc, or realloc, declared in <cstdlib>.

```cpp
void operator delete(void* ptr, const std::nothrow_t&) throw();
```

**Effects:** The deallocation function (3.7.3.2) 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 this function signature that displaces the default version defined by the C++ Standard library.

Default behavior: calls `operator delete(ptr)`.

### 18.5.1.2 Array forms

```c
void* operator new[](std::size_t size) throw(std::bad_alloc);
```

**Effects:** The allocation function (3.7.3.1) 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.\(^{222}\)

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)`.

```c
void* operator new[](std::size_t size, const std::nothrow_t&) throw();
```

**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 (3.7.3), 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) throw();
void operator delete[](void* ptr, const std::nothrow_t&) throw();
```

**Effects:** The deallocation function (3.7.3.2) 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 this function signature that displaces the default version defined by the C++ Standard library.

**Requires:** the value of `ptr` shall be null or it 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*)`.

**Default behavior:** Calls `operator delete(ptr)`.

---

\(^{222}\) 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.

Draft
13  Effects: The deallocation function (3.7.3.2) 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.

14  Replaceable: a C++ program may define a function with this function signature that displaces the default version defined by the C++ Standard library.

15  Default behavior: calls operator delete[](ptr).

18.5.1.3 Placement forms

These functions are reserved, a C++ program may not define functions that displace the versions in the Standard C++ library (17.4.3). The provisions of (3.7.3) do not apply to these reserved placement forms of operator new and operator delete.

void* operator new(std::size_t size, void* ptr) throw();

Returns: ptr.

Remarks: Intentionally performs no other action.

Example: This can be useful for constructing an object at a known address:

void* place = operator new(sizeof(Something));
Something* p = new (place) Something();

— end example

void* operator new[](std::size_t size, void* ptr) throw();

Returns: ptr.

Remarks: Intentionally performs no other action.

void operator delete(void* ptr, void*) throw();

Effects: Intentionally performs no action.

Remarks: Default function called when any part of the initialization in a placement new expression that invokes the library’s non-array placement operator new terminates by throwing an exception (5.3.4).

void operator delete[](void* ptr, void*) throw();

Effects: Intentionally performs no action.

Remarks: Default function called when any part of the initialization in a placement new expression that invokes the library’s array placement operator new terminates by throwing an exception (5.3.4).

18.5.2 Storage allocation errors

18.5.2.1 Class bad_alloc
The class `bad_alloc` defines the type of objects thrown as exceptions by the implementation to report a failure to allocate storage.

```cpp
bad_alloc() throw();
```

**Effects:** Constructs an object of class `bad_alloc`.

**Remarks:** The result of calling `what()` on the newly constructed object is implementation-defined.

```cpp
bad_alloc(const bad_alloc&) throw();
bad_alloc& operator=(const bad_alloc&) throw();
```

**Effects:** Copies an object of class `bad_alloc`.

```cpp
virtual const char* what() const throw();
```

**Returns:** An implementation-defined NTBS.

### 18.5.2.2 Type `new_handler`

```cpp
typedef void (*new_handler)();
```

The type of a *handler function* to be called by `operator new()` or `operator new[]( )` (18.5.1) when they cannot satisfy a request for additional storage.

**Required behavior:** A `new_handler` shall perform one of the following:
- make more storage available for allocation and then return;
- throw an exception of type `bad_alloc` or a class derived from `bad_alloc`;
- call either `abort()` or `exit()`;

### 18.5.2.3 set_new_handler

```cpp
new_handler set_new_handler(new_handler new_p) throw();
```

**Effects:** Establishes the function designated by `new_p` as the current `new_handler`.

**Returns:** 0 on the first call, the previous `new_handler` on subsequent calls.
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**

```cpp
namespace std {
    class type_info;
    class bad_cast;
    class bad_typeid;
}
```

See also: 5.2.7, 5.2.8.

### 18.6.1 Class `type_info`

```cpp
namespace std {
    class type_info {
        public:
            virtual ~type_info();
            bool operator==(const type_info& rhs) const; // cannot be copied
            bool operator!=(const type_info& rhs) const;
            bool before(const type_info& rhs) const;
            const char* name() const;

        private:
            type_info(const type_info& rhs) = delete; // cannot be copied
            type_info& operator=(const type_info& rhs) = delete; // cannot be copied
    }
}
```

The class `type_info` describes type information generated by the implementation. Objects of this class effectively store a pointer to a name for the type, and an encoded value suitable for comparing two types for equality or collating order. The names, encoding rule, and collating sequence for types are all unspecified and may differ between programs.

```cpp
bool operator==(const type_info& rhs) const;
```

**Effects:** Compares the current object with `rhs`.

**Returns:** `true` if the two values describe the same type.

```cpp
bool operator!=(const type_info& rhs) const;
```

**Returns:** `!*this == rhs`.

```cpp
bool before(const type_info& rhs) const;
```

**Effects:** Compares the current object with `rhs`.

**Returns:** `true` if `*this precedes rhs` in the implementation’s collation order.

```cpp
const char* name() const;
```
437 Language support library 18.6 Type identification

7 Returns: an implementation-defined NTBS.

8 Remarks: The message may be a null-terminated multibyte string (17.3.2.1.3.2), suitable for conversion and display as a wstring (21.2, 22.2.1.4)

9 type_info(const type_info& rhs);
type_info& operator=(const type_info& rhs);

9 Effects: Copies a type_info object.

9 Remarks: Since the copy constructor and assignment operator for type_info are private to the class, objects of this type cannot be copied.

18.6.2 Class bad_cast [bad.cast]

namespace std {
    class bad_cast : public exception {
        public:
            bad_cast() throw();
            bad_cast(const bad_cast&) throw();
            bad_cast& operator=(const bad_cast&) throw();
            virtual const char* what() const throw();
    };
}

1 The class bad_cast defines the type of objects thrown as exceptions by the implementation to report the execution of an invalid dynamic-cast expression (5.2.7).

    bad_cast() throw();

2 Effects: Constructs an object of class bad_cast.

3 Remarks: The result of calling what() on the newly constructed object is implementation-defined.

    bad_cast(const bad_cast&) throw();
    bad_cast& operator=(const bad_cast&) throw();

4 Effects: Copies an object of class bad_cast.

    virtual const char* what() const throw();

5 Returns: An implementation-defined NTBS.

6 Remarks: The message may be a null-terminated multibyte string (17.3.2.1.3.2), suitable for conversion and display as a wstring (21.2, 22.2.1.4)

18.6.3 Class bad_typeid [bad.typeid]

namespace std {
    class bad_typeid : public exception {
        public:
            bad_typeid() throw();

Draft
18.7 Exception handling

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).

```cpp
bad_typeid() throw();

Effects: Constructs an object of class `bad_typeid`.

Remarks: The result of calling `what()` on the newly constructed object is implementation-defined.
```
bad_typeid(const bad_typeid&) throw();
bad_typeid& operator=(const bad_typeid&) throw();
```

```cpp
virtual const char* what() const throw();
```

Returns: An implementation-defined NTBS.

Remarks: The message may be a null-terminated multibyte string (17.3.2.1.3.2), suitable for conversion and display as a `wstring` (21.2, 22.2.1.4)

18.7 Exception handling [support.exception]

The header `<exception>` defines several types and functions related to the handling of exceptions in a C++ program.

**Header `<exception>` synopsis**

```cpp
namespace std {
    class exception;
    class bad_exception;

    typedef void (*unexpected_handler)();
    unexpected_handler set_unexpected(unexpected_handler f) throw();
    void unexpected();

    typedef void (*terminate_handler)();
    terminate_handler set_terminate(terminate_handler f) throw();
    void terminate();

    bool uncaught_exception() throw();

    typedef unspecified_exception_ptr;

    exception_ptr current_exception();
    void rethrow_exception(exception_ptr p);
    template<class E> exception_ptr copy_exception(E e);
}
```

Draft
namespace std {
    class exception {
    public:
        exception() throw();
        exception(const exception&) throw();
        exception& operator=(const exception&) throw();
        virtual ~exception() throw();
        virtual const char* what() const throw();
    }
}

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.

exception() throw();

Effects: Constructs an object of class exception.
Remarks: Does not throw any exceptions.

exception(const exception&) throw();
exception& operator=(const exception&) throw();

Effects: Copies an exception object.
Remarks: The effects of calling what() after assignment are implementation-defined.

virtual ~exception() throw();

Effects: Destroys an object of class exception.
Remarks: Does not throw any exceptions.

virtual const char* what() const throw();

Returns: An implementation-defined NTBS.
Remarks: The message may be a null-terminated multibyte string (17.3.2.1.3.2), suitable for conversion and display as a wstring (21.2, 22.2.1.4). 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.

18.7.2 Violating exception-specification s

18.7.2.1 Class bad_exception
namespace std {
    class bad_exception : public exception {
    public:
        bad_exception() throw();
        bad_exception(const bad_exception&) throw();
        bad_exception& operator=(const bad_exception&) throw();
        virtual const char* what() const throw();
    };
}

The class bad_exception defines the type of objects thrown as described in (15.5.2).

bad_exception() throw();

**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&) throw();

**Effects:** Copies an object of class bad_exception.

virtual const char* what() const throw();

**Returns:** An implementation-defined NTBS.

**Remarks:** The message may be a null-terminated multibyte string (17.3.2.1.3.2), suitable for conversion and display as a wstring (21.2, 22.2.1.4).

18.7.2.2 Type unexpected_handler

typedef void (*unexpected_handler)();

**Type of a handler function** to be called by unexpected() when a function attempts to throw an exception not listed in its exception-specification.

**Required behavior:** An unexpected_handler shall not return. See also 15.5.2.

**Default behavior:** The implementation’s default unexpected_handler calls terminate().

18.7.2.3 set_unexpected

unexpected_handler set_unexpected(unexpected_handler f) throw();

**Effects:** Establishes the function designated by f as the current unexpected_handler.

**Requires:** f shall not be a null pointer.

**Returns:** The previous unexpected_handler.
18.7.2.4 unexpected

```c
void unexpected();
```

Called by the implementation when a function exits via an exception not allowed by its `exception-specification` (15.5.2). May also be called directly by the program.

*Effects:* Calls the `unexpected_handler` function in effect immediately after evaluating the `throw-expression` (18.7.2.2), if called by the implementation, or calls the current `unexpected_handler`, if called by the program.

18.7.3 Abnormal termination

18.7.3.1 Type terminate_handler

```c
typedef void (*terminate_handler)();
```

The type of a `handler function` to be called by `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()`.

18.7.3.2 set_terminate

```c
terminate_handler set_terminate(terminate_handler f) throw();
```

*Effects:* Establishes the function designated by `f` as the current handler function for terminating exception processing.

*Requires:* `f` shall not be a null pointer.

*Returns:* The previous `terminate_handler`.

18.7.3.3 terminate

```c
void terminate();
```

Called by the implementation when exception handling must be abandoned for any of several reasons (15.5.1). May also be called directly by the program.

*Effects:* Calls the `terminate_handler` function in effect immediately after evaluating the `throw-expression` (18.7.3.1), if called by the implementation, or calls the current `terminate_handler` function, if called by the program.

18.7.4 uncaught_exception

```c
bool uncaught_exception() throw();
```

Draft
18.7 Exception handling

Returns: true after completing evaluation of a throw-expression until either completing initialization of the exception-declaration in the matching handler or entering unexpected() due to the throw; or after entering terminate() for any reason other than an explicit call to terminate(). [Note: This includes stack unwinding (15.2). — end note]

Remarks: When uncaught_exception() is true, throwing an exception can result in a call of terminate() (15.5.1).

18.7.5 Exception Propagation

typedef unspecified exception_ptr;

The type exception_ptr can be used to refer to an exception object.

exception_ptr shall be DefaultConstructible, CopyConstructible, Assignable and EqualityComparable. exception_ptr’s operations shall not throw exceptions.

Two objects 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. The null value is equivalent only to itself.

An object of type exception_ptr can be compared for equality with a null pointer constant and assigned a null pointer constant. The effect shall be as if exception_ptr() had been used in place of the null pointer constant.

[Note: An implementation might use a reference-counted smart pointer as exception_ptr. — end note]

exception_ptr current_exception();

Returns: An exception_ptr object that refers to the currently handled exception or a copy of the currently handled exception, or a null exception_ptr object if no exception is being handled. 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]

Throws: nothing.

void rethrow_exception(exception_ptr p);

Requires: p shall not be null.

Throws: the exception object to which p refers.

template<class E> exception_ptr copy_exception(E e);

Effects: as if

try {
    throw e;
} catch(...)
12  

{ 
  return current_exception(); 
} 

[Note: this function is provided for convenience and efficiency reasons. — end note]

18.8 Other runtime support  

Headers <stdarg> (variable arguments), <csetjmp> (nonlocal jumps), <ctime> (system clock clock(), time()), <csignal> (signal handling), <cstdlib> (runtime environment getenv(), system()), and <cstdbool> (true_false_are_defined).

### Table 21: Header <stdarg> synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td>va_arg va_end va_start va_copy</td>
</tr>
<tr>
<td>Type:</td>
<td>va_list</td>
</tr>
</tbody>
</table>

### Table 22: Header <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 23: Header <ctime> synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro:</td>
<td>CLOCK_PER_SEC</td>
</tr>
<tr>
<td>Type:</td>
<td>clock_t</td>
</tr>
<tr>
<td>Function:</td>
<td>clock</td>
</tr>
</tbody>
</table>

The contents of these headers are the same as the Standard C library headers <stdarg.h>, <setjmp.h>, <time.h>, <signal.h>, and <stdlib.h> respectively, with the following changes:

The restrictions that ISO C places on the second parameter to the va_start() macro in header <stdarg.h> are different in this International Standard. The parameter paramN is the identifier of the rightmost parameter in the variable parameter list of the function definition (the one just before the ...). If the parameter paramN is declared with a function, array, or reference type, or with 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.


Note that va_start is required to work as specified even if unary operator& is overloaded for the type of paramN.
### 18.8 Other runtime support

Language support library 444

#### Table 24: Header `<csignal>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td>SIGABRT SIGILL SIGSEGV SIG_DFL</td>
</tr>
<tr>
<td></td>
<td>SIG_IGN SIGFPE SIGINT SIGTERM SIG_ERR</td>
</tr>
<tr>
<td>Type:</td>
<td>sig_atomic_t</td>
</tr>
<tr>
<td>Functions:</td>
<td>raise signal</td>
</tr>
</tbody>
</table>

#### Table 25: Header `<cstdlib>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions:</td>
<td>getenv system</td>
</tr>
</tbody>
</table>

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 destroy any automatic objects.

See also: ISO C subclause 7.10.4, 7.8, 7.6, 7.12.

5 The header `<cstdbool>` and the header `<stdbool.h>` shall not define macros named `bool`, `true`, and `false`.

6 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 indirectly use any function that is not a POF. All signal handlers shall have C linkage. A POF that could be used as a signal handler in a conforming C program does not produce undefined behavior when used as a signal handler in a C++ program. The behavior of any other function used as a signal handler in a C++ program is implementation-defined.224)

#### Table 26: Header `<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>

224) In particular, a signal handler using exception handling is very likely to have problems.
Chapter 19  Diagnostics library

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 27.

Table 27: Diagnostics library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.1 Exception classes</td>
<td>&lt;stdexcept&gt;</td>
</tr>
<tr>
<td>19.2 Assertions</td>
<td>&lt;cassert&gt;</td>
</tr>
<tr>
<td>19.3 Error numbers</td>
<td>&lt;cerrno&gt;</td>
</tr>
<tr>
<td>19.4 System error support</td>
<td>&lt;system_error&gt;</td>
</tr>
</tbody>
</table>

19.1 Exception classes

The Standard C++ library provides classes to be used to report certain errors (17.4.4.8) 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;
}
```
19.1 Exception classes

19.1.1 Class logic_error

namespace std {
    class logic_error : public exception {
    public:
        explicit logic_error(const string& what_arg);
        explicit logic_error(const char* what_arg);
    }
}

The class logic_error defines the type of objects thrown as exceptions to report errors presumably detectable before the program executes, such as violations of logical preconditions or class invariants.

logic_error(const string& what_arg);

Effects: Constructs an object of class logic_error.

Postcondition: strcmp(what(), what_arg.c_str()) == 0.

logic_error(const char* what_arg);

Effects: Constructs an object of class logic_error.

Postcondition: strcmp(what(), what_arg) == 0.

19.1.2 Class domain_error

namespace std {
    class domain_error : public logic_error {
    public:
        explicit domain_error(const string& what_arg);
        explicit domain_error(const char* what_arg);
    }
}

The class domain_error defines the type of objects thrown as exceptions by the implementation to report domain errors.

domain_error(const string& what_arg);

Effects: Constructs an object of class domain_error.

Postcondition: strcmp(what(), what_arg.c_str()) == 0.

domain_error(const char* what_arg);

Effects: Constructs an object of class domain_error.

Postcondition: strcmp(what(), what_arg) == 0.
19.1.3 Class invalid_argument

namespace std {
    class invalid_argument : public logic_error {
        public:
            explicit invalid_argument(const string& what_arg);
            explicit invalid_argument(const char* what_arg);
    };
}

The class invalid_argument defines the type of objects thrown as exceptions to report an invalid argument.

invalid_argument(const string& what_arg);

Effects: Constructs an object of class invalid_argument.

Postcondition: strcmp(what(), what_arg.c_str()) == 0.

invalid_argument(const char* what_arg);

Effects: Constructs an object of class invalid_argument.

Postcondition: strcmp(what(), what_arg) == 0.

19.1.4 Class length_error

namespace std {
    class length_error : public logic_error {
        public:
            explicit length_error(const string& what_arg);
            explicit length_error(const char* what_arg);
    };
}

The class length_error defines the type of objects thrown as exceptions to report an attempt to produce an object whose length exceeds its maximum allowable size.

length_error(const string& what_arg);

Effects: Constructs an object of class length_error.

Postcondition: strcmp(what(), what_arg.c_str()) == 0.

length_error(const char* what_arg);

Effects: Constructs an object of class length_error.

Postcondition: strcmp(what(), what_arg) == 0.

19.1.5 Class out_of_range

Draft
19.1 Exception classes

namespace std {
  class out_of_range : public logic_error {
  public:
    explicit out_of_range(const string& what_arg);
    explicit out_of_range(const char* what_arg);
  }
}

The class out_of_range defines the type of objects thrown as exceptions to report an argument value not in its expected range.

out_of_range(const string& what_arg);
  
  Effects: Constructs an object of class out_of_range.

Postcondition: strcmp(what(), what_arg.c_str()) == 0.

out_of_range(const char* what_arg);
  
  Effects: Constructs an object of class out_of_range.

Postcondition: strcmp(what(), what_arg) == 0.

19.1.6 Class runtime_error

namespace std {
  class runtime_error : public exception {
  public:
    explicit runtime_error(const string& what_arg);
    explicit runtime_error(const char* what_arg);
  }
}

The class runtime_error defines the type of objects thrown as exceptions to report errors presumably detectable only when the program executes.

runtime_error(const string& what_arg);
  
  Effects: Constructs an object of class runtime_error.

Postcondition: strcmp(what(), what_arg.c_str()) == 0.

runtime_error(const char* what_arg);
  
  Effects: Constructs an object of class runtime_error.

Postcondition: strcmp(what(), what_arg) == 0.

19.1.7 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);
    };
}

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.

Postcondition: strcmp(what(), what_arg.c_str()) == 0.

range_error(const char* what_arg);

Effects: Constructs an object of class range_error.

Postcondition: strcmp(what(), what_arg) == 0.

19.1.8 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);
    };
}

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.

Postcondition: strcmp(what(), what_arg.c_str()) == 0.

overflow_error(const char* what_arg);

Effects: Constructs an object of class overflow_error.

Postcondition: strcmp(what(), what_arg) == 0.

19.1.9 Class underflow_error

namespace std {
    class underflow_error : public runtime_error {
        public:
            explicit underflow_error(const string& what_arg);
    };
    
    
Draft
19.2 Assertions

```cpp
explicit underflow_error(const char* what_arg);
};
}
```

The class `underflow_error` defines the type of objects thrown as exceptions to report an arithmetic underflow error.

```
underflow_error(const string& what_arg);
```

**Effects:** Constructs an object of class `underflow_error`.

**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`.

19.2 Assertions

The header `<cassert>`, described in (Table 28), provides a macro for documenting C++ program assertions and a mechanism for disabling the assertion checks.

Table 28: 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>

The contents are the same as the Standard C library header `<assert.h>`.

SEE ALSO: ISO C subclause 7.2.

19.3 Error numbers

The header `<cerrno>` is described in (Table 29). Its contents are the same as the POSIX header `<errno.h>`, except that `errno` shall be defined as a macro, and an additional macro `EOTHER` shall be defined to represent errors not specified by the POSIX standard. [Note: The intent is to remain in close alignment with the POSIX standard. — end note]
451     Diagnostics library

19.4     System error support

Table 29: Header <errno> synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td>ECONNREFUSED, EIO, ENODEV, ENOTEMPTY, EPROTOTYPE</td>
</tr>
<tr>
<td>E2BIG</td>
<td>ECONNREFUSED, EISCONN, ENOENT, ENOTRECOVERABLE, ERANGE</td>
</tr>
<tr>
<td>EACCES</td>
<td>EDEADLK, EISDIR, ENOEXEC, ENOTSOCK, EROFS</td>
</tr>
<tr>
<td>EADDRINUSE</td>
<td>EDESTADDRREQ, ELOOP, ENOLCK, ENOTSUP, ESPPIPE</td>
</tr>
<tr>
<td>EADDRNOTAVAIL</td>
<td>EDOM, EMFILE, ENOLINK, ENOTTY, ESRCH</td>
</tr>
<tr>
<td>EAFNOSUPPORT</td>
<td>EEXIST, EMLINK, ENOMEM, EXIO, ETIME</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>EFAULT, EMSGSIZE, ENOMSG, EOPNOTSUPP, ETIMEDOUT</td>
</tr>
<tr>
<td>EALREADY</td>
<td>EFBIG, ENAMETOOLONG, ENOPROTOOPT, EOTHER, ETXTBSY</td>
</tr>
<tr>
<td>EBADF</td>
<td>EHOSTUNREACH, ENETDOW, ENOSPC, EOVERFLOW, EWOULDBLOCK</td>
</tr>
<tr>
<td>EBADMSG</td>
<td>EIDRM, ENETRESET, ENOSR, EOWNERDEAD, EXDEV</td>
</tr>
<tr>
<td>EBUSY</td>
<td>EILSEQ, ENETUNREACH, ENOSTR, EPERM, errno</td>
</tr>
<tr>
<td>ECANCELED</td>
<td>EINPROGRESS, ENFILE, ENOSYS, EPIPE</td>
</tr>
<tr>
<td>ECHILD</td>
<td>EINTR, ENOBUFS, ENOTCONN, EPROTO</td>
</tr>
<tr>
<td>ECONNABORTED</td>
<td>EINVAL, ENODATA, ENOTDIR, EPROTONOSUPPORT</td>
</tr>
</tbody>
</table>

See also: ISO C subclause 7.1.4, 7.2, Amendment 1 subclause 4.3.

19.4 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 (19.3). Implementations are encouraged but not required to leave unchanged the error states provided by other libraries.

Header <system_error> synopsis

    namespace std {

    class system_error;
    class error_code;
    class error_category;

    enum posix_errno {
        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,
    }

Draft
connection_refused = ECONNREFUSED,
connection_reset = ECONNRESET,
cross_device_link = EXDEV,
destination_address_required = EDESTADDRREQ,
device_or_resource_busy = EBUSY,
directory_not_empty = ENOTEMPTY,
executive_format_error = ENOEXEC,
file_exists = EXIST,
file_too_large = EFBIG,
filename_too_long = ENAMETOOLONG,
function_not_supported = ENOSYS,
host_unreachable = EHOSTUNREACH,
identifier_removed = EIDRM,
illegal_byte_sequence = EILSEQ,
inappropriate_io_control_operation = ENOTTY,
interrupted = EINTR,
invalid_argument = EINVAL,
invalid_seek = ESPIPE,
io_error = EIO,
is_a_directory = EISDIR,
message_size = EMSGSIZE,
network_down = ENETDOW,
network_reset = ENETRESET,
network_unreachable = ENETUNREACH,
no_buffer_space = ENOBUS,
no_child_process = ECHILD,
no_link = ENOLINK,
no_lock_available = ENOLCK,
no_message_available = ENODATA,
no_message = ENOMSG,
no_protocol_option = ENOPROTOO,
no_space_on_device = ENOSPC,
no_stream_resources = ENOSR,
no_such_device_or_address = ENXIO,
no_such_device = ENODEV,
no_such_file_or_directory = ENOENT,
no_such_process = ESRCCH,
ot_a_directory = ENOTDIR,
ot_a_socket = ENOTSUCK,
ot_a_stream = ENOTSTR,
ot_connected = ENOTCONN,
not_enough_memory = ENOMEM,
not_supported = ENOTSUP,
operation_canceled = ECANCELED,
operation_in_progress = EINPROGRESS,
operation_not_permitted = EPERM,
operation_not_supported = ENPRETP,
operation_would_block = EWOULDBLOCK,
other = EOTHER,
owner_dead = EOWNERDEAD,
permission_denied = EACCES,
protocol_error = EPROTONOSUPPORT,
read_only_file_system = EROFS,
resource_deadlock_would_occur = EDEADLK,
resource_unavailable_try_again = EAGAIN,
result_out_of_range = ERANGE,
state_not_recoverable = ENOTRECOVERABLE,
stream_timeout = ETIME,
text_file_busy = ETXTBSY,
timed_out = ETIMEDOUT,
too_many_files_open_in_system = ENFILE,
too_many_files_open = EMFILE,
too_many_links = EMLINK,
too_many_symbolic_link_levels = ELOOP,
value_too_large = EOVERFLOW,
wrong_protocol_type = EPROTOTYPE
};

bool operator==(const error_code& ec, posix_errno en);
bool operator==(posix_errno en, const error_code& ec);
bool operator!=(const error_code& ec, posix_errno en);
bool operator!=(posix_errno en, const error_code& ec);

namespace std {

    class error_category {
        public:
            virtual const string& name() const = 0;
            virtual posix_errno posix(error_code::value_type ev) const = 0;
            virtual string message(error_code::value_type ev) const = 0;
            virtual wstring wmessage(error_code::value_type ev) const = 0;
            
            bool operator==(const error_category& rhs) const;
            bool operator!=(const error_category& rhs) const;
    };

    extern const error_category& posix_category;

} // namespace std

19.4.1 Class error_category

19.4.1.1 Class error_category 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:
        virtual const string& name() const = 0;
        virtual posix_errno posix(error_code::value_type ev) const = 0;
        virtual string message(error_code::value_type ev) const = 0;
        virtual wstring wmessage(error_code::value_type ev) const = 0;

        bool operator==(const error_category& rhs) const;
        bool operator!=(const error_category& rhs) const;
    };

extern const error_category& posix_category;

Draft
extern const error_category& native_category;

} // namespace std

19.4.1.2 Class error_category virtual members

virtual const string& name() const = 0;

Returns: A string naming the error category.

Throws: Nothing.

virtual posix_errno posix(error_code::value_type ev) const = 0;

Returns: A value of type posix_errno that corresponds to ev if such a corresponding POSIX error number exists, otherwise other. [Note: Since the possible values of ev are not bounded, the intent is that implementations translate commonly encountered values to the equivalent POSIX error number and translate the rest to other. — end note]

Throws: Nothing.

virtual string message(error_code::value_type ev) const = 0;
virtual wstring wmessage(error_code::value_type ev) const = 0;

Returns: A string that describes the error condition denoted by ev. [Note: The intent is to return a locale-specific string that describes the error corresponding to ev. — end note]

Throws: Nothing.

19.4.1.3 Class error_category non-virtual members

bool operator==(const error_category& rhs) const;

Returns: this == &rhs.

bool operator!=(const error_category& rhs) const;

Returns: !(this == rhs).

19.4.1.4 Error category objects

extern const error_category& posix_category;
extern const error_category& native_category;

The predefined objects posix_category and native_category identify portable and native error codes, respectively.
19.4.2 Class `error_code`  

19.4.2.1 Class `error_code` overview

The class `error_code` describes an object used to hold error code values, such as those originating from the operating system or other low-level application program interfaces. [Note: Class `error_code` is an adjunct to error reporting by exception. — end note]

```cpp
namespace std {

    class error_code {
        public:
            typedef int_least32_t value_type;

            // constructors:
            error_code();
            error_code(value_type val, const error_category& cat);
            error_code(posix_errno val);

            // modifiers:
            void assign(value_type val, const error_category& cat);
            error_code& operator=(posix_errno val);
            void clear();

            // observers:
            value_type value() const;
            const error_category& category() const;
            posix_errno posix() const;
            string message() const;
            wstring wmessage() const;
            operator unspecified-bool-type() const;

            // relational operators:
            bool operator==(const error_code& rhs) const;
            bool operator!=(const error_code& rhs) const;

        private:
            value_type val_; // exposition only
            const error_category& cat_; // exposition only
    };

    template <class charT, class traits>
    basic_ostream<charT,traits>&
    operator<<(basic_ostream<charT,traits>& os, const error_code& ec);

    size_t hash_value(const error_code& ec);
}
```

19.4.2.2 Class `error_code` constructors
19.4 System error support

error_code();

Effects: Constructs an object of type error_code.
Postconditions: val_ == 0 and cat_ == posix_category.
Throws: Nothing.

error_code(value_type val, const error_category& cat);

Effects: Constructs an object of type error_code.
Postconditions: val_ == val and cat_ == cat.
Throws: Nothing.

error_code(posix_errno val);

Effects: Constructs an object of type error_code.
Postconditions: val_ == static_cast<value_type>(val) and cat_ == posix_category.
Throws: Nothing.

19.4.2.3 Class error_code modifiers

void assign(value_type val, const error_category& cat);

Postconditions: val_ == val and cat_ == cat.
Throws: Nothing.

error_code& operator=(posix_errno val);

Postconditions: val_ == static_cast<value_type>(val) and cat_ == posix_category.
Throws: Nothing.

19.4.2.4 Class error_code observers

value_type value() const;

Returns: val_.
Throws: Nothing.

cost error_category& category() const;

Returns: cat_.
Throws: Nothing.

posix_errno posix() const;

Draft
Returns: category().posix(value).

Throws: Nothing.

string message() const;
Returns: category().message(value()).

Throws: Nothing.

wstring wmessage() const;
Returns: category().wmessage(value()).

Throws: Nothing.

operator unspecified-bool-type() const;
Returns: If value() != value_type(), returns a value that will evaluate true in a boolean context; otherwise, returns a value that will evaluate false. The return type shall not be convertible to int.

Throws: Nothing.

[Note: This conversion can be used in contexts where a bool is expected (e.g., an if condition); however, implicit conversions (e.g., to int) that can occur with bool are not allowed, eliminating some sources of user error. One possible implementation choice for this type is pointer to member. —end note]

19.4.2.5 Class error_code relational operators

bool operator==(const error_code& rhs) const;
Returns: value() == rhs.value() && category() == rhs.category().

Throws: Nothing.

bool operator!=(const error_code& rhs) const;
Returns: !(*this == rhs).

throws: Nothing.

bool operator==(const error_code& ec, posix_errno en);
bool operator==(posix_errno en, const error_code& ec);
Returns: ec.value() == static_cast<error_code::value_type>(en) && ec.category() == posix_category.

Throws: Nothing.

bool operator!=(const error_code& ec, posix_errno en);
bool operator!=(posix_errno en, const error_code& ec);
Returns: !(ec == en).

Throws: Nothing.
19.4 System error support

19.4.2.6 Class error_code non-member functions

template <class charT, class traits>
basic_ostream<charT,traits>&
operator<<(basic_ostream<charT,traits>& os, const error_code& ec);

Effects: os << ec.category().name() << ':' << ec.value().

size_t hash_value(const error_code& ec);

Returns: A hash value representing ec.

19.4.3 Class system_error

19.4.3.1 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.

namespace std {

    class system_error : public runtime_error {
    public:
        system_error(error_code ec, const string& what_arg);
        system_error(error_code::value_type ev, const error_category& ecat,
                     const string& what_arg);
        const error_code& code() const throw();
        const char* what() const throw();
    };
}

19.4.3.2 Class system_error members

system_error(error_code ec, const string& what_arg);

Effects: Constructs an object of class system_error.

Postconditions: code() == ec and strcmp(runtime_error::what(), what_arg.c_str()) == 0.

system_error(error_code::value_type ev, const error_category& ecat,
              const string& what_arg);

Effects: Constructs an object of class system_error.

Postconditions: code() == error_code(ev, ecat) and strcmp(runtime_error::what(), what_arg.c_str()) == 0.

const error_code& code() const;

Returns: ec or error_code(ev, ecat), from the constructor, as appropriate.

const char *what() const;
Returns: An NTBS incorporating `runtime_error::what()` and `code().message()`.

[Note: One possible implementation would be:

```cpp
if (msg.empty())
{
    try
    {
        std::string tmp = runtime_error::what();
        if (code())
        {
            if (!tmp.empty())
                tmp += "\": ";
            tmp += code().message();
        }
        swap(msg, tmp);
    }
    catch(...) {}
    return runtime_error::what();
}
return msg.c_str();
```

—end note]
Chapter 20  General utilities library

This clause describes components used by other elements of the Standard C++ library. These components may also be used by C++ programs.

The following clauses describe utility and allocator requirements, utility components, tuples, type traits templates, function objects, dynamic memory management utilities, and date/time utilities, as summarized in Table 30.

Table 30: General utilities library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.1 Requirements</td>
<td></td>
</tr>
<tr>
<td>20.2 Utility components</td>
<td>&lt;utility&gt;</td>
</tr>
<tr>
<td>20.3 Tuples</td>
<td>&lt;tuple&gt;</td>
</tr>
<tr>
<td>20.4 Type traits</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>20.5 Function objects</td>
<td>&lt;functional&gt;</td>
</tr>
<tr>
<td>20.6 Memory</td>
<td>&lt;memory&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cstring&gt;</td>
</tr>
<tr>
<td>20.7 Date and time</td>
<td>&lt;ctime&gt;</td>
</tr>
</tbody>
</table>

20.1 Requirements

20.1.1 describes requirements on types used to instantiate templates defined in the C++ Standard Library. 20.1.2 describes the requirements on storage allocators.

20.1.1 Template argument requirements

The template definitions in the C++ Standard Library refer to various named requirements whose details are set out in tables 31–38. In these tables, T is a type to be supplied by a C++ program instantiating a template; a, b, and c are values of type const T; s and t are modifiable lvalues of type T; u is a value of type (possibly const) T; and rv is a non-const rvalue of type T.

In general, a default constructor is not required. Certain container class member function signatures specify the default constructor 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 31: EqualityComparable requirements \textbf{[equalitycomparable]}

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a == b )</td>
<td>convertible to bool</td>
<td>( == ) is an equivalence relation, that is, it has the following properties:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— For all ( a, a == a ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— If ( a == b ), then ( b == a ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— If ( a == b ) and ( b == c ), then ( a == c ).</td>
</tr>
</tbody>
</table>

Table 32: LessThanComparable requirements \textbf{[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 (25.3)</td>
</tr>
</tbody>
</table>

Table 33: MoveConstructible requirements \textbf{[moveconstructible]}

<table>
<thead>
<tr>
<th>expression</th>
<th>post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t = \text{rv} )</td>
<td>( t ) is equivalent to the value of ( \text{rv} ) before the construction</td>
</tr>
<tr>
<td></td>
<td>[\textit{Note: There is no requirement on the value of ( \text{rv} ) after the construction. — end note} ]</td>
</tr>
</tbody>
</table>

Table 34: CopyConstructible requirements \textbf{[copyconstructible]}

<table>
<thead>
<tr>
<th>expression</th>
<th>post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t = u )</td>
<td>the value of ( u ) is unchanged and is equivalent to ( t )</td>
</tr>
<tr>
<td></td>
<td>[\textit{Note: A type that satisfies the CopyConstructible requirements also satisfies the MoveConstructible requirements. — end note} ]</td>
</tr>
</tbody>
</table>

Table 35: MoveAssignable requirements \textbf{[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 = \text{rv} )</td>
<td>T&amp;</td>
<td>( t )</td>
<td>( t ) is equivalent to the value of ( \text{rv} ) before the assignment</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[\textit{Note: There is no requirement on the value of ( \text{rv} ) after the assignment. — end note} ]</td>
</tr>
</tbody>
</table>
Table 36: CopyAssignable requirements [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 = u</td>
<td>T&amp;</td>
<td>t</td>
<td>t is equivalent to u, the value of u is unchanged</td>
</tr>
</tbody>
</table>

[Note: A type that satisfies the CopyAssignable requirements also satisfies the MoveAssignable requirements. — end note]

Table 37: Swappable requirements [swappable]

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>swap(s,t)</td>
<td>void</td>
<td>t has the value originally held by u, and u has the value originally held by t</td>
</tr>
</tbody>
</table>

The Swappable requirement is met by satisfying one or more of the following conditions:

— T is Swappable if T satisfies the CopyConstructible requirements (Table 34) and the CopyAssignable requirements (Table 36);
— T is Swappable if a namespace scope function named swap exists in the same namespace as the definition of T, such that the expression swap(t,u) is valid and has the semantics described in this table.

Table 38: 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>

20.1.2 Allocator requirements [allocator.requirements]

1 The library describes a standard set of requirements for allocators, which are 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 containers (clause 23) are parameterized in terms of allocators.

2 Table 39 describes the requirements on types manipulated through allocators. All the operations on the allocators are expected to be amortized constant time. Table 40 describes the requirements on allocator types.

Table 39: Descriptive variable definitions

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>T, U</td>
<td>any non-const, non-reference type</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>
</tbody>
</table>
### Variable | Definition
--- | ---
\( t \)  | a value of type `const T&`
\( a, a1, a2 \)  | values of type `X&`
\( b \)  | a value of type `Y`
\( p \)  | a value of type `X::pointer`, obtained by calling `a1.allocate`, where `a1 == a`
\( q \)  | a value of type `X::const_pointer` obtained by conversion from a value `p`.
\( r \)  | a value of type `X::reference` obtained by the expression `*p`.
\( s \)  | a value of type `X::const_reference` obtained by the expression `*q` or by conversion from a value `r`.
\( u \)  | a value of type `Y::const_pointer` obtained by calling `Y::allocate`, or else 0.
\( v \)  | a value of type `V`
\( n \)  | a value of type `X::size_type`. 
Table 40: Allocator requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>assertion/note pre/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::pointer</td>
<td>Pointer to T.</td>
<td></td>
</tr>
<tr>
<td>X::const_pointer</td>
<td>Pointer to const T.</td>
<td></td>
</tr>
<tr>
<td>X::reference</td>
<td>T&amp;</td>
<td></td>
</tr>
<tr>
<td>X::const_reference</td>
<td>T const&amp;</td>
<td></td>
</tr>
<tr>
<td>X::value_type</td>
<td>Identical to T</td>
<td></td>
</tr>
<tr>
<td>X::size_type</td>
<td>unsigned integral</td>
<td>a type that can represent the size of the largest object in the allocation model.</td>
</tr>
<tr>
<td>X::difference_type</td>
<td>signed integral type</td>
<td>a type that can represent the difference between any two pointers in the allocation model.</td>
</tr>
<tr>
<td>typename X::template rebind&lt;U&gt;::other</td>
<td>Y</td>
<td>For all U (including T), Y::template rebind&lt;T&gt;::other is X.</td>
</tr>
<tr>
<td>a.address(r)</td>
<td>X::pointer</td>
<td></td>
</tr>
<tr>
<td>a.address(s)</td>
<td>X::const_pointer</td>
<td></td>
</tr>
<tr>
<td>a.allocate(n)</td>
<td>X::pointer</td>
<td>Memory is allocated for n objects of type T but objects are not constructed. allocate may raise an appropriate exception. The result is a random access iterator.(^{225}) [Note: If n == 0, the return value is unspecified. — end note]</td>
</tr>
<tr>
<td>a.allocate(n,u)</td>
<td>X::pointer</td>
<td></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 null. — end note]</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>
</tr>
<tr>
<td>a1 == a2</td>
<td>bool</td>
<td>returns true iff storage allocated from each can be deallocated via the other.</td>
</tr>
<tr>
<td>a1 != a2</td>
<td>bool</td>
<td>same as !(a1 == a2)</td>
</tr>
<tr>
<td>X()</td>
<td></td>
<td>creates a default instance. [Note: a destructor is assumed. — end note]</td>
</tr>
<tr>
<td>x a(b);</td>
<td>post: Y(a) == b</td>
<td></td>
</tr>
<tr>
<td>a.construct(p,t)</td>
<td>(not used)</td>
<td>Effect: ::new((void*)p) T(t)</td>
</tr>
<tr>
<td>a.construct(p,v)</td>
<td>(not used)</td>
<td>Effect: ::new((void*)p) T(std::forward&lt;V&gt;(v))</td>
</tr>
<tr>
<td>a.destroy(p)</td>
<td>(not used)</td>
<td>Effect: ((T*)p)-&gt;~T()</td>
</tr>
</tbody>
</table>

---

3 The member class template rebind in the table above is effectively a typedef template: if the name Allocator is bound to SomeAllocator<T>, then Allocator::rebind<U>::other is the same type as SomeAllocator<U>.

\(^{225}\) 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”.

Draft
Implementations of containers described in this International Standard are permitted to assume that their Allocator template parameter meets the following two additional requirements beyond those in Table 40.

— All instances of a given allocator type are required to be interchangeable and always compare equal to each other.
— The typedef members `pointer`, `const_pointer`, `size_type`, and `difference_type` are required to be `T*`, `T const*`, `std::size_t`, and `std::ptrdiff_t`, respectively.

Implementors are encouraged to supply libraries that can accept allocators that encapsulate more general memory models and that support non-equal instances. In such implementations, any requirements imposed on allocators by containers beyond those requirements that appear in Table 40, and the semantics of containers and algorithms when allocator instances compare non-equal, are implementation-defined.

If the alignment associated with a specific over-aligned type is not supported by an allocator, instantiation of the allocator for that type may fail. The allocator also may silently ignore the requested alignment. [Note: additionally, the member function `allocate` for that type may fail by throwing an object of type `std::bad_alloc`. — end note]

## 20.2 Utility components

This subclause contains some basic function and class templates that are used throughout the rest of the library.

**Namespace `<utility>` synopsis**

```cpp
namespace std {
    // 20.2.1, operators:
    namespace rel_ops {
        template<class T> bool operator!=(const T&, const T&);
        template<class T> bool operator> (const T&, const T&);
        template<class T> bool operator<=(const T&, const T&);
        template<class T> bool operator>=(const T&, const T&);
    }

    // 20.2.2, forward/move:
    template <class T> struct identity;
    template <class T> T&& forward(typename identity<T>::type&&);
    template <class T> typename remove_reference<T>::type&& move(T&&);

    // 20.2.3, pairs:
    template <class T1, class T2> struct pair;
    template <class T1, class T2>
        bool operator==(const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2>
        bool operator> (const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2>
        bool operator<=(const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2>
        bool operator>=(const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2>
        bool operator< (const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2>
        bool operator> (const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2>
        bool operator<=(const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2>
        bool operator>=(const pair<T1,T2>&, const pair<T1,T2>&);
}
```

Draft
template <class T1, class T2>
  void swap(pair<T1,T2>&, pair<T1,T2>&);
template <class T1, class T2>
  void swap(pair<T1,T2>&&, pair<T1,T2>&&);
template <class T1, class T2>
  void swap(pair<T1,T2>&, pair<T1,T2>&&);
template <class T1, class T2>
  pair<typename decay<T1>::type, typename decay<T2>::type> make_pair(T1&&, T2&&);

// 20.2.3, tuple-like access to pair:
template <class T> class tuple_size;
template <int I, class T> class tuple_element;
template <class T1, class T2> struct tuple_size<std::pair<T1, T2> >;
template <class T1, class T2> struct tuple_element<0, std::pair<T1, T2> >;
template <class T1, class T2> struct tuple_element<1, std::pair<T1, T2> >;

template<int I, class T1, class T2> P& get(std::pair<T1, T2>&);
template<int I, class T1, class T2> const P& get(const std::pair<T1, T2>&);
}

20.2.1 Operators [operators]

To avoid redundant definitions of operator!= out of operator== and operators >, <=, and >= out of operator<, the library provides the following:

template <class T> bool operator!=(const T& x, const T& y);
  Requires: Type T is EqualityComparable (31).
  Returns: !(x == y).

template <class T> bool operator>(const T& x, const T& y);
  Requires: Type T is LessThanComparable (32).
  Returns: y < x.

template <class T> bool operator<=(const T& x, const T& y);
  Requires: Type T is LessThanComparable (32).
  Returns: !(y < x).

template <class T> bool operator>=(const T& x, const T& y);
  Requires: Type T is LessThanComparable (32).
  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 Utility components

20.2.2 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.

```cpp
template <class T> struct identity { typedef T type; };

[Note: The use of identity in forward forces users to explicitly specify the template parameter. This is necessary to get the correct forwarding semantics. — end note]

template <class T> T&& forward(typename identity<T>::type&& t);
```

Returns: `t`.

Return type: If `T` is an lvalue-reference type, an lvalue; otherwise, an rvalue.

```cpp
[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()
{
    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 `int&&` (an rvalue). In the second call to `factory`, `A1` is deduced as `int&`, so `i` is forwarded to `A`'s constructor as an `int&` (an lvalue). In both cases, `A2` is deduced as `double`, so `1.414` is forwarded to `A`'s constructor as `double&&` (an rvalue).

— end example

```cpp
template <class T> typename remove_reference<T>::type&& move(T&& t);
```

Returns: `t`.

Return type: an rvalue.

[Example:

template <class T, class A1>
shared_ptr<T> factory(A1&& a1)
{
    return shared_ptr<T>(new T(std::forward<A1>(a1)));
}
```

Draft
struct A
{
    A();                // copies from lvalues
    A(const A&);       // moves from rvalues
    A(A&&);            // moves from rvalues
};

void g()
{
    A a;
    shared_ptr<A> sp1 = factory<A>(a);  // "a" binds to A(const A&)
    shared_ptr<A> sp1 = factory<A>(std::move(a)); // "a" binds to A(A&&)
}

In the first call to factory, A1 is deduced as A&, so a is forwarded as a non-const lvalue. This binds to the
constructor A(const A&), which copies the value from a. In the second call to factory, because of the call
std::move(a), A1 is deduced as A, so a is forwarded as an rvalue. This binds to the constructor A(A&&), which
moves the value from a.

— end example

20.2.3 Pairs

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 20.3.1.3 and 20.3.1.4).

template <class T1, class T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;
    pair();
    pair(const T1& x, const T2& y);
    template<class U, class V> pair(U&& x, V&& y);
    pair(pair&& p);
    template<class U, class V> pair(const pair<U, V>&& p);
    template<class U, class V> pair(pair<U, V>&& p);
    template<class U, class... Args> pair(U&& x, Args&&... args);

    pair& operator=(pair&& p);
    template<class U, class V> pair& operator=(pair<U, V>&& p);

    void swap(pair&& p);
};
pair();

Effects: Initializes its members as if implemented: pair() : first(), second() {}

pair(const T1& x, const T2& y);

Effects: The constructor initializes first with x and second with y.

template<class U, class V> pair(U& x, V& y);

Effects: The constructor initializes first with std::forward<U>(x) and second with std::forward<T>(y).

template<class U, class... Args> pair(U& x, Args&... args);

Effects: The constructor initializes first with std::forward<U>(x) and second with std::forward<Args>(args)...
template<class T1, class T2> void swap(pair<T1, T2>& x, pair<T1, T2>& y);

template<class T1, class T2> void swap(pair<T1, T2>&& x, pair<T1, T2>& y);

template<class T1, class T2> void swap(pair<T1, T2>& x, pair<T1, T2>&& y);

Effects: x.swap(y)


template <class T1, class T2>
pair<typename decay<T1>::type, typename decay<T2>::type> make_pair(T1&& x, T2&& y);

Returns: pair<typename decay<T1>::type, typename decay<T2>::type>(std::forward<T1>(x), std::forward<T2>(y));

[Example: In place of:

    return pair<int, double>(5, 3.1415926); // explicit types

    a C++ program may contain:

    return make_pair(5, 3.1415926); // types are deduced

— end example ]

tuple_size<pair<T1, T2> >::value

Returns: integral constant expression.

Value: 2.

tuple_element<0, pair<T1, T2> >::type

Value: the type T1.

tuple_element<1, pair<T1, T2> >::type

Value: the type T2.

template<int I, class T1, class T2>
P& get(pair<T1, T2>&);

template<int I, class T1, class T2>
const P& get(const pair<T1, T2>&);

Return type: If I == 0 then P is T1, if I == 1 then P is T2, and otherwise the program is ill-formed.

Returns: If I == 0 returns p.first, otherwise returns p.second.

20.3 Tuples [tuple]

20.3 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.

Header <tuple> synopsis

Draft
namespace std {

    // 20.3.1, class template tuple:
template <class... Types> class tuple;

    // 20.3.1.2, tuple creation functions:
const unspecified ignore;

template <class... Types>
tuple<Types...> make_tuple(Types&&...);

template <class... Types>
tuple<Types&...> tie(Types&...);

template <class... TTypes, class... UTypes>
tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>&, const tuple<UTypes...>&);

template <class... TTypes, class... UTypes>
tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&, const tuple<UTypes...>&);

template <class... TTypes, class... UTypes>
tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>&, tuple<UTypes...>&&);

template <class... TTypes, class... UTypes>
tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&& , tuple<UTypes...>&&);

    // 20.3.1.3, tuple helper classes:
template <class T> class tuple_size; // undefined

template <class... Types> class tuple_size<tuple<Types...> >;

    template <int I, class T> class tuple_element; // undefined
    template <int I, class... Types> class tuple_element<I, tuple<Types...> >;

    // 20.3.1.4, element access:
template <int I, class... Types>
typename tuple_element<I, tuple<Types...> >::type& get(tuple<Types...>&);

    template <int I, class ... types>
typename tuple_element<I, tuple<Types...> >::type const& get(const tuple<Types...>&);

    // 20.3.1.5, relational operators:
template<class... TTypes, class... UTypes>
bool operator==(const tuple<TTypes...>&, const tuple<UTypes...>&);

    template<class... TTypes, class... UTypes>
bool operator<=(const tuple<TTypes...>&, const tuple<UTypes...>&);

    template<class... TTypes, class... UTypes>
bool operator!=(const tuple<TTypes...>&, const tuple<UTypes...>&);

    template<class... TTypes, class... UTypes>
bool operator>(const tuple<TTypes...>&, const tuple<UTypes...>&);

    template<class... TTypes, class... UTypes>
bool operator>(const tuple<TTypes...>&, const tuple<UTypes...>&);

    template<class... TTypes, class... UTypes>
    Draft
bool operator<=(const tuple<TTypes...>&, const tuple<UTypes...>&);

template<class... TTypes, class... UTypes>
bool operator>=(const tuple<TTypes...>&, const tuple<UTypes...>&);

} // namespace std

20.3.1 Class template tuple

template <class... Types>
class tuple
{
  public:
    tuple();
    explicit tuple(const Types&...);
    template <class... UTypes>
      explicit tuple(UTypes&&...);
    tuple(const tuple&);
    tuple(tuple&&);
    template <class... UTypes>
      tuple(const tuple<UTypes...>&);
    template <class... UTypes>
      tuple(tuple<UTypes...>&&);
    template <class U1, class U2>
      tuple(const pair<U1, U2>&);  // iff sizeof...(Types) == 2
    template <class U1, class U2>
      tuple(pair<U1, U2>&&);   // iff sizeof...(Types) == 2
    tuple& operator=(const tuple&);
    tuple& operator=(tuple&&);
    template <class... UTypes>
      tuple& operator=(const tuple<UTypes...>&);
    template <class... UTypes>
      tuple& operator=(tuple<UTypes...>&&);
    template <class U1, class U2>
      tuple& operator=(const pair<U1, U2>&);  // iff sizeof...(Types) == 2
    template <class U1, class U2>
      tuple& operator=(pair<U1, U2>&&);   // iff sizeof...(Types) == 2

};

20.3.1.1 Construction

tuple();

1 Requires: Each type in Types shall be default constructible.

Draft
Effects: Default initializes each element.

tuple(const Types&...);

Requires: Each type in Types shall be copy constructible.

Effects: Copy initializes each element with the value of the corresponding parameter.

template <class... UTypes>
    explicit tuple(UTypes&&... u);

Requires: Each type in Types shall be move constructible from the corresponding type in UTypes. sizeof...(Types) == sizeof...(UTypes).

Effects: Initializes the elements in the tuple with the corresponding value in std::forward<UTypes>(u).

tuple(const tuple& u);

Requires: Each type in Types shall be copy constructible.

Effects: Copy constructs each element of *this with the corresponding element of u.

tuple(tuple&& u);

Requires: Each type in Types shall be move constructible.

Effects: Move-constructs each element of *this with the corresponding element of u.

template <class... UTypes> tuple(const tuple<UTypes...>& u);

Requires: Each type in Types shall be constructible from the corresponding type in UTypes. sizeof...(Types) == sizeof...(UTypes).

Effects: Constructs each element of *this with the corresponding element of u.

[Note: enable_if can be used to make the converting constructor and assignment operator exist only in the cases where the source and target have the same number of elements. — end note]

template <class... UTypes> tuple(tuple<UTypes...>&& u);

Requires: Each type in Types shall be move-constructible from the corresponding type in UTypes. sizeof...(Types) == sizeof...(UTypes).

Effects: Move-constructs each element of *this with the corresponding element of u.

[Note: enable_if can be used to make the converting constructor and assignment operator exist only in the cases where the source and target have the same number of elements. — end note]

template <class U1, class U2> tuple(const pair<U1, U2>& u);

Requires: The first type in Types shall be constructible from U1 and the second type in Types shall be constructible from U2. sizeof...(Types) == 2.

Effects: Constructs the first element with u.first and the second element with u.second.

template <class U1, class U2> tuple(pair<U1, U2>&& u);
\textit{Requires:} The first type in \texttt{Types} shall be move constructible from \texttt{U1} and the second type in \texttt{Types} shall be move-constructible from \texttt{U2}. \#sizeof\ldots\texttt{(Types)} == 2.

\textit{Effects:} Constructs the first element with \texttt{std::move(u.first)} and the second element with \texttt{std::move(u.second)}.

tuple& operator=(const tuple& u);

\textit{Requires:} Each type in \texttt{Types} shall be assignable.

\textit{Effects:} Assigns each element of \texttt{u} to the corresponding element of \texttt{*this}.

\textit{Returns:} \texttt{*this}

tuple& operator=(tuple&& u);

\textit{Requires:} Each type in \texttt{Types} shall be move-assignable.

\textit{Effects:} Move-assigns each element of \texttt{u} to the corresponding element of \texttt{*this}.

\textit{Returns:} \texttt{*this}.

\textit{Note:} There are rare conditions where the converting copy constructor is a better match than the element-wise construction, even though the user might intend differently. An example of this is if one is constructing a one-element tuple where the element type is another tuple type \texttt{T} and if the parameter passed to the constructor is not of type \texttt{T}, but rather a tuple type that is convertible to \texttt{T}. The effect of the converting copy construction is most likely the same as the effect of the element-wise construction would have been. However, if it possible to compare the “nesting depths” of the source and target tuples and decide to select the element-wise constructor if the source...
nesting depth is smaller than the target nesting-depth. This can be accomplished using an enable_if template or other tools for constrained templates. — end note]

{template <class U1, class U2> tuple& operator=(pair<U1, U2>&& u);

Requires: The first type in Types shall be assignable from U1 and the second type in Types shall be assignable from U2. sizeof...(Types) == 2.

Effects: Assigns std::move(u.first) to the first element of *this and std::move(u.second) to the second element of *this.

Returns: *this.

20.3.1.2 Tuple creation functions [tuple.creation]

template<class... Types>
tuple<VTypes...> make_tuple(Types&&... t);

where each Vi in VTypes is X& if, for the corresponding type Ti in Types, remove_cv<remove_reference<Ti>::type>::type equals reference_wrapper<X>, otherwise Vi is decay<Ti>::type.

Returns: tuple<VTypes...>(std::forward<Types>(t)...).

[Example:
  int i; float j;
  make_tuple(1, ref(i), cref(j))

  creates a tuple of type
  tuple<int, int&, const float&>

  — end example]

template<class... Types>
tuple<Types&...> tie(Types&... t);

Returns: tuple<Types&>(t...). When an argument in t is ignore, assigning any value to the corresponding tuple element has no effect.

[Example: tie functions allow one to create tuples that unpack tuples into variables. ignore can be used for elements that are not needed:

  int i; std::string s;
  tie(i, ignore, s) = make_tuple(42, 3.14, "C++");
  // i == 42, s == "C++"

  — end example]

template <class... TTypes, class... UTypes>
tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
Requires: All the types in TTypes shall be CopyConstructible. All the types in UTypes shall be CopyConstructible.

Returns: A tuple object constructed by copy constructing its first sizeof...(TTypes) elements from the corresponding elements of t and copy constructing its last sizeof...(UTypes) elements from the corresponding elements of u.

\[
\text{template } \langle \text{class... TTypes, class... UTypes}\rangle \\
tuple\langle\text{TTypes...}, \text{UTypes...}\rangle \text{ tuple}\_\text{cat}(\text{tuple}\langle\text{TTypes...}\rangle&& t, \text{const tuple}\langle\text{UTypes...}\rangle&& u);
\]

Requires: All the types in TTypes shall be MoveConstructible. All the types in UTypes shall be CopyConstructible.

Returns: A tuple object constructed by move constructing its first sizeof...(TTypes) elements from the corresponding elements of t and copy constructing its last sizeof...(UTypes) elements from the corresponding elements of u.

\[
\text{template } \langle \text{class... TTypes, class... UTypes}\rangle \\
tuple\langle\text{TTypes...}, \text{UTypes...}\rangle \text{ tuple}\_\text{cat}(\text{const tuple}\langle\text{TTypes...}\rangle& t, \text{tuple}\langle\text{UTypes...}\rangle&& u);
\]

Requires: All the types in TTypes shall be CopyConstructible. All the types in UTypes shall be MoveConstructible.

Returns: A tuple object constructed by copy constructing its first sizeof...(TTypes) elements from the corresponding elements of t and move constructing its last sizeof...(UTypes) elements from the corresponding elements of u.

\[
\text{template } \langle \text{class... TTypes, class... UTypes}\rangle \\
tuple\langle\text{TTypes...}, \text{UTypes...}\rangle \text{ tuple}\_\text{cat}(\text{tuple}\langle\text{TTypes...}\rangle&& t, \text{tuple}\langle\text{UTypes...}\rangle&& u);
\]

Requires: All the types in TTypes shall be MoveConstructible. All the types in UTypes shall be MoveConstructible.

Returns: A tuple object constructed by move constructing its first sizeof...(TTypes) elements from the corresponding elements of t and move constructing its last sizeof...(UTypes) elements from the corresponding elements of u.

### 20.3.1.3 Tuple helper classes

\[
\text{template } \langle \text{class... Types}\rangle \\
\text{class tuple}\_\text{size}\langle\text{tuple}\langle\text{Types...}\rangle\rangle > \\
: \text{public integral constant}<\text{size}_t, \text{sizeof...}(\text{Types})> \{ \}
\]

\[
\text{template } \langle \text{int I, class... Types}\rangle \\
\text{class tuple}\_\text{element}\langle\text{I, tuple}\langle\text{Types...}\rangle\rangle > \\
\{ \\
\text{public:} \\
\text{typedef TI type;} \\
\}
\]

Requires: 0 <= I and I < sizeof...(Types). The program is ill-formed if I is out of bounds.
20.3 Tuples

Type: TI is the type of the Ith element of Types, where indexing is zero-based.

20.3.1.4 Element access

template <int I, class... types>
  typename tuple_element<I, tuple<Types...> >::type& get(tuple<Types...>& t);

1 Requires: 0 <= I and I < sizeof...(Types). The program is ill-formed if I is out of bounds.
2 Returns: A reference to the Ith element of t, where indexing is zero-based.

template <int I, class... types>
  typename tuple_element<I, tuple<Types...> >::type const& get(const tuple<Types...>& t);

3 Requires: 0 <= I and I < sizeof...(Types). The program is ill-formed if I is out of bounds.
4 Returns: A const reference to the Ith element of t, where indexing is zero-based.

[ Note: Constness is shallow. If a T in Types is some reference type X&, the return type is X&, not const X&. However, if the element type is non-reference type T, the return type is const T&. This is consistent with how constness is defined to work for member variables of reference type. — end note ]

6 [ Note: The reason 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 template keyword. — end note ]

20.3.1.5 Relational operators

template<class... TTypes, class... UTypes>
  bool operator==(const tuple<TTypes...>& t, const tuple<UTypes...>& u);

1 Requires: For all i, where 0 <= i and i < sizeof...(Types), get<i>(t) == get<i>(u) is a valid expression returning a type that is convertible to bool. sizeof...(TTypes) == sizeof...(UTypes).
2 Returns: true iff get<i>(t) == get<i>(u) for all i. For any two zero-length tuples e and f, e == f returns true.
3 Effects: The elementary comparisons are performed in order from the zeroth index upwards. No comparisons or element accesses are performed after the first equality comparison that evaluates to false.

template<class... TTypes, class... UTypes>
  bool operator<(const tuple<TTypes...>& t, const tuple<UTypes...>& u);

4 Requires: For all i, where 0 <= i and i < sizeof...(Types), get<i>(t) == get<i>(u) is a valid expression returning a type that is convertible to bool. sizeof...(TTypes) == sizeof...(UTypes).
4 Returns: The result of a lexicographical comparison between t and u. The result is defined as: (bool)(get<0>(t) < get<0>(u)) || (!bool)(get<0>(u) < get<0>(t)) && t_tail < u_tail, where t_tail for some tuple r is a tuple containing all but the first element of r. For any two zero-length tuples e and f, e < f returns false.

Draft
Returns: !(t == u).

```cpp
template<class... TTypes, class... UTypes>
bool operator>(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

Returns: u < t.

```cpp
template<class... TTypes, class... UTypes>
bool operator<=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

Returns: !u < t)

```cpp
template<class... TTypes, class... UTypes>
bool operator>=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

Returns: !(t < u)

[Note: The above definitions for comparison operators do not require \( t_{\text{tail}} \) (or \( u_{\text{tail}} \)) to be constructed. It may not even be possible, as \( t \) and \( u \) are not required to be copy constructible. Also, all comparison 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 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.

#### 20.4.1 Requirements

1. 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 derived, directly or indirectly, from a specialization of the template `integral_constant` (20.4.3), with the arguments to the template `integral_constant` determined by the requirements for the particular property being described.

2. 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 derived, directly or indirectly, from an instance of the template `integral_constant` (20.4.3), with the arguments to the template `integral_constant` determined by the requirements for the particular relationship being described.

3. 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 nested type named \( \text{type} \), which shall be a synonym for the modified type.

#### 20.4.2 Header `<type_traits> synopsis`

```cpp
namespace std {
    // [20.4.3] helper class:

    Draft
```
template <class T, T v> struct integral_constant;
typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;

// [20.4.4.1] primary type categories:
template <class T> struct is_void;
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;

// [20.4.4.2] 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;

// [20.4.4.3] 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_standard_layout;
template <class T> struct is_pod;
template <class T> struct is_empty;
template <class T> struct is_polymorphic;
template <class T> struct is_abstract;
template <class T> struct has_trivial_default_constructor;
template <class T> struct has_trivial_copy_constructor;
template <class T> struct has_trivial_assign;
template <class T> struct has_trivial_destructor;
template <class T> struct has_nothrow_default_constructor;
template <class T> struct has_nothrow_copy_constructor;
template <class T> struct has_nothrow_assign;
template <class T> struct has_virtual_destructor;
template <class T> struct is_signed;
template <class T> struct is_unsigned;
template <class T> struct alignment_of;
template <class T> struct rank;
template <class T, unsigned I = 0> struct extent;
// [20.4.5] 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;

// [20.4.6.1] const-volatile modifications:
template <class T> struct remove_const;
template <class T> struct remove_volatile;
template <class T> struct remove_cv;
template <class T> struct add_const;
template <class T> struct add_volatile;
template <class T> struct add_cv;

// [20.4.6.2] reference modifications:
template <class T> struct remove_reference;
template <class T> struct add_lvalue_reference;
template <class T> struct add_rvalue_reference;

// [20.4.6.3] sign modifications:
template <class T> struct make_signed;
template <class T> struct make_unsigned;

// [20.4.6.4] array modifications:
template <class T> struct remove_extent;
template <class T> struct remove_all_extents;

// [20.4.6.5] pointer modifications:
template <class T> struct remove_pointer;
template <class T> struct add_pointer;

// [20.4.7] other transformations:
template <std::size_t Len, std::size_t Align> struct aligned_storage;
template <std::size_t Len, class... Types> struct aligned_union;
template <class T> struct decay;
template <bool, class T = void> struct enable_if;
template <bool, class T, class F> struct conditional;
} // namespace std

The behavior of a program that adds specializations for any of the class templates defined in this subclause is undefined.

20.4.3 Helper classes

template <class T, T v>
struct integral_constant
{
    static const T value = v;
    typedef T value_type;
    typedef integral_constant<T,v> type;
};
assert constexpr integral_constant

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.

### 20.4.4 Unary Type Traits

This sub-clause contains templates that may be used to query the properties of a type at compile time.

Each of these templates shall be a UnaryTypeTrait (20.4.1), publicly derived directly or indirectly from true_type if the corresponding condition is true, otherwise from false_type.

#### 20.4.4.1 Primary Type Categories

The primary type categories correspond to the descriptions given in section 3.9 of the C++ standard.

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.

[Note: For any given type T, exactly one of the primary type categories has a value member that evaluates to true. — end note]}

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>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_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 (23.2.1) 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>
20.4.4.2 Composite type traits

These templates provide convenient compositions of the primary type categories, corresponding to the descriptions given in section 3.9.

For any given type \( T \), the result of applying one of these templates to \( T \), and to \( cv\-\text{qualified} \ T \) shall yield the same result.

Table 42: 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 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>

20.4.4.3 Type properties

These templates provide access to some of the more important properties of types.

It is unspecified whether the library defines any full or partial specialisations of any of these templates. A program may specialise any of these templates on a user-defined type, provided the semantics of the specialisation match those given for the template in its description.

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 43: Type Property Predicates

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Draft</td>
</tr>
</tbody>
</table>
### General Utilities Library

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;class T&gt; struct is_const;</code></td>
<td>T is const-qualified (3.9.3)</td>
<td></td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct is_volatile;</code></td>
<td>T is volatile-qualified (3.9.3)</td>
<td></td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct is_trivial;</code></td>
<td>T is a trivial type (3.9)</td>
<td>T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) <code>void</code>.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct is_standard_layout;</code></td>
<td>T is a standard-layout type (3.9)</td>
<td>T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) <code>void</code>.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct is_pod;</code></td>
<td>T is a POD type (3.9)</td>
<td>T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) <code>void</code>.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct is_empty;</code></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>T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) <code>void</code>.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct is_polymorphic;</code></td>
<td>T is a polymorphic class (10.3)</td>
<td>T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) <code>void</code>.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct is_abstract;</code></td>
<td>T is an abstract class (10.4)</td>
<td>T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) <code>void</code>.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct has_trivial_default_constructor;</code></td>
<td>T is a <a href="#">POD trivial</a> type (3.9) or a class type with a trivial default constructor (12.1) or an array of such a class type.</td>
<td>T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) <code>void</code>.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct has_trivial_copy_constructor;</code></td>
<td>T is a <a href="#">POD trivial</a> type (3.9) or a reference type or a class type with a trivial copy constructor (12.8).</td>
<td>T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) <code>void</code>.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct has_trivial_assign;</code></td>
<td>T is neither const nor a reference type, and T is a <a href="#">POD trivial</a> type (3.9) or a class type with a trivial copy assignment operator (12.8).</td>
<td>T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) <code>void</code>.</td>
</tr>
<tr>
<td>Template</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct has_trivial Destructor;</td>
<td>T is a <em>POD trivial</em> type (3.9) or a reference type or a class type with a trivial destructor (12.4) or an array of such a class type. T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) void.</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct has_nothrow_default_constructor;</td>
<td>has_trivial_default_constructor&lt;T&gt;::value is true or T is a class type with a default constructor that is known not to throw any exceptions or T is an array of such a class type. T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) void.</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct has_nothrow_copy_constructor;</td>
<td>has_trivial_copy_constructor&lt;T&gt;::value is true or T is a class type with a copy constructor that is known not to throw any exceptions or T is an array of such a class type. T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) void.</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct has_nothrow_assign;</td>
<td>T is neither const nor a reference type, and has_trivial_assign&lt;T&gt;::value is true or T is a class type with a copy assignment operator taking an lvalue of type T that is known not to throw any exceptions or T is an array of such a class type. T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) void.</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct has_virtualDestructor;</td>
<td>T has a virtual destructor (12.4) T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) void.</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_signed;</td>
<td>is_arithmetic&lt;T&gt;::value &amp;&amp; T(-1) &lt; T(0)</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_unsigned;</td>
<td>is_arithmetic&lt;T&gt;::value &amp;&amp; T(0) &lt; T(-1)</td>
<td></td>
</tr>
</tbody>
</table>

**Example:**

```cpp
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 ]

[ Example:
remove_const<const volatile int>::type // volatile int
remove_const<const int* const>::type // const int*
remove_const<const int&>::type // const int&

— end example ]

<table>
<thead>
<tr>
<th>Template</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt; struct alignment_of;</td>
<td>alignof(T).</td>
</tr>
<tr>
<td>struct rank;</td>
<td>If T names an array type, an integer value representing the number of dimensions of T; otherwise, 0.</td>
</tr>
<tr>
<td>template &lt;class T, template &lt;unsigned I = 0&gt; struct extent;</td>
<td>If T is not an array type (8.3.4), or if it has rank less than I, or if I is 0 and T has type “array of unknown bound of U”, then 0; otherwise, the size of the I’th dimension of T</td>
</tr>
</tbody>
</table>

[ Example:
// the following assertions hold:
assert(rank<int>::value == 0);
assert(rank<int[2]>::value == 1);
assert(rank<int[][4]>::value == 2);

— end example ]

[ 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[][4]>::value == 0);
assert(extent<int[2], 1>::value == 0);
assert(extent<int[2][4], 1>::value == 4);
assert(extent<int[][4], 1>::value == 4);

— end example ]

20.4.5 Relationships between types [meta.rel]

This sub-clause contains templates that may be used to query relationships between types at compile time.
Each of these templates shall be a `BinaryTypeTrait` (20.4.1), publicly derived directly or indirectly from `true_type` if the corresponding condition is true, otherwise from `false_type`.

### Table 45: Type Relationship Predicates

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;class T, class U&gt; struct is_same;</code></td>
<td>T and U name the same type with the same cv-qualifications</td>
<td>If Base and Derived are 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><code>template &lt;class Base, class Derived&gt; struct is_base_of;</code></td>
<td>Base is a base class of Derived (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></td>
</tr>
<tr>
<td><code>template &lt;class From, class To&gt; struct is_convertible;</code></td>
<td>From and To shall be complete types, arrays of unknown bound, or (possibly cv-qualified) void types. The following code is well formed: template &lt;class T&gt; typename add_rvalue_reference&lt;T&gt;::type create(); To test() { return create&lt;From&gt;(); } [Note: This requirement gives well defined results for reference types, void types, array types, and function types. — end note]</td>
<td></td>
</tr>
</tbody>
</table>

3 [Example:

```cpp
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 // true
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]

20.4.6 Transformations between types [meta.trans]

This sub-clause contains templates that may be used to transform one type to another following some predefined rule.
20.4 Metaprogramming and type traits

General utilities library

Each of the templates in this subclause shall be a TransformationTrait (20.4.1).

### 20.4.6.1 Const-volatile modifications

Table 46: Const-volatile modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt; struct remove_const;</td>
<td>The member typedef type shall name the same type as T except that any top-level const-qualifier has been removed. [Example: remove_const&lt;const volatile int&gt;::type evaluates to volatile int, whereas remove_const&lt;const int*&gt; is const int*. — end example]</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct remove_volatile;</td>
<td>The member typedef type shall name the same type as T except that any top-level volatile-qualifier has been removed. [Example: remove_volatile&lt;const volatile int&gt;::type evaluates to const int, whereas remove_volatile&lt;volatile int*&gt; is volatile int*. — end example]</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct remove_cv;</td>
<td>The member typedef type shall be the same as T except that any top-level cv-qualifier has been removed. [Example: remove_cv&lt;const volatile int&gt;::type evaluates to int, whereas remove_cv&lt;const volatile int*&gt; is const volatile int*. — end example]</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct add_const;</td>
<td>If T is a reference, function, or top-level const-qualified type, then type shall name the same type as T, otherwise T const.</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct add_volatile;</td>
<td>If T is a reference, function, or top-level volatile-qualified type, then type shall name the same type as T, otherwise T volatile.</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct add_cv;</td>
<td>The member typedef type shall name the same type as add_const&lt;typename add_volatile&lt;T&gt;::type&gt;::type.</td>
</tr>
</tbody>
</table>

### 20.4.6.2 Reference modifications

Table 47: Reference modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt; struct remove_reference;</td>
<td>If T has type “reference to T1” then the member typedef type shall name T1; otherwise, 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 type shall name T&amp;; otherwise, if T names a type “rvalue reference to T1” then the member typedef type shall name T&amp;&amp;; otherwise, type shall name T. [Note: This rule reflects the semantics of reference collapsing. For example, when a type T names a type T1&amp;, the type add_rvalue_reference&lt;T&gt;::type is not an rvalue reference. — end note]</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 type shall name T&amp;&amp;; otherwise, type shall name T.</td>
</tr>
</tbody>
</table>

### 20.4.6.3 Sign modifications

Draft
Table 48: Sign modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
</table>
| template <class T> struct make_signed; | If T names a (possibly cv-qualified) signed integral type (3.9.1) then the member typedef type shall name the type T; otherwise, if T names a (possibly cv-qualified) unsigned integral type then type shall name the corresponding signed integral type, with the same cv-qualifiers as T; otherwise, type shall name the signed integral type with smallest rank (4.13) for which sizeof(T) == sizeof(type), with the same cv-qualifiers as T. 

Requires: T shall be a (possibly cv-qualified) integral type or enumeration but not a bool type. |

| template <class T> struct make_unsigned; | If T names a (possibly cv-qualified) unsigned integral type (3.9.1) then the member typedef type shall name the type T; otherwise, if T names a (possibly cv-qualified) signed integral type then type shall name the corresponding unsigned integral type, with the same cv-qualifiers as T; otherwise, type shall name the unsigned integral type with smallest rank (4.13) for which sizeof(T) == sizeof(type), with the same cv-qualifiers as T. 

Requires: T shall be a (possibly cv-qualified) integral type or enumeration but not a bool type. |

20.4.6.4 Array modifications [meta.trans.arr]

Table 49: Array modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
</table>
| template <class T> struct remove_extent; | If T names a type “array of U”, the member typedef type shall be U, otherwise T. 

[Note: For multidimensional arrays, only the first array dimension is removed. For a type “array of const U”, the resulting type is const U. — end note] |

| template <class T> struct remove_all_extents; | If T is “multi-dimensional array of U”, the resulting member typedef type is U, otherwise T. |

1 [Example]

// the following assertions hold:
assert((is_same<remove_extent<int>::type, int>::value));
assert((is_same<remove_extent<int[2]>::type, int>::value));
assert((is_same<remove_extent<int[2][3]>::type, int[3]>::value));
assert((is_same<remove_extent<int[][3]>::type, int[3]>::value));

— end example ]

2 [Example]

// the following assertions hold:
assert((is_same<remove_all_extents<int>::type, int>::value));
assert((is_same<remove_all_extents<int[2]>::type, int>::value));
20.4 Metaprogramming and type traits

assert((is_same<remove_all_extents<int[2][3]>::type, int>::value));
assert((is_same<remove_all_extents<int[][3]>::type, int>::value));

— end example

20.4.6.5 Pointer modifications

Table 50: Pointer modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt; struct remove_pointer;</td>
<td>If T has type “(possibly cv-qualified) pointer to T1” then the member typedef type shall name T1; otherwise, it shall name T.</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct add_pointer;</td>
<td>The member typedef type shall name the same type as remove_reference&lt;T&gt; :: type*.</td>
</tr>
</tbody>
</table>

20.4.7 Other transformations

Table 51: Other transformations

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;std::size_t Len, std::size_t Align = default-alignment&gt; struct aligned_storage;</td>
<td>Len shall not be zero. Align shall be equal to alignment_of&lt;T&gt; :: value for some type T or to default-alignment.</td>
<td>The value of default-alignment shall be the most stringent alignment requirement for any C++ object type whose size is no greater than Len (3.9). The member typedef type shall be a POD type suitable for use as uninitialized storage for any object whose size is at most Len and whose alignment is a divisor of Align.</td>
</tr>
<tr>
<td>template &lt;std::size_t Len, class... Types&gt; struct aligned_union;</td>
<td>At least one type shall be provided.</td>
<td>The member typedef type shall be a POD type suitable for use as uninitialized storage for any object whose type is listed in Types; its size shall be at least Len. The static member alignment_value shall be an integral constant of type std::size_t whose value is the strictest alignment of all types listed in Types.</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct decay;</td>
<td>Let U be remove_reference&lt;T&gt; :: type. If is_array&lt;U&gt; :: value is true, the member typedef type shall equal remove_extent&lt;U&gt; :: type*. If is_function&lt;U&gt; :: value is true, the member typedef type shall equal add_pointer&lt;U&gt; :: type. Otherwise the member typedef type equals U.</td>
<td></td>
</tr>
</tbody>
</table>
template <bool B, class T = void> struct enable_if;

If T is true, the member typedef type shall equal T; otherwise, there shall be no member typedef type.

template <bool B, class T, class F> struct conditional;

If B is true, the member typedef type shall equal T. If B is false, the member typedef type shall equal F.

---

1. [Note: a typical implementation of aligned_storage would define type as:

   union type
   {
      unsigned char __data[Len];
      Aligner __align;
   };

   where Aligner is the smallest POD type for which alignment_of<Aligner>::value is a multiple of align — end note]

   [Note: a typical implementation would define aligned_storage as:

   template <std::size_t Len, std::size_t Alignment>
   struct aligned_storage {
      typedef struct {
         alignas(Alignment) unsigned char __data[Len];
      } type;
   }; — end note]

2. It is implementation-defined whether any extended alignment is supported (3.11).

20.5 Function objects [function.objects]

Function objects are objects with an operator() defined. 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 an object with an operator() defined. This not only makes algorithmic templates work with pointers to functions, but also enables them to work with arbitrary function objects.

2. Header <functional> synopsis

   namespace std {
   // 20.5.3, base:
   template <class Arg, class Result> struct unary_function;
   template <class Arg1, class Arg2, class Result> struct binary_function;

   // 20.5.4 result_of:
   template <class T> class result_of; // undefined
   template <class F, class... Args> class result_of<F(ArgTypes...)>
   ;

   // 20.5.5, reference_wrapper:
   template <class T> class reference_wrapper;

   Draft
template <class T> reference_wrapper<T> ref(T&);
template <class T> reference_wrapper<const T> cref(const T&);

template <class T> reference_wrapper<T> ref(reference_wrapper<T>);
template <class T> reference_wrapper<const T> cref(reference_wrapper<T>);

// 20.5.6, arithmetic operations:
template <class T> struct plus;
template <class T> struct minus;
template <class T> struct multiplies;
template <class T> struct divides;
template <class T> struct modulus;
template <class T> struct negate;

// 20.5.7, comparisons:
template <class T> struct equal_to;
template <class T> struct not_equal_to;
template <class T> struct greater;
template <class T> struct less;
template <class T> struct greater_equal;
template <class T> struct less_equal;

// 20.5.8, logical operations:
template <class T> struct logical_and;
template <class T> struct logical_or;
template <class T> struct logical_not;

// 20.5.9, negators:
template <class Predicate> class unary_negate;
template <class Predicate> class binary_negate;

// 20.5.10, bind:
template<class T> struct is_bind_expression;
template<class T> struct is_placeholder;

template<class Fn, class... Types>
  unspecified bind(Fn, Types...);
template<class R, class Fn, class... Types>
  unspecified bind(Fn, Types...);

namespace placeholders {
  // M is the implementation-defined number of placeholders
  extern unspecified _1;
  extern unspecified _2;
  ...

Draft
extern unspecified _M;
}

// D.8, binders (deprecated):
template <class Fn> class binder1st;
template <class Fn, class T>
  binder1st<Fn> bind1st(const Fn&, const T&);
template <class Fn> class binder2nd;
template <class Fn, class T>
  binder2nd<Fn> bind2nd(const Fn&, const T&);

// 20.5.11, adaptors:
template <class Arg, class Result> class pointer_to_unary_function;
template <class Arg, class Result>
  pointer_to_unary_function<Arg,Result> ptr_fun(Result (*)(Arg));
template <class Arg1, class Arg2, class Result>
  class pointer_to_binary_function;
template <class Arg1, class Arg2, class Result>
  pointer_to_binary_function<Arg1,Arg2,Result>
    ptr_fun(Result (*)(Arg1,Arg2));

// 20.5.12, adaptors:
template<class S, class T> class mem_fun_t;
template<class S, class T, class A> class mem_fun1_t;
template<class S, class T>
  mem_fun_t<S,T> mem_fun(S (T::*f)());
template<class S, class T, class A>
  mem_fun1_t<S,T,A> mem_fun(S (T::*f)(A));
template<class S, class T> class mem_fun_ref_t;
template<class S, class T, class A> class mem_fun1_ref_t;
template<class S, class T>
  mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)());
template<class S, class T, class A>
  mem_fun1_ref_t<S,T,A> mem_fun_ref(S (T::*f)(A));

template <class S, class T> class const_mem_fun_t;
template <class S, class T, class A> class const_mem_fun1_t;
template <class S, class T>
  const_mem_fun_t<S,T> mem_fun(S (T::*f)() const);
template <class S, class T, class A>
  const_mem_fun1_t<S,T,A> mem_fun(S (T::*f)(A) const);
template <class S, class T> class const_mem_fun_ref_t;
template <class S, class T, class A> class const_mem_fun1_ref_t;
template <class S, class T>
  const_mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)() const);
template <class S, class T, class A>
  const_mem_fun1_ref_t<S,T,A> mem_fun_ref(S (T::*f)(A) const);

// 20.5.13, member function adaptors:

Draft
20.5 Function objects

Function objects

General utilities library

\begin{verbatim}
// 20.5.14 polymorphic function wrappers:
class bad_function_call;

// undefined

template<class R, class... ArgTypes> class function;

template<class R, class... ArgTypes>
void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&);

// 20.5.15, hash function base 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<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<long long>;
template <> struct hash<unsigned long>;
template <> struct hash<unsigned long long>;
template <> struct hash<float>;
template <> struct hash<double>;
template <> struct hash<long double>;

template<class T> struct hash<T*>;
\end{verbatim}

Draft
template <> struct hash<std::string>
{);
template <> struct hash<std::u16string>
{);
template <> struct hash<std::u32string>
{);
template <> struct hash<std::wstring>
{)

3 [Example: If a C++ program wants to have a by-element addition of two vectors \texttt{a} and \texttt{b} containing \texttt{double} and put the result into \texttt{a}, it can do:
\begin{verbatim}
transform(a.begin(), a.end(), b.begin(), a.begin(), plus<double>());
\end{verbatim}
— end example ]

4 [Example: To negate every element of \texttt{a}:
\begin{verbatim}
transform(a.begin(), a.end(), a.begin(), negate<double>());
\end{verbatim}
— end example ]

5 To enable adaptors and other components to manipulate function objects that take one or two arguments it is required that the function objects correspondingly provide typedefs \texttt{argument\_type} and \texttt{result\_type} for function objects that take one argument and \texttt{first\_argument\_type}, \texttt{second\_argument\_type}, and \texttt{result\_type} for function objects that take two arguments.

20.5.1 Definitions \[func.def\]

1 The following definitions apply to this clause:
2 A \textit{call signature} is the name of a return type followed by a parenthesized comma-separated list of zero or more argument types.
3 A \textit{call wrapper} is an object of a call wrapper type.
4 A \textit{call wrapper type} is a type that holds a callable object and supports a call operation that forwards to that object.
5 A \textit{callable object} is an object of a callable type.
6 A \textit{callable type} is a pointer to function, a pointer to member function, a pointer to member data, or a class type whose objects can appear immediately to the left of a function call operator.
7 A \textit{target object} is the callable object held by a call wrapper object.

20.5.2 Requirements \[func.require\]

1 Define \texttt{INVOKE}(f, t_1, t_2, \ldots, t_N) as follows:
\begin{itemize}
\item \((t_1.*f)(t_2, \ldots, t_N)\) when \(f\) is a pointer to a member function of a class \(T\) and \(t_1\) 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 \(T\);
\item \(((t_1).*f)(t_2, \ldots, t_N)\) when \(f\) is a pointer to a member function of a class \(T\) and \(t_1\) is not one of the types described in the previous item;
\item \(t_1.*f\) when \(f\) is a pointer to member data of a class \(T\) and \(t_1\) 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 \(T\);
\end{itemize}
— \((*t1)\cdot f\) when \(f\) is a pointer to member data of a class \(T\) and \(t1\) is not one of the types described in the previous item;
— \(f(t1, t2, \ldots, tN)\) in all other cases.

2 Define \(\text{INVOKE}(f, t1, t2, \ldots, tN, R)\) as \(\text{INVOKE}(f, t1, t2, \ldots, tN)\) implicitly converted to \(R\).

3 If a call wrapper ([20.5.1]) 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 ([20.5.1]):
— if \(T\) is a function, reference to function, or pointer to function type, result_type shall be a synonym for the return type of \(T\);
— if \(T\) is a pointer to member function, result_type shall be a synonym for the return type of \(T\);
— if \(T\) is a class type with a member type result_type, then result_type shall be a synonym for \(T::\text{result}_-\text{type}\);
— otherwise result_type shall not be defined.

4 Every call wrapper [20.5.1] shall be CopyConstructible. A simple call wrapper is a call wrapper that is CopyAssignable and whose copy constructor and assignment operator do not throw exceptions. A forwarding call wrapper is a call wrapper that can be called with an argument list. [Note: in a typical implementation forwarding call wrappers have an overloaded function call operator of the form

```c++
template<class... ArgTypes>
R operator()(ArgTypes&&... args) cv-qual;
```

— end note ]

20.5.3 Base [base]

1 The following classes are provided to simplify the typedefs of the argument and result types:

```c++
template <class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
};

template <class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```

20.5.4 Function object return types [func.ret]

```c++
namespace std {
    template <class> class result_of; // undefined
}
```

Draft
Given an rvalue fn of type Fn and values t1, t2, ..., tN of types T1, T2, ..., TN in ArgTypes, respectively, the type member is the result type of the expression fn(t1, t2, ..., tN). The values ti are lvalues when the corresponding type Ti is a reference type, and rvalues otherwise.

The implementation may determine the type member via any means that produces the exact type of the expression fn(t1, t2, ..., tN) for the given types. [Note: The intent is that implementations are permitted to use special compiler hooks — end note]

If Fn is not a function object defined by the standard library, and if either the implementation cannot determine the type of the expression fn(t1, t2, ..., tN) or the expression is ill-formed, the implementation shall use the following process to determine the type member:

1. If Fn is a function pointer or function reference type, type shall be the return type of the function type.
2. If Fn is a member function pointer type, type shall be the return type of the member function type.
3. If Fn is a member data pointer type RT::*_type shall be cv R& when T1 is cv U1&, R otherwise.
4. If Fn is a possibly cv-qualified class type with a member type result_type, type shall be typename Fn::result_type.
5. If Fn is a possibly cv-qualified class type with no member named result_type or if typename Fn::result_type is not a type:
   (a) If N=0 (no arguments), type shall be void.
   (b) If N>0, type shall be typename Fn::template result<Fn(ArgTypes...)>::type.
6. Otherwise, the program is ill-formed.

20.5.5 Class template reference_wrapper

```cpp
template <class T> class reference_wrapper
    : public unary_function<T, R> // see below
    : public binary_function<T1, T2, R> // see below
{
    public :
        // types
        typedef T type;
        typedef seebelow result_type; // not always defined

    // construct/copy/destroy
    explicit reference_wrapper(T&);
    explicit reference_wrapper(T&&) = delete; // do not bind to temporary objects
    reference_wrapper(const reference_wrapper<T>& x);
}
```

Draft
20.5 Function objects

```
// assignment
reference_wrapper& operator=(const reference_wrapper<T>& x);

// access
operator T& () const;
T& get() const;

// invocation

```

1. `reference_wrapper<T>` is a CopyConstructible and Assignable wrapper around a reference to an object of type `T`.
2. `reference_wrapper` has a weak result type ([20.5.2]).
3. The template instantiation `reference_wrapper<T>` shall be derived from `std::unary_function<T1, R>` only if the type `T` is any of the following:
   - a function type or a pointer to function type taking one argument of type `T1` and returning `R`
   - 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 that is derived from `std::unary_function<T1, R>`
4. The template instantiation `reference_wrapper<T>` shall be derived from `std::binary_function<T1, T2, R>` 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` and returning `R`
   - 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 that is derived from `std::binary_function<T1, T2, R>`

20.5.5.1 `reference_wrapper construct/copy/destroy` [refwrap.const]

```
explicit reference_wrapper(T& t);

Effects: Constructs a `reference_wrapper` object that stores a reference to `t`.

Throws: nothing.

reference_wrapper(const reference_wrapper<T>& x);

Effects: Constructs a `reference_wrapper` object that stores a reference to `x.get()`.

Throws: nothing.
```
20.5.5.2 reference_wrapper assignment

```
reference_wrapper& operator=(const reference_wrapper<T>& x);
```

1 Postconditions: *this stores a reference to x.get().
2 Throws: Nothing.

20.5.5.3 reference_wrapper access

```
operator T& () const;
```

1 Returns: The stored reference.
2 Throws: nothing.

```
T& get() const;
```

1 Returns: The stored reference.
2 Throws: nothing.

20.5.5.4 reference_wrapper invocation

```
template <class... ArgTypes>
    typename result_of<T(ArgTypes... )>::type
    operator()(ArgTypes&&... args) const;
```

1 Returns: INVOKE(get(), std::forward<ArgTypes>(args)...).(20.5.2)
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.

20.5.5.5 reference_wrapper helper functions

```
template <class T> reference_wrapper<T> ref(T& t);
```

1 Returns: reference_wrapper<T>(t)
2 Throws: nothing.

```
template <class T> reference_wrapper<T> ref(reference_wrapper<T>t);
```

1 Returns: ref(t.get())
2 Throws: nothing.

```
template <class T> reference_wrapper<const T> cref(const T& t);
```

1 Returns: reference_wrapper <const T>(t)
2 Throws: nothing.
template <class T> reference_wrapper<const T> cref(reference_wrapper<T> t);

Returns: cref(t.get());

Throws: nothing.

20.5.6 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> struct plus : binary_function<T,T,T> {
    T operator()(const T& x, const T& y) const;
};

operator() returns \( x + y \).

template <class T> struct minus : binary_function<T,T,T> {
    T operator()(const T& x, const T& y) const;
};

operator() returns \( x - y \).

template <class T> struct multiplies : binary_function<T,T,T> {
    T operator()(const T& x, const T& y) const;
};

operator() returns \( x \times y \).

template <class T> struct divides : binary_function<T,T,T> {
    T operator()(const T& x, const T& y) const;
};

operator() returns \( x / y \).

template <class T> struct modulus : binary_function<T,T,T> {
    T operator()(const T& x, const T& y) const;
};

operator() returns \( x \% y \).

template <class T> struct negate : unary_function<T,T> {
    T operator()(const T& x) const;
};

operator() returns \(-x\).

20.5.7 Comparisons

The library provides basic function object classes for all of the comparison operators in the language (5.9, 5.10).

template <class T> struct equal_to : binary_function<T,T,bool> {
    bool operator()(const T& x, const T& y) const;
}
operator() returns \( x == y \).

```cpp
template <class T> struct not_equal_to : binary_function<T,T,bool> {
    bool operator()(const T& x, const T& y) const;
};
```

operator() returns \( x != y \).

```cpp
template <class T> struct greater : binary_function<T,T,bool> {
    bool operator()(const T& x, const T& y) const;
};
```

operator() returns \( x > y \).

```cpp
template <class T> struct less : binary_function<T,T,bool> {
    bool operator()(const T& x, const T& y) const;
};
```

operator() returns \( x < y \).

```cpp
template <class T> struct greater_equal : binary_function<T,T,bool> {
    bool operator()(const T& x, const T& y) const;
};
```

operator() returns \( x \geq y \).

```cpp
template <class T> struct less_equal : binary_function<T,T,bool> {
    bool operator()(const T& x, const T& y) const;
};
```

operator() returns \( x \leq y \).

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.

### 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> struct logical_and : binary_function<T,T,bool> {
    bool operator()(const T& x, const T& y) const;
};
```

operator() returns \( x \&\& y \).

```cpp
template <class T> struct logical_or : binary_function<T,T,bool> {
    bool operator()(const T& x, const T& y) const;
};
```

operator() returns \( x || y \).
template <typename T> struct logical_not : unary_function<T, bool> {
  bool operator()(const T& x) const;
};

operator() returns \!x.

### 20.5.9 Negators

Negators `not1` and `not2` take a unary and a binary predicate, respectively, and return their complements (5.3.1).

```
template <typename Predicate>
  class unary_negate
    : public unary_function<typename Predicate::argument_type, bool> {
    public:
      explicit unary_negate(const Predicate& pred);
      bool operator()(const typename Predicate::argument_type& x) const;
    };
	nooperator() returns \!pred(x).
```

```
template <typename Predicate>
  unary_negate<Predicate> not1(const Predicate& pred);

Returns: unary_negate<Predicate>(pred).
```

```
template <typename Predicate>
  class binary_negate
    : public binary_function<typename Predicate::first_argument_type,
                              typename Predicate::second_argument_type, bool> {
    public:
      explicit binary_negate(const Predicate& pred);
      bool operator()(const typename Predicate::first_argument_type& x,
                      const typename Predicate::second_argument_type& y) const;
    };

binary_negate<Predicate>() returns \!pred(x,y).
```

```
template <typename Predicate>
  binary_negate<Predicate> not2(const Predicate& pred);

Returns: binary_negate<Predicate>(pred).
```

### 20.5.10 Template function `bind`

The template function `bind` returns an object that binds a function object passed as an argument to additional arguments.

```
template <typename Predicate>
  binary_negate<Predicate> not2(const Predicate& pred);

Returns: binary_negate<Predicate>(pred).
```

```
20.5.10 Template function `bind`

The template function `bind` returns an object that binds a function object passed as an argument to additional arguments.

Binders `bind1st` and `bind2nd` take a function object \(\text{fn}\) of two arguments and a value \(x\) and return a function object of
one argument constructed out of fn with the first or second argument correspondingly bound to x.

20.5.10.1 Function object binders

This subclause describes a uniform mechanism for binding arguments of function objects.

20.5.10.1.1 Class template is_bind_expression

namespace std {
    template<class T> struct is_bind_expression {
        static const bool value = see below;
    };
} // namespace std

is_bind_expression can be used to detect function objects generated by bind. bind uses is_bind_expression to detect subexpressions. Users may specialize this template to indicate that a type should be treated as a subexpression in a bind call.

static const bool value;

1 true if T is a type returned from bind, false otherwise.

20.5.10.1.2 Class template is_placeholder

namespace std {
    template<class T> struct is_placeholder {
        static const int value = see below;
    };
} // namespace std

is_placeholder can be used to detect the standard placeholders _1, _2, and so on. bind uses is_placeholder to detect placeholders. Users may specialize this template to indicate a placeholder type.

static const int value;

1 value is J if T is the type of std::placeholders::_J, 0 otherwise.

20.5.10.1.3 Function template bind

template<class F, class... BoundArgs>
unspecified bind(F f, BoundArgs... bound_args);

1 Requires: F and each Ti in BoundArgs shall be CopyConstructible. INVOKE (f, w1, w2, ..., wN) (20.5.2) shall be a valid expression for some values w1, w2, ..., wN, where N == sizeof...(bound_args).

2 Returns: A forwarding call wrapper g with a weak result type (20.5.2). The effect of g(u1, u2, ..., uM) shall be INVOKE(f, v1, v2, ..., vN, result_of<F cv (V1, V2, ..., VN>::type), where cv represents the cv-qualifiers of g and the values and types of the bound arguments v1, v2, ..., vN are determined as specified below.

Draft
template<class R, class F, class... BoundArgs>
    unspecified bind(F f, BoundArgs... bound_args);

    Requires: F and each Ti in BoundArgs shall be CopyConstructible. INVoke(f, w1, w2, ..., wN) shall be a
    valid expression for some values w1, w2, ..., wN, where N == sizeof...(bound_args).

Returns: A forwarding call wrapper g with a nested type result_type defined as a synonym for R. The effect of
    g(u1, u2, ..., uM) shall be INVoke(f, v1, v2, ..., vN, R), where the values and types of the bound
    arguments v1, v2, ..., vN are determined as specified below.

The values of the bound arguments v1, v2, ..., vN and their corresponding types V1, V2, ..., VN depend on the
type of the corresponding argument ti in bound_args of type Ti in BoundArgs in the call to bind and the cv-qualifiers
cv of the call wrapper g as follows:

- if ti is of type reference_wrapper<T> the argument is ti.get() and its type Vi is T&;
- if the value of std::is_bind_expression<Ti>::value is true the argument is ti(u1, u2, ..., uM) and
  its type Vi is result_of<Ti cv (U1&, U2&, ..., UM&)>::type;
- if the value j of std::is_placeholder<Ti>::value is not zero the argument is std::forward<Uj>(uj) and
  its type Vi is Uj&&;
- otherwise the value is ti and its type Vi is Ti cv &.

20.5.10.1.4 Placeholders

namespace std {
    namespace placeholders {
        // M is the implementation-defined number of placeholders
        extern unspecified _1;
        extern unspecified _2;
        ...
        extern unspecified _M;
    }
} // namespace std

All placeholder types shall be DefaultConstructible and CopyConstructible, and their default constructors and
copy constructors shall not throw exceptions. It is implementation defined whether placeholder types are Assignable.
Assignable placeholders’ copy assignment operators shall not throw exceptions.

20.5.11 Adaptors for pointers to functions

To allow pointers to (unary and binary) functions to work with function adaptors the library provides:

    template <class Arg, class Result>
    class pointer_to_unary_function : public unary_function<Arg, Result> {
    public:
        explicit pointer_to_unary_function(Result (*f)(Arg));
        Result operator()(Arg x) const;
    };
operator() returns \( f(x) \).

```cpp
template <class Arg, class Result>
pointer_to_unary_function<Arg, Result> ptr_fun(Result (*)(Arg));
```

**Returns:** `pointer_to_unary_function<Arg, Result>(f)`.

```cpp
template <class Arg1, class Arg2, class Result>
class pointer_to_binary_function :
  public binary_function<Arg1, Arg2, Result> {
public:
  explicit pointer_to_binary_function(Result (*)(Arg1, Arg2));
  Result operator()(Arg1 x, Arg2 y) const;
};
```

operator() returns \( f(x,y) \).

```cpp
template <class Arg1, class Arg2, class Result>
pointer_to_binary_function<Arg1, Arg2, Result>
ptr_fun(Result (*)(Arg1, Arg2));
```

**Returns:** `pointer_to_binary_function<Arg1, Arg2, Result>(f)`.

**Example:**

```cpp
int compare(const char*, const char*);
replace_if(v.begin(), v.end(),
  not1(bind2nd(ptr_fun(compare), "abc")), "def");
```

replaces each `abc` with `def` in sequence `v`. — end example

### 20.5.12 Adaptors for pointers to members

The purpose of the following is to provide the same facilities for pointer to members as those provided for pointers to functions in 20.5.11.

```cpp
template <class S, class T> class mem_fun_t :
  public unary_function<T*, S> {
public:
  explicit mem_fun_t(S (T::*p)());
  S operator()(T* p) const;
};
```

`mem_fun_t` calls the member function it is initialized with given a pointer argument.

```cpp
template <class S, class T, class A> class mem_fun1_t :
  public binary_function<T*, A, S> {
public:
  explicit mem_fun1_t(S (T::*p)(A));
  S operator()(T* p, A x) const;
};
```

Draft
mem_fun1_t calls the member function it is initialized with given a pointer argument and an additional argument of the appropriate type.

```cpp
template<class S, class T> mem_fun_t<S, T>
mem_fun(S (T::*f)());
template<class S, class T, class A> mem_fun1_t<S, T, A>
mem_fun(S (T::*f)(A));
```

mem_fun(&X::f) returns an object through which X::f can be called given a pointer to an X followed by the argument required for f (if any).

```cpp
template <class S, class T> class mem_fun_ref_t
: public unary_function<T, S> {
public:
    explicit mem_fun_ref_t(S (T::*p)());
    S operator()(T& p) const;
};
```

mem_fun_ref_t calls the member function it is initialized with given a reference argument.

```cpp
template <class S, class T, class A> class mem_fun1_ref_t
: public binary_function<T, A, S> {
public:
    explicit mem_fun1_ref_t(S (T::*p)(A));
    S operator()(T& p, A x) const;
};
```

mem_fun1_ref_t calls the member function it is initialized with given a reference argument and an additional argument of the appropriate type.

```cpp
template<class S, class T> mem_fun_ref_t<S, T>
mem_fun_ref(S (T::*f)());
template<class S, class T, class A> mem_fun1_ref_t<S, T, A>
mem_fun_ref(S (T::*f)(A));
```

mem_fun_ref(&X::f) returns an object through which X::f can be called given a reference to an X followed by the argument required for f (if any).

```cpp
template <class S, class T> class const_mem_fun_t
: public unary_function<const T*, S> {
public:
    explicit const_mem_fun_t(S (T::*p)() const);
    S operator()(const T* p) const;
};
```

const_mem_fun_t calls the member function it is initialized with given a pointer argument.

```cpp
template <class S, class T, class A> class const_mem_fun1_t
: public binary_function<const T*, A, S> {
public:
    explicit const_mem_fun1_t(S (T::*p)(A) const);
    S operator()(const T* p, A x) const;
};
```
};
9 \textit{const\_mem\_fun1\_t} calls the member function it is initialized with given a pointer argument and an additional argument of the appropriate type.

\textit{template} <\textit{class S, class T}> \textit{const\_mem\_fun\_t}\textsubscript{S,T} \textit{mem\_fun}(S \textup{(}T::*f\textup{)}() \textup{const});
\textit{template} <\textit{class S, class T, class A}> \textit{const\_mem\_fun1\_t}\textsubscript{S,T,A} \textit{mem\_fun}(S \textup{(}T::*f\textup{)}(A) \textup{const});

10 \textit{mem\_fun}(\&X::f) returns an object through which X::f can be called given a pointer to an X followed by the argument required for f (if any).

\textit{template} <\textit{class S, class T}> class \textit{const\_mem\_fun\_ref\_t}
\textit{:\ }\textit{public\ unary\_function}\langle T, S \rangle \{ 
\textit{\textbf{public}:}
\textit{explicit\ const\_mem\_fun\_ref\_t}(S \textup{(}T::*p\textup{)}() \textup{const});
S\ \textit{operator()}(\textit{const T}\&\ p)\ \textup{const};
\};

11 \textit{const\_mem\_fun\_ref\_t} calls the member function it is initialized with given a reference argument.

\textit{template} <\textit{class S, class T, class A}> class \textit{const\_mem\_fun1\_ref\_t}
\textit{:\ }\textit{public\ binary\_function}\langle T, A, S \rangle \{ 
\textit{\textbf{public}}:
\textit{explicit\ const\_mem\_fun1\_ref\_t}(S \textup{(}T::*p\textup{)}(A) \textup{const});
S\ \textit{operator()}(\textit{const T}\&\ p, A x)\ \textup{const};
\};

12 \textit{const\_mem\_fun1\_ref\_t} calls the member function it is initialized with given a reference argument and an additional argument of the appropriate type.

\textit{template} <\textit{class S, class T}> \textit{const\_mem\_fun\_ref\_t}\textsubscript{S,T} \textit{mem\_fun\_ref}(S \textup{(}T::*f\textup{)}() \textup{const});
\textit{template} <\textit{class S, class T, class A}> \textit{const\_mem\_fun1\_ref\_t}\textsubscript{S,T,A} \textit{mem\_fun\_ref}(S \textup{(}T::*f\textup{)}(A) \textup{const});

13 \textit{mem\_fun\_ref}(\&X::f) returns an object through which X::f can be called given a reference to an X followed by the argument required for f (if any).

\textbf{20.5.13 Function template mem\_fn}

\textit{template} <\textit{class R, class T}> \textit{unspecified\ mem\_fn(R T::* pm)};

1 \textit{Returns:} A simple call wrapper ([20.5.1]) \textit{fn} such that the expression \textit{fn}(t, a2, \ldots, aN) is equivalent to \textit{INVOKE}(pm, t, a2, \ldots, aN) ([20.5.2]). \textit{fn} shall have a nested type \textit{result\_type} that is a synonym for the return type of \textit{pm} when \textit{pm} is a pointer to member function.

2 The simple call wrapper shall be derived from \textit{std::unary\_function}\langle cv\ T*, Ret \rangle when \textit{pm} is a pointer to member function with cv-qualifier \textit{cv} and taking no arguments, where \textit{Ret} is \textit{pm}'s return type.

\textit{Draft}
The simple call wrapper shall be derived from
\texttt{std::binary\_function<cv T*, T1, Ret>},
when \texttt{pm} is a pointer to member function with \texttt{cv}-qualifier \texttt{cv}
and taking one argument of type \texttt{T1}, where \texttt{Ret} is \texttt{pm}'s return type.

\textit{Throws}: Nothing.

\textit{Remarks}: Implementations may implement \texttt{mem\_fn} as a set of overloaded function templates.

\section{Polymorphic function wrappers} \label{func.wrap}

This subclause describes a polymorphic wrapper class that encapsulates arbitrary function objects.

\subsection{Class \texttt{bad\_function\_call}} \label{func.wrap.badcall}

An exception of type \texttt{bad\_function\_call} is thrown by \texttt{function::operator()} ([\ref{func.wrap.func.2}]) when the function wrapper object has no target.

\begin{verbatim}
namespace std {
  class bad_function_call : public std::exception {
   public:
     // 20.5.14.1.1, constructor:
     bad_function_call();
  }; // namespace std

  20.5.14.1.1 bad_function_call constructor \label{func.wrap.badcall.const}

  bad_function_call();
  \hspace*{1em} \textit{Effects}: constructs a \texttt{bad\_function\_call} object.

  20.5.14.2 Class template \texttt{function} \label{func.wrap.func}

  namespace std {
   template<class> class function; // undefined
   template<class R, class... ArgTypes>
   class function<R(ArgTypes...)> :
      public unary_function<T1, R> // iff sizeof...(ArgTypes) == 1 and ArgTypes contains T1
      public binary_function<T1, T2, R> // iff sizeof...(ArgTypes) == 2 and ArgTypes contains T1 and T2
   {
      public:
         typedef R result_type;

         // 20.5.14.2.1, construct/copy/destroy:
         explicit function();
         function(unspecified-null-pointer-type);
         function(const function&);
         function(function&&);
   }

  Draft
\end{verbatim}
template<class F> function(F);
-template<class F, class A> function(F, const A&);
template<class F> function(F&&);

function& operator=(const function&);
function& operator=(function&&);
function& operator=(unspecified-null-pointer-type);
template<class F> function& operator=(F);
template<class F> function& operator=(F&&);
template<class F> function& operator=(reference_wrapper<F>);

~function();

// 20.5.14.2.2, function modifiers:
void swap(function&);
-template<class F, class A> void assign(F, const A&);

// 20.5.14.2.3, function capacity:
operator unspecified-bool-type () const;

// deleted overloaded close possible hole in the type system
-template<class R2, class... ArgTypes2>
    bool operator==(const function<R2(ArgTypes2...)>&) = delete;
template<class R2, class... ArgTypes2>
    bool operator!=(const function<R2(ArgTypes2...)>&) = delete;

// 20.5.14.2.4, function invocation:
R operator()(ArgTypes...) const;

// 20.5.14.2.5, function target access:
const std::type_info& target_type() const;
template <typename T> T* target();
template <typename T> const T* target() const;

private:
// 20.5.14.2.6, undefined operators:
template<class R2, class... ArgTypes2> bool operator==(const function<R2(ArgTypes2...)>&, unspecified-null-pointer-type);
template<class R2, class... ArgTypes2> bool operator!=(const function<R2(ArgTypes2...)>&, unspecified-null-pointer-type);

// 20.5.14.2.7, Null pointer comparisons:
template <class R, class... ArgTypes>
    bool operator==(const function<R(ArgTypes...)>&, unspecified-null-pointer-type);
template <class R, class... ArgTypes>
    bool operator==(unspecified-null-pointer-type, const function<R(ArgTypes...)>&);
template <class R, class... ArgTypes>
    bool operator!=(const function<R(ArgTypes...)>&, unspecified-null-pointer-type);
The function class template provides polymorphic wrappers that generalize the notion of a function pointer. Wrappers can store, copy, and call arbitrary callable objects ([20.5.1]), given a call signature ([20.5.1]), allowing functions to be first-class objects.

A function object \( f \) of type \( F \) is Callable for argument types \( T_1, T_2, \ldots, T_N \) in \( \text{ArgTypes} \) and a return type \( R \), if, given lvalues \( t_1, t_2, \ldots, t_N \) of types \( T_1, T_2, \ldots, T_N \), respectively, \( \text{INVOKE}(f, t_1, t_2, \ldots, t_N) \) is well-formed (20.5.2) and, if \( R \) is not void, convertible to \( R \).

The function class template is a call wrapper (20.5.1) whose call signature ([20.5.1]) is \( R(\text{ArgTypes}...) \).

### 20.5.14.2.1 function construct/copy/destroy

**explicit function();**

*Postconditions:* \(!*this.*

*Throws:* nothing.

**function(unspecified-null-pointer-type);**

*Postconditions:* \(!*this.*

*Throws:* nothing.

**function(const function& f);**

*Postconditions:* \(!*this if !f; otherwise, *this targets a copy of f.target().

*Offers:* shall not throw exceptions if \( f \)'s target is a function pointer or a function object passed via reference wrapper. Otherwise, may throw \( \text{bad_alloc} \) or any exception thrown by the copy constructor of the stored function object. [Note: Implementations are encouraged to avoid the use of dynamically allocated memory for small function objects, e.g., where \( f \)'s target is an object holding only a pointer or reference to an object and a member function pointer. —end note]

**function(function&& f);**

*Effects:* If \( f \), \*this has no target; otherwise, move-constructs the target of \( f \) into the target of \*this, leaving \( f \) in a valid state with an unspecified value.

**template<class F> function(F f);**

**template<class F, class A> function(F f, const A& a);**

**template<class F> function(F&& f);**

*Requires:* \( f \) shall be callable for argument types \( \text{ArgTypes} \) and return type \( R \). A shall be an allocator (20.1.2). The copy constructor and destructor of \( A \) shall not throw exceptions.
Postconditions: !*this if any of the following hold:

- \( f \) is a NULL function pointer.
- \( f \) is a NULL member function pointer.
- \( F \) is an instance of the function class template, and !f

Otherwise, *this targets a copy of \( f \) or std::move(\( f \)) if \( f \) is not a pointer to member function, and targets a copy of mem_fn(\( f \)) if \( f \) is a pointer to member function. The second constructor shall use a copy of if dynamic storage must be allocated for the copy of \( f \). [Note: implementations are encouraged to avoid the use of dynamically allocated memory for small function 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.

```cpp
function& operator=(const function& f);
```

Effects: function(\( f \)).swap(*this);

Returns: *this

```cpp
function& operator=(function&& f);
```

Effects: Replaces the target of *this with the target of \( f \), leaving \( f \) in a valid but unspecified state.

Returns: *this

```cpp
function& operator=(unspecified-null-pointer-type);
```

Effects: If *this != NULL, destroys the target of this.

Postconditions: !(*this).

Returns: *this

```cpp
template<class F> function& operator=(F f);
```

Effects: function(\( f \)).swap(*this);

Returns: *this

```cpp
template<class F> function& operator=(F&& f);
```

Effects: Replaces the target of *this with \( f \), leaving \( f \) in a valid but unspecified state. [Note: A valid implementation is function(\( f \)).swap(*this).

Returns: *this.

```cpp
template<class F> function& operator=(reference_wrapper<F> f);
```

Effects: function(\( f \)).swap(*this);

Returns: *this

Throws: nothing.
"function();

Effects: If *this != NULL, destroys the target of this.

20.5.14.2.2 function modifiers

void swap(function& other);

Effects: interchanges the targets of *this and other.

Throws: nothing.

template<class F, class A> void assign(F f, const A& a);

Effects: function(f, a).swap(*this)

20.5.14.2.3 function capacity

operator unspecified-bool-type() const

Returns: if *this has a target, returns a value that will evaluate true in a boolean context; otherwise, returns a value that will evaluate false in a boolean context. The value type returned shall not be convertible to int.

Throws: nothing.

[ Note: This conversion can be used in contexts where a bool is expected (e.g., an if condition); however, implicit conversions (e.g., to int) that can occur with bool are not allowed, eliminating some sources of user error. One possible implementation choice for this type is pointer-to-member. — end note ]

20.5.14.2.4 function invocation

R operator()(ArgTypes... args) const

Effects: INVOKE (f, t1, t2, ..., tN, R) (20.5.2), where f is the target object (20.5.1) of *this and t1, t2, ..., tN are the values in args....

Returns: Nothing if R is void, otherwise the return value of INVOKE (f, t1, t2, ..., tN, R).

Throws: bad_function_call if !*this; otherwise, any exception thrown by the wrapped function object.

20.5.14.2.5 function target access

const std::type_info& target_type() const;

Returns: If *this has a target of type T, typeid(T); otherwise, typeid(void).

Throws: nothing.

template<typename T> T* target();
template<typename T> const T* target() const;

Draft
3  Requires: T shall be a function object type that is Callable (20.5.14.2) for parameter types ArgTypes and return type R.

4  Returns: If target_type() == typeid(T) a pointer to the stored function target; otherwise a null pointer.

5  Throws: nothing.

20.5.14.2.6 undefined operators  [func.wrap.func.undef]

    template<class R2, class... ArgTypes2> bool operator==(const function<R2(ArgTypes2...)>&, function<R2(ArgTypes2...)>);
    template<class R2, class... ArgTypes2> bool operator!=(const function<R2(ArgTypes2...)>&, function<R2(ArgTypes2...)>);

1  These member functions shall be left undefined.

2  [Note: the boolean-like conversion opens a loophole whereby two function instances can be compared via == or !=. These undefined void operators close the loophole and ensure a compile-time error. — end note]

20.5.14.2.7 null pointer comparison operators  [func.wrap.func.nullptr]

    template <class R, class... ArgTypes>
    bool operator==(const function<R(ArgTypes...)>& f, unspecified-null-pointer-type);

    template <class R, class... ArgTypes>
    bool operator==(unspecified-null-pointer-type, const function<R(ArgTypes...)>& f);

1  Returns: !f.

2  Throws: nothing.

    template <class R, class... ArgTypes>
    bool operator!=(const function<R(ArgTypes...)>& f, unspecified-null-pointer-type);

    template <class R, class... ArgTypes>
    bool operator!=(unspecified-null-pointer-type, const function<R(ArgTypes...)>& f);

3  Returns: (bool) f.

4  Throws: nothing.

20.5.14.2.8 specialized algorithms  [func.wrap.func.alg]

    template<class R, class... ArgTypes>
    void swap(function<R(ArgTypes...)>& f1, function<R(ArgTypes...)>& f2);

1  Effects: f1.swap(f2);
20.5.15 Class template hash

The unordered associative containers defined in clause 23.4 use specializations of hash as the default hash function. This class template is only required to be instantiable for integer types (3.9.1), floating point types (3.9.1), pointer types (8.3.1), and `std::string`, `std::u16string`, `std::u32string`, and `std::wstring`.

```cpp
template <class T>
struct hash : public std::unary_function<T, std::size_t>
{
    std::size_t operator()(T val) const;
};
```

The return value of `operator()` is unspecified, except that equal arguments shall yield the same result. `operator()` shall not throw exceptions.

20.6 Memory

Header `<memory>` synopsis

```cpp
namespace std {

// 20.6.1, the default allocator:
template <class T> class allocator;
template <> class allocator<void>;
template <class T, class U>
bool operator==(const allocator<T>&, const allocator<U>&) throw();
template <class T, class U>
bool operator!=(const allocator<T>&, const allocator<U>&) throw();

// 20.6.2, raw storage iterator:
template <class OutputIterator, class T> class raw_storage_iterator;

template <class T>
pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n);
template <class T>
void return_temporary_buffer(T* p);

// 20.6.3, temporary buffers:
template <class T>
    pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n);
template <class T>
    void return_temporary_buffer(T* p);

// 20.6.4, specialized algorithms:
template <class InputIterator, class ForwardIterator>
ForwardIterator
    uninitialized_copy(InputIterator first, InputIterator last, 
    ForwardIterator result);
template <class ForwardIterator, class T>
    void uninitialized_fill(ForwardIterator first, ForwardIterator last, 
    const T& x);
template <class ForwardIterator, class Size, class T>
    void uninitialized_fill_n(ForwardIterator first, Size n, const T& x);

// 20.6.5 Class unique_ptr:
template <class X> class unique_ptr;
template <class X> class default_delete;

Draft
// 20.6.6.1, Class bad_weak_ptr:
class bad_weak_ptr;

// 20.6.6.2, Class template shared_ptr:
template<class T> class shared_ptr;

// 20.6.6.7, shared_ptr comparisons:
template<class T, class U>
  bool operator==(shared_ptr<T> const& a, shared_ptr<U> const& b);
template<class T, class U>
  bool operator!=(shared_ptr<T> const& a, shared_ptr<U> const& b);
template<class T, class U>
  bool operator<(shared_ptr<T> const& a, shared_ptr<U> const& b);

// 20.6.6.9, shared_ptr specialized algorithms:
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b);

// 20.6.6.10, shared_ptr casts:
template<class T, class U>
  shared_ptr<T> static_pointer_cast(shared_ptr<U> const& r);
template<class T, class U>
  shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const& r);
template<class T, class U>
  shared_ptr<T> const_pointer_cast(shared_ptr<U> const& r);

// 20.6.6.8, 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);

// 20.6.6.11, shared_ptr get_deleter:
template<class D, class T> D* get_deleter(shared_ptr<T> const& p);

// 20.6.6.3, Class template weak_ptr:
template<class T> class weak_ptr;

// 20.6.6.6, weak_ptr comparison:
template<class T, class U>
  bool operator<(weak_ptr<T> const& a, weak_ptr<U> const& b);

// 20.6.6.7, weak_ptr specialized algorithms:
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b);

// 20.6.7, Class enable_shared_from_this:
template<class T> class enable_shared_from_this;

// 20.6.7, Pointer alignment function
void *align(std::size_t alignment, std::size_t size,
            void **)&ptr, std::size_t & space);
20.6 Memory

### 20.6.1 The default allocator

```cpp
namespace std {

    template <class T> class allocator;

    // specialize for void:
    template <> class allocator<void> {
        public:
            typedef void* pointer;
            typedef const void* const_pointer;
            // reference-to-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; };
            allocator() throw();
            allocator(const allocator&) throw();
            template <class U> allocator(const allocator<U>&) throw();
            ~allocator() throw();

            pointer address(reference x) const;
            const_pointer address(const_reference x) const;

            pointer allocate(
                size_type, allocator<void>::const_pointer hint = 0);
            void deallocate(pointer p, size_type n);
            size_type max_size() const throw();

            void construct(pointer p, const T& val);
            template <class U> void construct(pointer p, U& val);
            template<class... Args> void construct(pointer p, Args&&... args);
            void destroy(pointer p);
    };

    20.6.1.1 allocator members

    pointer address(reference x) const;

    Returns: &x.

Draft
const_pointer address(const_reference x) const;

Returns: &x.

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) (18.5.1), 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*) (18.5.1), but it is unspecified when this function is called.

size_type max_size() const throw();

Returns: the largest value N for which the call allocate(N,0) might succeed.

void construct(pointer p, const_reference val);

Effects: ::new((void *)p) T(val)

template <class U> void construct(pointer p, U&& val);

Effects: ::new((void *)p) T(std::forward<U>(val))

template <class... Args> void construct(pointer p, Args&&... args);

Effects: ::new((void *)p) T(std::forward<Args>(args)...)

void destroy(pointer p);

Effects: p->~T()

20.6.1.2 allocator globals

[allocator.globals]

template <class T1, class T2>
bool operator==(const allocator<T1>&, const allocator<T2>&) throw();

Returns: true.
template <class T1, class T2>
bool operator!=(const allocator<T1>&, const allocator<T2>&) throw();

20.6.2 Raw storage iterator
[storage.iterator]

raw_storage_iterator is provided to enable algorithms to store their results into uninitialized memory. The formal 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 (24.1.2).

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<OutputIterator,T>& operator*();
      raw_storage_iterator<OutputIterator,T>& operator=(const T& element);
      raw_storage_iterator<OutputIterator,T>& operator++();
      raw_storage_iterator<OutputIterator,T> operator++(int);
    };

  raw_storage_iterator<OutputIterator x>;

2  Effects: Initializes the iterator to point to the same value to which x points.

  raw_storage_iterator<OutputIterator,T>& operator*();
3  Returns: *this

  raw_storage_iterator<OutputIterator,T>& operator=(const T& element);
4  Effects: Constructs a value from element at the location to which the iterator points.
5  Returns: A reference to the iterator.

  raw_storage_iterator<OutputIterator,T>& operator++();
6  Effects: Pre-increment: advances the iterator and returns a reference to the updated iterator.

  raw_storage_iterator<OutputIterator,T> operator++(int);  
7  Effects: Post-increment: advances the iterator and returns the old value of the iterator.

20.6.3 Temporary buffers
[temporary.buffer]

template <class T>
pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n);
519 General utilities library 20.6 Memory

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).

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 \( n \leq 0 \).

```cpp
template <class T> void return_temporary_buffer(T* p);
```

Effects: Deallocates the buffer to which \( p \) points.

Requires: The buffer shall have been previously allocated by get_temporary_buffer.

20.6.4 Specialized algorithms

All the iterators that are used as formal 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 formal template parameter InputIterator is required to satisfy the requirements of an input iterator (24.1.1). In all of the following algorithms, the formal template parameter ForwardIterator is required to satisfy the requirements of a forward iterator (24.1.3) and also to satisfy the requirements of a mutable iterator (24.1), and is required to have the property that no exceptions are thrown from increment, assignment, comparison, or dereference of valid iterators. In the following algorithms, if an exception is thrown there are no effects.

20.6.4.1 uninitialized_copy

```cpp
template <class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_copy(InputIterator first, InputIterator last, ForwardIterator result);
```

Effects:

```cpp
for (; first != last; ++result, ++first)
    new (static_cast<void*>(&*result)) typename iterator_traits<ForwardIterator>::value_type(*first);
```

Returns: \( \text{result} \)

20.6.4.2 uninitialized_fill

```cpp
template <class ForwardIterator, class T>
void uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x);
```

Effects:

```cpp
for (; first != last; ++first)
    new (static_cast<void*>(&*first)) typename iterator_traits<ForwardIterator>::value_type(x);
```
20.6.4.3 uninitialized_fill_n

```cpp
template <class ForwardIterator, class Size, class T>
    void uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
```

**Effects:**

```cpp
    for (; n--; ++first)
        new (static_cast<void*>(&*first))
            typename iterator_traits<ForwardIterator>::value_type(x);
```

20.6.5 Class template unique_ptr

1 Template unique_ptr stores a pointer to an object and deletes that object using the associated deleter when it is itself destroyed (such as when leaving block scope (6.7)).

2 The unique_ptr provides a semantics of strict ownership. A unique_ptr owns the object it holds a pointer to. A unique_ptr is not CopyConstructible, nor CopyAssignable, however it is MoveConstructible and MoveAssignable. [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, size_t N> struct default_delete<T[N]>

    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 D, size_t N> class unique_ptr<T[N], D>;

    template<class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y);
    template<class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y);
    template<class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& 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);
    template<class T1, class D1, class T2, class D2>
        bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

} // namespace std
```

Draft
20.6.5.1 Default deleters

20.6.5.1.1 default_delete

```cpp
template <class T> struct default_delete {
    default_delete();
    template <class U> default_delete(const default_delete<U>&);
    void operator()(T*) const;
};
```

- **default_delete();**
  - Effects: Default constructs a `default_delete`.

- **template <class U> default_delete(const default_delete<U>& other);**
  - Effects: Constructs a `default_delete` from a `default_delete<U>`.

- **void operator()(T* ptr) const;**
  - Effects: Calls `delete` on `ptr`. A diagnostic is required if `T` is an incomplete type.

20.6.5.1.2 `default_delete<T[]>`

```cpp
template <class T> struct default_delete<T[]> {
    void operator()(T* ptr) const;
}
```

- **void operator()(T* ptr) const;**
  - Effects: Calls `delete[]` on `ptr`. A diagnostic is required if `T` is an incomplete type.

20.6.5.1.3 `default_delete<T[N]>`

```cpp
template <class T, size_t N> struct default_delete<T[N]> {
    void operator()(T*, size_t) const;
};
```

- **void operator()(T* ptr, size_t) const;**
  - Effects: Calls `delete[]` on `ptr`. A diagnostic is required if `T` is an incomplete type. The `size_t` parameter is ignored.

20.6.5.2 `unique_ptr for single objects`

```cpp
template <class T, class D = default_delete<T>> class unique_ptr {
    public:
        typedef T element_type;
        typedef D deleter_type;
    }
```

Draft
20.6 Memory

General utilities library 522

// constructors
unique_ptr();
explicit unique_ptr(T* p);
unique_ptr(T* p, implementation defined (see description below) d);
unique_ptr(T* p, implementation defined (see description below) d);
unique_ptr(uniquePtr&& u);
template <class U, class E> unique_ptr(unique_ptr<U, E>&& u);

// destructor
~unique_ptr();

// assignment
unique_ptr& operator=(unique_ptr&& u);
template <class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u);
unique_ptr& operator=(unspecified-pointer-type);

// observers
T& operator*() const;
T* operator->() const;
T* get() const;
deleter_type& get_deleter();
const deleter_type& get_deleter() const;
operator unspecified-bool-type() const;

// modifiers
T* release();
void reset(T* p = 0);
void swap(unique_ptr&& u);

private:
// disable copy from lvalue
unique_ptr(const unique_ptr&) = delete;
template <class U, class E> unique_ptr(const unique_ptr<U, E>&) = delete;
unique_ptr& operator=(const unique_ptr&) = delete;
template <class U, class E> unique_ptr& operator=(const unique_ptr<U, E>&) = delete;
};

1 The default type for the template parameter D is default_delete. A client-supplied template argument D must be a
function pointer or functor for which, given a value d of type D and a pointer ptr of type T*, the expression d(ptr) is
valid and has the effect of deallocating the pointer as appropriate for that deleter. D may also be an lvalue-reference to a
deleter.

2 If the deleter D maintains state, it is intended that this state stay with the associated pointer as ownership is transferred
from unique_ptr to unique_ptr. The deleter state need never be copied, only moved or swapped as pointer ownership
is moved around. That is, the deleter need only be MoveConstructible, MoveAssignable, and Swappable, and need
not be CopyConstructible (unless copied into the unique_ptr) nor CopyAssignable.

20.6.5.2.1 unique_ptr constructors

unique_ptr();
requirements: D must be default constructible, and that construction must not throw an exception. D must not be a reference type.

Effects: Constructs a unique_ptr which owns nothing.

postconditions: get() == 0. get_deleter() returns a reference to a default constructed deleter D.

Throws: nothing.

unique_ptr(T* p);

requirements: The expression D()(p) must be well formed. The default constructor of D must not throw an exception. D must not be a reference type.

Effects: Constructs a unique_ptr which owns p.

postconditions: get() == p. get_deleter() returns a reference to a default constructed deleter D.

Throws: nothing.

unique_ptr(T* p, implementation defined d);
unique_ptr(T* p, implementation defined d);

The signature of these constructors depends upon whether D is a reference type or not. If D is non-reference type A, then the signatures are:

unique_ptr(T* p, const A& d);
unique_ptr(T* p, A&& d);

If D is an lvalue-reference type A&, then the signatures are:

unique_ptr(T* p, A& d);
unique_ptr(T* p, A&& d);

If D is an lvalue-reference type const A&, then the signatures are:

unique_ptr(T* p, const A& d);
unique_ptr(T* p, const A&& d);

requirements: The expression d(p) must be well formed.

If D is not an lvalue-reference type then

— If d is an lvalue or const rvalue then the first constructor of this pair will be selected. D must be CopyConstructible, and this unique_ptr will hold a copy of d. The copy constructor of D must not throw an exception.

— Otherwise d is a non-const rvalue and the second constructor of this pair will be selected. D need only be MoveConstructible, and this unique_ptr will hold a value move constructed from d. The move constructor of D must not throw an exception.

Otherwise D is an lvalue-reference type. d must be reference-compatible with one of the constructors. If d is an rvalue, it will bind to the second constructor of this pair. That constructor must subsequently emit a diagnostic. [Note: The diagnostic could be implemented using a static assert which assures that D is not a reference type.]
20.6 Memory

General utilities library

— 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 CopyConstructible nor MoveConstructible. This unique_ptr will hold a D which refers to the lvalue d. [ Note: D may not be an rvalue-reference type. — end note ]

Postconditions: get() == p. get_deleter() returns a reference to the internally stored deleter. If D is a reference type then get_deleter() returns a reference to the lvalue d.

Throws: nothing.

[ Example:

D d;
unique_ptr<int, D> p1(new int, D()); // D must be MoveConstructible
unique_ptr<int, D> p2(new int, d); // D must be CopyConstructible
unique_ptr<int, D&> p3(new int, d); // p3 holds a reference to d
unique_ptr<int, const D&> p4(new int, D()); // error: rvalue deleter object combined // with reference deleter type

— end example ]

unique_ptr(const unique_ptr& u);

Declared private and left undefined to inhibit copy constructing from lvalues and const rvalues.

[ Example:

template <class U, class E> unique_ptr(const unique_ptr<U, E>&& u);

Declared private and left undefined to inhibit copy constructing from lvalues and const rvalues.

[ Example:

template <class U, class E> unique_ptr(const unique_ptr<U, E>&& u);

Declared private and left undefined to inhibit copy constructing from lvalues and const rvalues.

Requires: If D is not a reference type, construction of the deleter D from an rvalue D must not throw an exception.

Effects: Constructs a unique_ptr which owns the pointer which u owns (if any). If the deleter is not a reference type, it is move constructed from u’s deleter, otherwise the reference is copy constructed from u’s deleter. After the construction, u no longer owns a pointer. [ Note: The deleter constructor can be implemented with std::forward<D>. — end note ]

Postconditions: get() == value u.get() had before the construction. get_deleter() returns a reference to the internally stored deleter which was constructed from u.get_deleter(). If D is a reference type then get_deleter() and u.get_deleter() both reference the same lvalue deleter.

Throws: nothing.

Draft
postconditions $\text{get()} == \text{value}$ had before the construction, modulo any required offset adjustments resulting from the cast from $U^*$ to $T^*$. $\text{get\_deleter()}$ returns a reference to the internally stored deleter which was constructed from $\text{u\_get\_deleter()}$.

Throws: nothing.

20.6.5.2.2 unique\_ptr destructor

$\sim\text{unique\_ptr}();$

Effects: If $\text{get()} == 0$ there are no effects. Otherwise $\text{get\_deleter()}(\text{get()})$.

Throws: nothing.

20.6.5.2.3 unique\_ptr assignment

$\text{unique\_ptr\& operator=}(\text{const unique\_ptr\& } u);$

Requires: Assignment of the deleter $D$ from an rvalue $D$ must not throw an exception.

Effects: $\text{reset(u\_release()}()$ followed by a move assignment from $u$'s deleter to this deleter.

Postconditions: This unique\_ptr now owns the pointer which $u$ owned, and $u$ no longer owns it. [Note: If $D$ is a reference type, then the referenced lvalue deleter are move assigned. — end note]

Returns: *this.

Throws: nothing.

$\text{template <class U, class E> unique\_ptr\& operator=/(\text{const unique\_ptr\&<U, E\&\& } u);}$

Requires: Assignment of the deleter $D$ from an rvalue $D$ must not throw an exception. $U^*$ must be implicitly convertible to $T^*$.

Effects: $\text{reset(u\_release()}()$ followed by a move assignment from $u$'s deleter to this deleter. If either $D$ or $E$ is a reference type, then the referenced lvalue deleter participates in the move assignment.

Postconditions: This unique\_ptr now owns the pointer which $u$ owned, and $u$ no longer owns it.

Returns: *this.

Throws: nothing.

$\text{unique\_ptr\& operator=/(unspecified-pointer-type);}$
Assigns from the literal 0 or NULL. [Note: The unspecified-pointer-type is often implemented as a pointer to a private data member, avoiding many of the implicit conversion pitfalls. — end note]

Effects: reset().
Postcondition: get() == 0
Returns: *this.
Throws: nothing.

20.6.5.2.4 unique_ptr observers
[unique.ptr.single.observers]

T& operator*() const;
Requires: get() != 0.
Returns: *get().
.Throws: nothing.

T* operator->() const;
Requires: get() != 0.
Returns: get().
Throws: nothing.

T* get() const;
Returns: The stored pointer.
Throws: nothing.

deleter_type& get_deleter();
const deleter_type& get_deleter() const;
Returns: A reference to the stored deleter.
Throws: nothing.

operator unspecified-bool-type() const;
Returns: An unspecified value that, when used in boolean contexts, is equivalent to get() != 0.
Throws: nothing.
[Note: The unspecified-bool-type is often implemented as a pointer to a private data member, avoiding many of the implicit conversion pitfalls. — end note]

20.6.5.2.5 unique_ptr modifiers
[unique.ptr.single.modifiers]

T* release();
Postcondition: get() == 0.

Returns: The value get() had at the start of the call to release.

Throws: nothing.

void reset(T* p = 0);

Effects: If p == get() there are no effects. Otherwise get_deleter()(get()).

Postconditions: get() == p.

Throws: nothing.

void swap(unique_ptr&& u);

Requires: The deleter D is Swappable and will not throw an exception under swap.

Effects: The stored pointers of this and u are exchanged. The stored deleters are swap'd (unqualified).

Throws: nothing.

20.6.5.3 unique_ptr for array objects with a runtime length

```cpp
template <class T, class D> class unique_ptr<T[], D> {
  public:
    typedef T element_type;
    typedef D deleter_type;

    // constructors
    unique_ptr();
    explicit unique_ptr(T* p);
    unique_ptr(T* p, implementation defined d);
    unique_ptr(T* p, implementation defined d);
    unique_ptr(unique_ptr&& u);

    // destructor
    ~unique_ptr();

    // assignment
    unique_ptr& operator=(unique_ptr&& u);
    unique_ptr& operator=(unspeicified-pointer-type);

    // observers
    T& operator[]() const;
    T* get() const;
    deleter_type& get_deleter();
    const deleter_type& get_deleter() const;
    operator unspecified-bool-type() const;

    // modifiers
    T* release();
}
```

Draft
20.6 Memory

void reset(T* p = 0);
void swap(unique_ptr& u);

private:
// disable copy from lvalue
unique_ptr(const unique_ptr&) = delete;
unique_ptr& operator=(const unique_ptr&) = delete;
};

A specialization for array types is provided with a slightly altered interface.

— Conversions among different types of unique_ptr<T[], D> or to or from the non-array forms of unique_ptr
are disallowed (diagnostic required).
— Pointers to types derived from T are rejected by the constructors, and by reset.
— The observers operator* and operator-> are not provided.
— The indexing observer operator[] is provided.
— The default deleter will call delete[].

Descriptions are provided below only for member functions that have behavior different from the primary template.

20.6.5.3.1 unique_ptr constructors

unique_ptr(T* p);
unique_ptr(T* p, implementation defined d);
unique_ptr(T* p, implementation defined d);

These constructors behave the same as in the primary template except that they do not accept pointer types which
are convertible to T*. [Note: One implementation technique is to create private templated overloads of these
members. — end note]

20.6.5.3.2 unique_ptr observers

T& operator[](size_t i) const;

Requires: i < the size of the array to which the stored pointer points.
Returns: get()[i].
Throws: nothing.

20.6.5.3.3 unique_ptr modifiers

void reset(T* p = 0);

Requires: Does not accept pointer types which are convertible to T* (diagnostic required). [Note: One imple-
mentation technique is to create a private templated overload. — end note]
Effects: If \( p == \) get() there are no effects. Otherwise get_deleter()(get()).

Postcondition: get() == p.

Throws: nothing.

20.6.5.4 unique_ptr for array objects with a compile time length

This specialization gives the array a length known at compile time. There are three differences between this specialization, and the specialization for arrays of length not known until run time.
20.6 Memory General utilities library

— The indexing observer for \( i \geq N \) is undefined for `unique_ptr<T[]>` and implementation defined for `unique_ptr<T[N]>`.

— The deleter is called with `get_deleter()(get(), N)` instead of `get_deleter()(get())`. Client-defined deleters may be able to make use of this extra information.

— The size of the array is available as a `static const size_t` named `size`.

Descriptions are provided below only for member functions that have behavior different from the `unique_ptr<T[]>` specialization.

### 20.6.5.4.1 unique_ptr destructor

```cpp
~unique_ptr();
```

**Effects:** If `get() == 0` there are no effects. Otherwise `get_deleter()(get(), N)`.

**Throws:** nothing.

### 20.6.5.4.2 unique_ptr observers

```cpp
T& operator[](size_t i) const;
```

**Returns:** `get()[i]`.

**Throws:** nothing.

### 20.6.5.4.3 unique_ptr modifiers

```cpp
void reset(T* p = 0);
```

**Requires:** Does not accept pointer types which are convertible to `T*` (diagnostic required). [Note: One implementation technique is to create a private templated overload. — end note]

**Effects:** If `p == get()` there are no effects. Otherwise `get_deleter()(get(), N)`.

**Postcondition:** `get() == p`.

**Throws:** nothing.

### 20.6.5.5 unique_ptr specialized algorithms

```cpp
template <class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y);
template <class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>&& y);
template <class T, class D> void swap(unique_ptr<T, D>&& x, unique_ptr<T, D>& y);
```

**Effects:** Calls `x.swap(y)`. Draft
template <class T1, class D1, class T2, class D2>
bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

Returns: x.get() == y.get().

template <class T1, class D1, class T2, class D2>
bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

Returns: x.get() != y.get().

template <class T1, class D1, class T2, class D2>
bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

Returns: x.get() < y.get().

template <class T1, class D1, class T2, class D2>
bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

Returns: x.get() <= y.get().

template <class T1, class D1, class T2, class D2>
bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

Returns: x.get() > y.get().

template <class T1, class D1, class T2, class D2>
bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

Returns: x.get() >= y.get().

20.6.6 Smart pointers [util.smartptr]

20.6.6.1 Class bad_weak_ptr [util.smartptr.weakptr]

namespace std {
    class bad_weak_ptr: public std::exception {
        public:
            bad_weak_ptr();
    };
} // namespace std

An exception of type bad_weak_ptr is thrown by the shared_ptr constructor taking a weak_ptr.

bad_weak_ptr();

Postconditions: what() returns "bad_weak_ptr".

Throws: nothing.
20.6.6.2 Class template `shared_ptr` [util.smartptr.shared]

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.

```cpp
namespace std {
    template<class T> class shared_ptr {
        public:
            typedef T element_type;

            // 20.6.6.2.1, constructors:
            shared_ptr();
            template<class Y> explicit shared_ptr(Y* p);
            template<class Y, class D> shared_ptr(Y* p, D d);
            template<class Y, class D, class A> shared_ptr(Y* p, D d, A a);
            template<class Y> shared_ptr(shared_ptr<Y> const& r);
            template<class Y> shared_ptr(shared_ptr<Y>& r);
            template<class Y> explicit shared_ptr(weak_ptr<Y> const& r);
            template<class Y> explicit shared_ptr(auto_ptr<Y>& r);

            // 20.6.6.2.2, destructor:
            ~shared_ptr();

            // 20.6.6.2.3, assignment:
            shared_ptr& operator=(shared_ptr const& r);
            template<class Y> shared_ptr& operator=(shared_ptr<Y> const& r);
            shared_ptr& operator=(shared_ptr&& r);
            template<class Y> shared_ptr& operator=(shared_ptr<Y>&& r);
            template<class Y> shared_ptr& operator=(auto_ptr<Y>& r);

            // 20.6.6.2.4, modifiers:
            void swap(shared_ptr& r);
            void reset();
            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);

            // 20.6.6.2.5, observers:
            T* get() const;
            T& operator*() const;
            T* operator->() const;
            long use_count() const;
            bool unique() const;
            operator unspecified-bool-type() const;
    };
}
```

Draft
Specializations of `shared_ptr` shall be CopyConstructible, Assignable, 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.

```c++
// Example:
if(shared_ptr<X> px = dynamic_pointer_cast<X>(py))
{
    // do something with px
}
```

### 20.6.6.2.1 `shared_ptr` constructors

```c++
shared_ptr();
```

**Effects:** Constructs an *empty* `shared_ptr` object.
Postconditions: use_count() == 0 && get() == 0.

Throws: nothing.

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.

Throws: bad_alloc, or an implementation-defined exception when a resource other than memory could not be obtained.

Exception safety: If an exception is thrown, delete p is called.

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);

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 (20.1.2). The copy constructor and destructor of A shall not throw exceptions.

Effects: Constructs a shared_ptr object that owns the pointer p and the deleter d. The second constructor shall use a copy of a to allocate memory for internal use.

Postconditions: use_count() == 1 && get() == p.

Throws: bad_alloc, or an implementation-defined exception when a resource other than memory could not be obtained.

Exception safety: If an exception is thrown, d(p) is called.

template<class Y> shared_ptr(shared_ptr<Y> const& r, T *p);

Effects: Constructs a shared_ptr instance that stores p and shares ownership with r.

Postconditions: get() == p && use_count() == r.use_count()

Throws: nothing.

[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]

[Note: this constructor allows creation of an empty shared_ptr instance with a non-NULL stored pointer. —end note]

shared_ptr(shared_ptr const& r);

template<class Y> shared_ptr(shared_ptr<Y> const& r);

Requires: For the second constructor Y* shall be convertible to T*.
Effects: If \( r \) is empty, constructs an empty \texttt{shared\_ptr} object; otherwise, constructs a \texttt{shared\_ptr} object that shares ownership with \( r \).

\texttt{shared\_ptr(shared\_ptr&& r);} 
\texttt{template<class Y> shared\_ptr(shared\_ptr<Y>&& r);} 

\textbf{Requires:} For the second constructor \( Y* \) shall be convertible to \( T* \).

\textbf{Effects:} Move-constructs a \texttt{shared\_ptr} instance from \( r \).

\textbf{Postconditions:} \( \ast \texttt{this} \) shall contain the old value of \( r \). \( r \) shall be empty.

\textbf{Throws:} nothing.

\texttt{template<class Y> explicit shared\_ptr(weak\_ptr<Y> const& r);} 

\textbf{Requires:} \( Y* \) shall be convertible to \( T* \).

\textbf{Effects:} Constructs a \texttt{shared\_ptr} object that shares ownership with \( r \) and stores a copy of the pointer stored in \( r \).

\textbf{Postconditions:} \( \texttt{use\_count()} == r.use\_count() \).

\textbf{Throws:} \texttt{bad\_weak\_ptr} when \( r.expired() \).

\textbf{Exception safety:} If an exception is thrown, the constructor has no effect.

\texttt{template<class Y> shared\_ptr(auto\_ptr<Y>& r);} 

\textbf{Requires:} \( r.release() \) shall be convertible to \( T* \). \( Y \) shall be a complete type. The expression \( \texttt{delete} \ r.release() \) shall be well-formed, shall have well defined behavior, and shall not throw exceptions.

\textbf{Effects:} Constructs a \texttt{shared\_ptr} object that stores and owns \( r.release() \).

\textbf{Postconditions:} \( \texttt{use\_count()} == 1 \&\& r.get() == 0 \).

\textbf{Throws:} \texttt{bad\_alloc}, or an implementation-defined exception when a resource other than memory could not be obtained.

\textbf{Exception safety:} If an exception is thrown, the constructor has no effect.

\textbf{20.6.6.2.2 shared\_ptr destructor} 

\texttt{\textasciitilde shared\_ptr();}

\textbf{Effects:}

\begin{itemize}
  \item If \( \ast \texttt{this} \) is empty or shares ownership with another \texttt{shared\_ptr} instance (\( \texttt{use\_count()} > 1 \)), there are no side effects.
  \item Otherwise, if \( \ast \texttt{this} \) owns a pointer \( p \) and a deleter \( d \), \( d(p) \) is called.
\end{itemize}
— Otherwise, \*this owns a pointer p, and delete p is called.

Throws: nothing.

[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]

20.6.6.2.3 shared_ptr assignment

shared_ptr& operator=(shared_ptr const& r);
template<class Y> shared_ptr& operator=(shared_ptr<Y> const& r);
template<class Y> shared_ptr& operator=(auto_ptr<Y>& r);

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]

shared_ptr& operator=(shared_ptr&& r);
template<class Y> shared_ptr& operator=(shared_ptr<Y>&& r);

Effects: Equivalent to shared_ptr(std::move(r)).swap(*this).

Returns: *this.

20.6.6.2.4 shared_ptr modifiers

void swap(shared_ptr& r);

Effects: Exchanges the contents of \*this and r.

Throws: nothing.

void reset();

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)`.

```cpp
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)`.

### 20.6.6.2.5 `shared_ptr` observers

#### `get()` const

`get()` const;  
Returns: the stored pointer. Returns a null pointer if `*this` is empty.  
Throws: nothing.

#### `operator*()` const

`operator*()` const;  
Requires: `get()` != 0.  
Returns: `*get()`.  
Throws: nothing.

#### `operator->()` const

`operator->()` const;  
Requires: `get()` != 0.  
Returns: `get()`.  
Throws: nothing.

#### `use_count()` const

`use_count()` const;  
Returns: the number of `shared_ptr` objects, `*this` included, that share ownership with `*this`, or 0 when `*this` is empty.  
Throws: nothing.  
[Note: `use_count()` is not necessarily efficient. — end note]

#### `unique()` const

`unique()` const;  
Returns: `use_count()` == 1.  
Throws: nothing.  
[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() == 0`. — end note]

#### `operator unspecified-bool-type()` const

Draft
Returns: an unspecified value that, when used in boolean contexts, is equivalent to get() != 0.

Throws: nothing.

[Note: This conversion operator allows shared_ptr objects to be used in boolean contexts. [Example: if (p && p->valid()) — end example] One possible choice for the return type is a pointer to member function, which avoids many of the implicit conversion pitfalls of a bool or void* return type. — end note]

20.6.6.2.6 shared_ptr creation

\[
\text{template\langle class T, class... Args\rangle shared\_ptr\langle T\rangle \ make\_shared(Args&&... args);} \\
\text{template\langle class T, class A, class... Args\rangle} \\
\text{shared\_ptr\langle T\rangle \ allocate\_shared(const A& a, Args&&... args);} \\
\]

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 (20.1.2). 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() or 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 are encouraged, but not required, to 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]

20.6.6.2.7 shared_ptr comparison

\[
\text{template\langle class T, class U\rangle bool operator==(shared\_ptr\langle T\rangle \ const\& a, shared\_ptr\langle U\rangle \ const\& b);} \\
\text{template\langle class T, class U\rangle bool operator!=(shared\_ptr\langle T\rangle \ const\& a, shared\_ptr\langle U\rangle \ const\& b);} \\
\text{template\langle class T, class U\rangle bool operator<(shared\_ptr\langle T\rangle \ const\& a, shared\_ptr\langle U\rangle \ const\& b);} \\
\]

Returns: a.get() == b.get().

Throws: nothing.

Returns: a.get() != b.get().

Throws: nothing.

Returns: an unspecified value such that

— operator< is a strict weak ordering as described in 25.3;
— under the equivalence relation defined by \( \text{operator}<, ! (a < b) \&\& ! (b < a) \), two `shared_ptr` instances are equivalent if and only if they `share ownership` or are both empty.

**Throws:** nothing.

[ Note: Defining a comparison operator allows `shared_ptr` objects to be used as keys in associative containers. — end note ]

### 20.6.6.2.8 shared_ptr I/O

```cpp
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`.

### 20.6.6.2.9 shared_ptr specialized algorithms

```cpp
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b);
```

**Effects:** Equivalent to \( a \text{.swap}(b) \).

**Throws:** nothing.

### 20.6.6.2.10 shared_ptr casts

```cpp
template<class T, class U> shared_ptr<T> static_pointer_cast(shared_ptr<U> const& r);
```

**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 \( r \).

**Throws:** nothing.

[ 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 ]

```cpp
template<class T, class U> shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const& r);
```

**Requires:** The expression `dynamic_cast<T*>(r.get())` shall be well-formed and shall have well defined behavior.

**Returns:**

— 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 \( r \);

— Otherwise, an `empty` `shared_ptr<T>` object.
20.6 Memory

Throws: nothing.

[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(shared_ptr<U> const& r);
```

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`.

Throws: nothing.

[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]

20.6.6.2.11 get_deleter

```
template<class D, class T> D* get_deleter(shared_ptr<T> const& p);
```

Returns: If `p` owns a deleter `d` of type cv-unqualified `D`, returns `&d`; otherwise returns 0. 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]

Throws: nothing.

20.6.6.3 Class template weak_ptr

The `weak_ptr` class template stores a weak reference to an object that is already managed by a `shared_ptr`. To access the object, a `weak_ptr` can be converted to a `shared_ptr` using the member function `lock`.

```
namespace std {
    template<class T> class weak_ptr
    {
        public:
            typedef T element_type;

            // constructors
            weak_ptr();
            template<class Y> weak_ptr(shared_ptr<Y> const& r);
            weak_ptr(weak_ptr const& r);
            template<class Y> weak_ptr(weak_ptr<Y> const& r);

            // destructor
            ~weak_ptr();

            // assignment
            weak_ptr& operator=(weak_ptr const& r);
            template<class Y> weak_ptr& operator=(weak_ptr<Y> const& r);

        } // end class weak_ptr

    } // end template
```

Draft
template<class Y> weak_ptr& operator=(shared_ptr<Y> const& r);

// modifiers
void swap(weak_ptr& r);
void reset();

// observers
long use_count() const;
bool expired() const;
shared_ptr<T> lock() const;
};

// comparison
template<class T, class U> bool operator<(weak_ptr<T> const& a, weak_ptr<U> const& b);

// specialized algorithms
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b);
}

namespace std

Specializations of weak_ptr shall be CopyConstructible, Assignable, and LessThanComparable, allowing their use in standard containers. The template parameter T of weak_ptr may be an incomplete type.

20.6.6.3.1 weak_ptr constructors

weak_ptr();

Effects: Constructs an empty weak_ptr object.

Postconditions: use_count() == 0.

Throws: nothing.

template<class Y> weak_ptr(shared_ptr<Y> const& r);
weak_ptr(weak_ptr const& r);
template<class Y> weak_ptr(weak_ptr<Y> const& r);

Requires: For the first and third constructors, Y* shall be 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().

Throws: nothing.

20.6.6.3.2 weak_ptr destructor

~weak_ptr();

Effects: Destroys this weak_ptr object but has no effect on the object its stored pointer points to.

Throws: nothing.
20.6.6.3 weak_ptr assignment

weak_ptr& operator=(weak_ptr const& r);
template<class Y> weak_ptr& operator=(weak_ptr<Y> const& r);
template<class Y> weak_ptr& operator=(shared_ptr<Y> const& r);

1 Effects: Equivalent to weak_ptr(r).swap(*this).
2 Throws: nothing.
3 Remarks: The implementation may meet the effects (and the implied guarantees) via different means, without creating a temporary.

20.6.6.4 weak_ptr modifiers

void swap(weak_ptr& r);

1 Effects: Exchanges the contents of *this and r.
2 Throws: nothing.

void reset();

3 Effects: Equivalent to weak_ptr().swap(*this).

20.6.6.5 weak_ptr observers

long use_count() const;

1 Returns: 0 if *this is empty; otherwise, the number of shared_ptr instances that share ownership with *this.
2 Throws: nothing.
3 [ Note: use_count() is not necessarily efficient. — end note ]

bool expired() const;

4 Returns: use_count() == 0.
5 Throws: nothing.
6 [ Note: expired() may be faster than use_count(). — end note ]

shared_ptr<T> lock() const;

7 Returns: expired() ? shared_ptr<T>() : shared_ptr<T>(*this).
8 Throws: nothing.

20.6.6.6 weak_ptr comparison

template<class T, class U> bool operator<(weak_ptr<T> const& a, weak_ptr<U> const& b);
Returns: an unspecified value such that
— operator< is a strict weak ordering as described in 25.3;
— under the equivalence relation defined by operator<, !(a < b) && !(b < a), two weak_ptr instances are equivalent if and only if they share ownership or are both empty.

Throws: nothing.

[Note: Allows weak_ptr objects to be used as keys in associative containers. —end note ]

20.6.6.3.7 weak_ptr specialized algorithms

[util.smartptr.weak.spec]

template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b)

Effects: Equivalent to a.swap(b).

Throws: nothing.

20.6.6.4 Class template enable_shared_from_this

[util.smartptr.enab]

A class T can inherit from enable_shared_from_this<T> to inherit the shared_from_this member functions that obtain a shared_ptr instance pointing to *this.

[Example:

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) && !(q < p)); // p and q share ownership
}

—end example ]

namespace std {
    template<class T> class enable_shared_from_this {
    protected:
        enable_shared_from_this();
        enable_shared_from_this(enable_shared_from_this const&);
        enable_shared_from_this& operator=(enable_shared_from_this const&);
        "enable_shared_from_this();
    public:
        shared_ptr<T> shared_from_this();
        shared_ptr<T const> shared_from_this() const;
    };
} // namespace std

Draft
The template parameter T of `enable_shared_from_this` may be an incomplete type.

```cpp
equal_shared_from_this();
equal_shared_from_this(equal_shared_from_this<T> const&);
```

**Effects:** Constructs an `equal_shared_from_this<T>` object.

**Throws:** nothing.

```cpp
equal_shared_from_this<T>& operator=(equal_shared_from_this<T> const&);
```

**Returns:** *this.

**Throws:** nothing.

```cpp
~equal_shared_from_this();
```

**Effects:** Destroys *this.

**Throws:** nothing.

```cpp
shared_ptr<T> shared_from_this();
shared_ptr<T const> shared_from_this() const;
```

**Requires:** `equal_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:

```cpp
template<class T> class enable_shared_from_this
{
private:
    weak_ptr<T> __weak_this;

protected:
    enable_shared_from_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]
void *align(std::size_t alignment, std::size_t size, void *ptr, std::size_t &space);

Effects: If it is possible to fit size bytes of storage aligned by alignment into the buffer pointed to by ptr with length space, the function updates ptr to 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.

Requires:
— alignment shall be a fundamental alignment value or an extended alignment value supported by the implementation in this context
— ptr shall point to contiguous storage of at least space bytes

Returns: a null pointer if the requested aligned buffer would not fit into the available space, otherwise the adjusted value of ptr.

[Note: the function updates its ptr and space arguments so that it can be called repeatedly with possibly different alignment and size arguments for the same buffer.]

### 20.6.8 C Library
[c.malloc]

1 Header <cstdlib> (Table 52):

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions:</td>
<td>calloc malloc free realloc</td>
</tr>
</tbody>
</table>

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() (18.5).

4 The function free() does not attempt to deallocate storage by calling ::operator delete().

See also: ISO C clause 7.11.2.

5 Header <cstring> (Table 53):

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td>NULL</td>
</tr>
<tr>
<td>Type:</td>
<td>size_t</td>
</tr>
<tr>
<td>Functions:</td>
<td>memchr memcmp memcmp memcpy memmove memset</td>
</tr>
</tbody>
</table>

6 The contents are the same as the Standard C library header <string.h>, with the change to memchr() specified in 21.4.
20.7 Date and time

SEE ALSO: ISO C clause 7.11.2.

20.7 Date and time

1 Header <ctime> (Table 54):

2 The contents are the same as the Standard C library header <time.h>.\(^\text{226}\)

SEE ALSO: ISO C clause 7.12, Amendment 1 clause 4.6.4.

---

\(^{226}\) \textit{strftime} supports the C99 conversion specifiers C, D, e, F, g, G, h, r, R, t, T, u, V, and z, and the modifiers E and O.
Chapter 21  Strings library [strings]

This clause describes components for manipulating sequences of any POD literal (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.

The following subclauses describe a character traits class, a string class, and null-terminated sequence utilities, as summarized in Table 55:

Table 55: Strings library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.1 Character traits</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>21.2 String classes</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>21.4 Null-terminated sequence</td>
<td>&lt;cstring&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cwchar&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cwctype&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cctype&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cuchar&gt;</td>
</tr>
</tbody>
</table>

21.1 Character traits [char.traits]

This subclause defines requirements on classes representing character traits, and defines a class template char_traits<charT>, along with four specializations, char_traits<char>, char_traits<char16_t>, char_traits<char32_t>, and char_traits<wchar_t>, that satisfy those requirements.

Most classes specified in clauses 21.2 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 guaranteed by these members.

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 formal parameters charT and traits. Traits::char_type shall be the same as CharT.

This subclause specifies a struct template, char_traits<charT>, and two explicit specializations of it, char_traits<char> and char_traits<wchar_t>, all of which appear in the header <string> and satisfy the requirements below.

21.1.1 Character traits requirements [char.traits.require]

In Table 56, 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.

Table 56: Traits requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>assertion/note</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::char_type</td>
<td>charT</td>
<td>(described in 21.1.2)</td>
<td>compile-time</td>
</tr>
<tr>
<td>X::int_type</td>
<td>(described in 21.1.2)</td>
<td>compile-time</td>
<td></td>
</tr>
<tr>
<td>X::off_type</td>
<td>(described in 21.1.2)</td>
<td>compile-time</td>
<td></td>
</tr>
<tr>
<td>X::pos_type</td>
<td>(described in 21.1.2)</td>
<td>compile-time</td>
<td></td>
</tr>
<tr>
<td>X::state_type</td>
<td>(described in 21.1.2)</td>
<td>compile-time</td>
<td></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>expression</td>
<td>return type</td>
<td>assertion/note pre/post condition</td>
<td>complexity</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>-------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td><code>X::not_eof(e)</code></td>
<td><code>int_type</code></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><code>X::to_char_type(e)</code></td>
<td><code>X::char_type</code></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>
<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 <code>to_char_type</code> and <code>eq_int_type</code>.</td>
<td>constant</td>
</tr>
<tr>
<td><code>X::eq_int_type(e, f)</code></td>
<td><code>bool</code></td>
<td>yields: for all ( c ) and ( d ), ( X::eq(c, d) ) is equal to ( X::eq(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.

### 21.1.2 traits typedefs

The type `char_type` is used to refer to the character container type in the implementation of the library classes defined in 21.2 and clause 27.

```cpp
typedef CHAR_T char_type;
```

1 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
21.1 Character traits

...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.227)

typedef OFF_T off_type;
typedef POS_T pos_type;

Requires: Requirements for off_type and pos_type are described in 27.1.2.

typedef STATE_T state_type;

Requires: state_type shall meet the requirements of Assignable (23.1), CopyConstructible (34), and DefaultConstructible types.

21.1.3 char_traits specializations

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);
        static constexpr bool eq(const char_type& c1, const char_type& c2);
        static constexpr bool lt(const char_type& c1, const char_type& c2);

        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);
    }
}

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 21.1.1.

21.1.3.1 struct char_traits<char>

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);
        static constexpr bool eq(const char_type& c1, const char_type& c2);
        static constexpr bool lt(const char_type& c1, const char_type& c2);

        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);
    }
}

227) If eof() can be held in char_type then some iostreams operations may give surprising results.
The defined types for int_type, pos_type, off_type, and state_type shall be int, streampos, streamoff, and mbstate_t respectively.

The type streampos shall be an implementation-defined type that satisfies the requirements for POS_T in 21.1.2.

The type streamoff shall be an implementation-defined type that satisfies the requirements for OFF_T in 21.1.2.

The type mbstate_t is defined in <cwchar> and can represent any of the conversion states that can occur in an implementation-defined set of supported multibyte character encoding rules.

The two-argument member assign 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.

The member eof() shall return EOF.

21.1.3.2 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);
      static constexpr int_type not_eof(const char_type& c);
      static char_type to_char_type(const int_type& c);
      static int_type to_int_type(const char_type& c);
      static bool eq_int_type(const int_type& c1, const int_type& c2);
      static constexpr int_type eof();
    }
  }
}

Draft
The type u16streampos shall be an implementation-defined type that satisfies the requirements for POS_T in 21.1.2.

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.

21.1.3.3 struct char_traits<char32_t> [char.traits.specializations.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);
        static constexpr bool eq(const char_type& c1, const char_type& c2);
        static constexpr bool lt(const char_type& c1, const char_type& c2);

        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(const char_type& c);
        static constexpr char_type to_char_type(const int_type& c);
        static constexpr int_type to_int_type(const char_type& c);
        static constexpr bool eq_int_type(const int_type& c1, const int_type& c2);
        static constexpr int_type eof();
    };
}

The type u32streampos shall be an implementation-defined type that satisfies the requirements for POS_T in 21.1.2.

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 point.

21.1.3.4 struct char_traits<wchar_t> [char.traits.specializations.wchar_t]

namespace std {
    template<>
    struct char_traits<wchar_t> {
        typedef wchar_t char_type;
        typedef unicode_int32_t int_type;
        typedef wcharoff off_type;
        typedef w32streampos pos_type;
        typedef mbstate_t state_type;

        static void assign(wchar_t& c1, const wchar_t& c2);
        static constexpr bool eq(wchar_t& c1, const wchar_t& c2);
        static constexpr bool lt(wchar_t& c1, const wchar_t& c2);

        static int compare(const wchar_t* s1, const wchar_t* s2, size_t n);
        static size_t length(const wchar_t* s);
        static const wchar_t* find(const wchar_t* s, size_t n, const wchar_t& a);
        static wchar_t* move(wchar_t* s1, const wchar_t* s2, size_t n);
        static wchar_t* copy(wchar_t* s1, const wchar_t* s2, size_t n);
        static wchar_t* assign(wchar_t* s, size_t n, wchar_t a);

        static constexpr int_type not_eof(const wchar_t& c);
        static constexpr wchar_t to_char_type(const int_type& c);
        static constexpr int_type to_int_type(const wchar_t& c);
        static constexpr bool eq_int_type(const int_type& c1, const int_type& c2);
        static constexpr wchar_t to_wchar_t(const char_type& c);
        static constexpr char_type to_char_type(const int_type& c);
        static constexpr int_type to_int_type(const wchar_t& c);
        static constexpr bool eq_int_type(const int_type& c1, const int_type& c2);
        static constexpr wchar_t eof();
    };
}
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);
    static constexpr bool eq(const char_type& c1, const char_type& c2);
    static constexpr bool lt(const char_type& c1, const char_type& c2);
    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(const char_type& c);
    static constexpr char_type to_char_type(const int_type& c);
    static constexpr int_type to_int_type(const char_type& c);
    static constexpr bool eq_int_type(const int_type& c1, const int_type& c2);
    static constexpr int_type eof();
};

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_T in 21.1.2.
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.
4 The two-argument members assign, eq, and lt shall be defined identically to the built-in operators =, ==, and < respectively.
5 The member eof() shall return WEOF.

21.2 String classes

The header <string> defines the basic_string class template for manipulating varying-length sequences of character-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

namespace std {
    // 21.1. 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>;

// 21.3, basic_string:
template<class charT, class traits = char_traits<charT>,
class Allocator = allocator<charT> >
class basic_string;

template<class charT, class traits, class Allocator>
  basic_string<
    charT, traits, Allocator>
  operator+(const basic_string<
    charT, traits, Allocator>& & lhs,
    const basic_string<
    charT, traits, Allocator>& & rhs);

template<class charT, class traits, class Allocator>
  basic_string<
    charT, traits, Allocator>&&
  operator+(basic_string<
    charT, traits, Allocator>&& lhs,
    const basic_string<
    charT, traits, Allocator>& & rhs);

template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>&&
  operator+(const basic_string<charT, traits, Allocator>& & lhs,
    basic_string<charT, traits, Allocator>& & rhs);

template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>&&
  operator+(basic_string<charT, traits, Allocator>&& lhs,
    basic_string<charT, traits, Allocator>& & rhs);

template<class charT, class traits, class Allocator>
  basic_string<
    charT, traits, Allocator>
  operator+(const charT* lhs,
    const basic_string<
    charT, traits, Allocator>& & rhs);

template<class charT, class traits, class Allocator>
  basic_string<
    charT, traits, Allocator>&&
  operator+(const charT* lhs,
    basic_string<charT, traits, Allocator>& & rhs);

template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
  operator+(charT lhs, const basic_string<charT, traits, Allocator>& & rhs);

template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>&&
  operator+(charT lhs, basic_string<charT, traits, Allocator>& & rhs);

template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
  operator+(const basic_string<charT, traits, Allocator>& & lhs,
    const charT* rhs);

template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>&&
  operator+(basic_string<charT, traits, Allocator>& & lhs,
    const charT* rhs);

template<class charT, class traits, class Allocator>
  basic_string<
    charT, traits, Allocator>
  operator+(const charT* lhs,
    basic_string<charT, traits, Allocator>& & rhs);

  basic_string<charT, traits, Allocator>&&
  operator+(basic_string<charT, traits, Allocator>& & lhs,
    const charT* rhs);

template<class charT, class traits, class Allocator>
  basic_string<
    charT, traits, Allocator>
  operator+(const charT* lhs,
    basic_string<charT, traits, Allocator>&&
basic_string<charT,traits,Allocator
  operator+(const basic_string<charT,traits,Allocator>& lhs, charT rhs);

template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>&
  operator+(basic_string<charT,traits,Allocator>&& lhs, charT rhs);

template<class charT, class traits, class Allocator>
  bool operator==(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);

template<class charT, class traits, class Allocator>
  bool operator==(const charT* lhs, const basic_string<charT,traits,Allocator>& rhs);

template<class charT, class traits, class Allocator>
  bool operator==(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);

template<class charT, class traits, class Allocator>
  bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);

template<class charT, class traits, class Allocator>
  bool operator!=(const charT* lhs, const basic_string<charT,traits,Allocator>& rhs);

template<class charT, class traits, class Allocator>
  bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);

template<class charT, class traits, class Allocator>
  bool operator<( const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);

template<class charT, class traits, class Allocator>
  bool operator<( const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);

template<class charT, class traits, class Allocator>
  bool operator<( const charT* lhs, const basic_string<charT,traits,Allocator>& rhs);

template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);

template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);

Draft
template<class charT, class traits, class Allocator>
bool operator<=(const charT* lhs,
const basic_string<charT,traits,Allocator>& rhs);

template<class charT, class traits, class Allocator>
bool operator>=(const basic_string<charT,traits,Allocator>& lhs,
const charT* rhs);

template<class charT, class traits, class Allocator>
bool operator>=(const basic_string<charT,traits,Allocator>& lhs,
const basic_string<charT,traits,Allocator>& rhs);

template<class charT, class traits, class Allocator>
bool operator>=(const charT* lhs,
const basic_string<charT,traits,Allocator>& rhs);

// 21.3.8.8:
template<class charT, class traits, class Allocator>
void swap(basic_string<charT,traits,Allocator>& lhs,
basic_string<charT,traits,Allocator>& rhs);

template<class charT, class traits, class Allocator>
void swap(basic_string<charT,traits,Allocator>&& lhs,
basic_string<charT,traits,Allocator>&& rhs);

template<class charT, class traits, class Allocator>
void swap(basic_string<charT,traits,Allocator>& lhs,
basic_string<charT,traits,Allocator>&& rhs);

template<class charT, class traits, class Allocator>
basic_istream<charT,traits>&
operator>>(basic_istream<charT,traits>&& is,
basic_string<charT,traits,Allocator>& str);

template<class charT, class traits, class Allocator>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>&& os,
const basic_string<charT,traits,Allocator>& str);

template<class charT, class traits, class Allocator>
basic_istream<charT,traits>&
getline(basic_istream<charT,traits>&& is,
basic_string<charT,traits,Allocator>& str,
charT delim);

typedef basic_string<char> string;
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
typedef basic_string<wchar_t> wstring;
}
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. 

The class template `basic_string` conforms to the requirements for a Sequence (23.1.1) and for a Reversible Container (23.1). Thus, the iterators supported by `basic_string` are random access iterators (24.1.5).

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` (19.1.4);
- an out-of-range error is associated with exceptions of type `out_of_range` (19.1.5).

```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::size_type size_type;
            typedef typename Allocator::difference_type difference_type;
            typedef typename Allocator::reference reference;
            typedef typename Allocator::const_reference const_reference;
            typedef typename Allocator::pointer pointer;
            typedef typename Allocator::const_pointer const_pointer;

            typedef implementation-defined iterator;     // See 23.1
            typedef implementation-defined const_iterator;  // See 23.1
            typedef reverse_iterator<iterator> reverse_iterator;
            typedef reverse_iterator<const_iterator> const_reverse_iterator;
            static const size_type npos = -1;

            // 21.3.2 construct/copy/destroy:
            explicit basic_string(const Allocator& a = Allocator());
            basic_string(const basic_string& str);
            basic_string(basic_string&& str);
            basic_string(const basic_string& str, size_type pos, size_type n = npos,
                          const Allocator& a = Allocator());

    } // basic_string

    // Note: Allocator::value_type must name the same type as charT (21.3.1). — end note
}
```

Draft
21.3  Class template basic_string

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();
basic_string& operator=(const basic_string& str);
basic_string& operator=(basic_string&& str);
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);

// 21.3.3 iterators:
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// 21.3.4 capacity:
size_type size() const;
size_type length() const;
size_type max_size() const;
void resize(size_type n, charT c);
void resize(size_type n);
size_type capacity() const;
void reserve(size_type res_arg = 0);
void clear();
bool empty() const;

// 21.3.5 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();

Draft
// 21.3.6 modifiers:
basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str, size_type pos, size_type n);
basic_string& append(const charT* s, size_type n);
basic_string& append(charT c);
template<class InputIterator>
basic_string& append(InputIterator first, InputIterator last);
void push_back(charT c);

basic_string& assign(const basic_string& str);
basic_string& assign(basic_string&& str);
basic_string& assign(const basic_string& str, size_type pos, size_type n);
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& insert(size_type pos1, const basic_string& str);
basic_string& insert(size_type pos1, const basic_string& str, size_type pos2, size_type n);
basic_string& insert(size_type pos, const charT* s, size_type n);
basic_string& insert(size_type pos, const charT* s);
iterator insert(iterator p, charT c);
void insert(iterator p, size_type n, charT c);
template<class InputIterator>
void insert(iterator p, InputIterator first, InputIterator last);

basic_string& erase(size_type pos = 0, size_type n = npos);
iterator erase(iterator position);
iterator erase(iterator first, 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);
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(iterator i1, iterator i2, const basic_string& str);
basic_string& replace(iterator i1, iterator i2, const charT* s, size_type n);
basic_string& replace(iterator i1, iterator i2, const charT* s);
basic_string& replace(iterator i1, iterator i2, size_type n, charT c);

template<class InputIterator>
basic_string& replace(iterator i1, iterator i2, InputIterator j1, InputIterator j2);

size_type copy(charT* s, size_type n, size_type pos = 0) const;
void swap(basic_string&& str);

size_type copy(charT* s, size_type n, size_type pos = 0) const;

// 21.3.7 string operations:
const charT* c_str() const; // explicit
cost charT* data() const;
allocator_type get_allocator() const;

size_type find (const basic_string& str, size_type pos = 0) const;
size_type find (const charT* s, size_type pos, size_type n) const;
size_type find (const charT* s, size_type pos = 0) const;
size_type find (charT c, size_type pos = 0) const;
size_type rfind(const basic_string& str, size_type pos = npos) const;
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(const charT* s, size_type pos = npos) const;
size_type find_first_of(const basic_string& str, size_type pos = 0) const;
size_type find_first_of(const charT* s, size_type pos, size_type n) const;
size_type find_first_of(const charT* s, size_type pos = 0) const;
size_type find_first_of(charT c, size_type pos = 0) const;
size_type find_last_of (const basic_string& str, size_type pos = npos) const;
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 (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;
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;
21.3.1 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. In every specialization basic_string<charT, traits, Allocator>, the nested type 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 a copy of the Allocator object passed to the basic_string object's constructor or, if the constructor does not take an Allocator argument, a copy of a default-constructed Allocator object.

3. The char-like objects in a basic_string object shall be stored contiguously. That is, for any basic_string object s, the identity *(s.begin() + n) == &s.begin() + n shall hold for all values of n such that 0 <= n < s.size().

4. References, pointers, and iterators referring to the elements of a basic_string sequence may be invalidated by the following uses of that basic_string object:
   - As an argument to non-member functions swap() (21.3.8.8), operator>>() (21.3.8.9), and getline() (21.3.8.9).
   - As an argument to basic_string::swap().
   - Calling data() and c_str() member functions. Calling non-const member functions, except operator[], at, begin, rbegin, end, and rend.
   - Following construction or any of the above uses, except the forms of insert and erase that return iterators, the first call to non-const member functions operator[], at, begin, rbegin, end, or rend.
21.3  Class template basic_string  

[Note: These rules are formulated to allow, but not require, a reference counted implementation. A reference counted implementation must have the same semantics as a non-reference counted implementation. [Example:

```cpp
string s1("abc");

string::iterator i = s1.begin();
string s2 = s1;

*i = 'a'; // Must modify only s1
```

— end example] — end note]

21.3.2  basic_string constructors and assignment operators  [string.cons]

```cpp
explicit basic_string(const Allocator& a = Allocator());
```

1  **Effects:** Constructs an object of class basic_string. The postconditions of this function are indicated in Table 57.

<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>

```cpp
basic_string(const basic_string<charT,traits,Allocator>& str);
basic_string(basic_string<charT,traits,Allocator>&& str);
```

2  **Effects:** Constructs an object of class basic_string as indicated in Table 58. In the first form, the stored Allocator value is copied from `str.get_allocator()`. In the second form, the stored Allocator value is move constructed from `str.get_allocator()`, and `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>data()</td>
<td>points at the first element of an allocated copy of the array whose first element is pointed at by <code>str.data()</code></td>
</tr>
<tr>
<td>size()</td>
<td><code>str.size()</code></td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as <code>size()</code></td>
</tr>
</tbody>
</table>

```cpp
basic_string(const basic_string<charT,traits,Allocator>& str,
             size_type pos, size_type n = npos,
             const Allocator& a = Allocator());
```

---

Draft
Requir: pos <= str.size()
Thros: out_of_range if pos > str.size().
Effets: 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 59.

Table 59: 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());
Requir: s shall not be a null pointer and n < npos.
Effets: 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 60.

Table 60: basic_string(const charT*, size_type, const Allocator&) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>points at the first element of an allocated copy of the array whose first element is pointed at by s</td>
</tr>
<tr>
<td>size()</td>
<td>n</td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as size()</td>
</tr>
</tbody>
</table>

basic_string(const charT* s, const Allocator& a = Allocator());
Requir: s shall not be a null pointer.
Effets: 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 61.

Table 61: basic_string(const charT*, const Allocator&) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>points at the first element of an allocated copy of the array whose first element is pointed at by s</td>
</tr>
<tr>
<td>size()</td>
<td>traits::length(s)</td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as size()</td>
</tr>
</tbody>
</table>

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 62:

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>points at the first element of an allocated array of n elements, each storing the initial value c</td>
</tr>
<tr>
<td>size()</td>
<td>n</td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as size()</td>
</tr>
</tbody>
</table>

template<class InputIterator>
basic_string(InputIterator begin, InputIterator end, const Allocator& a = Allocator());

Effects: If InputIterator is an integral type, equivalent to

    basic_string(static_cast<size_type>(begin), static_cast<value_type>(end), a)

Otherwise constructs a string from the values in the range [begin, end), as indicated in the Sequence Requirements table (see 23.1.1).

basic_string<charT, traits, Allocator>&
operator=(const basic_string<charT, traits, Allocator>& str);

Effects: If *this and str are not the same object, modifies *this as shown in Table 63. The second form leaves str in a valid but unspecified state. [Note: A valid implementation is swap(str). —end note]

If *this and str are the same object, the member has no effect.

Throw: Nothing.

Returns: *this

<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<charT, traits, Allocator>&
operator=(const basic_string<charT, traits, Allocator>&& str);
Effects: If *this and str are not the same object, modifies *this as shown in Table 64. The constructor leaves str in a valid but unspecified state. [Note: A valid implementation is swap(str). —end note]

Table 64: operator=(const basic_string<T, traits, Allocator>&&) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>points at the array whose first element was pointed at by str.data()</td>
</tr>
<tr>
<td>size()</td>
<td>previous value of str.size()</td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as size()</td>
</tr>
</tbody>
</table>

21.3.3 basic_string iterator support

Returns: an iterator referring to the first character in the string.

Returns: an iterator which is the past-the-end value.

Returns: an iterator which is semantically equivalent to reverse_iterator(end()).

Returns: an iterator which is semantically equivalent to reverse_iterator(begin()).
21.3 Class template basic_string

21.3.4 basic_string capacity

size_type size() const;
  
  Returns: a count of the number of char-like objects currently in the string.

size_type length() const;
  
  Returns: size().

size_type max_size() const;
  
  Returns: The maximum size of the string.
  
  Remark: See Container requirements table (23.1).

void resize(size_type n, charT c);
  
  Requires: n <= max_size()
  
  Throws: length_error if n > max_size().
  
  Effects: Alters the length of the string designated by *this as follows:

  — If n <= size(), the function replaces the string designated by *this with a string of length n whose elements are a copy of the initial elements of the original string designated by *this.
  
  — If n > size(), the function replaces the string designated by *this with a string of length n whose first size() elements are a copy of the original string designated by *this, and whose remaining elements are all initialized to c.

void resize(size_type n);
  
  Effects: resize(n, charT()).

size_type capacity() const;
  
  Returns: the size of the allocated storage in the string.

void reserve(size_type res_arg=0);
  
  The member function reserve() is a directive that informs a basic_string object of a planned change in size, so that it can manage the storage allocation accordingly.
  
  Effects: After reserve(), capacity() is greater or equal to the argument of reserve. [Note: Calling reserve() with a res_arg argument less than capacity() is in effect a non-binding shrink request. A call with res_arg <= size() is in effect a non-binding shrink-to-fit request. — end note]
  
  Throws: length_error if res_arg > max_size().

void clear();
  
  Effects: Behaves as if the function calls:

  erase(begin(), end());

---

229) reserve() uses Allocator::allocate() which may throw an appropriate exception.
bool empty() const;

Returns: size() == 0.

21.3.5 basic_string element access

const_reference operator[](size_type pos) const;
reference operator[](size_type pos);

Returns: If pos < size(), returns *(begin() + pos). Otherwise, if pos == size(), the const version
returns charT(). Otherwise, the behavior is undefined.

const_reference at(size_type pos) const;
reference at(size_type pos);

Requires: pos < size()

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).

21.3.6 basic_string modifiers

21.3.6.1 basic_string::operator+=

basic_string<charT,traits,Allocator>&
  operator+=(const basic_string<charT,traits,Allocator>& str);

Returns: append(str).

basic_string<charT,traits,Allocator>& operator+=(const charT* s);

Returns: *this += basic_string<charT,traits,Allocator>(s).

Remarks: Uses traits::length().

basic_string<charT,traits,Allocator>& operator+=(charT c);

Returns: *this += basic_string<charT,traits,Allocator>(1,c).
21.3.6.2 basic_string::append

basic_string<charT,traits,Allocator>&
    append(const basic_string<charT,traits>& str);

Returns: append(str, 0, npos).

basic_string<charT,traits,Allocator>&
    append(const basic_string<charT,traits>& str, size_type pos, size_type n);

Requires: pos <= str.size()

Throws: out_of_range if pos > str.size().

Effects: Determines the effective length rlen of the string to append as the smaller of n and str.size() - pos. The function then throws length_error if size() >= npos - rlen. Otherwise, the function replaces the string controlled by *this with a string of length size() + rlen whose first size() elements are a copy of the original string controlled by *this and whose remaining elements are a copy of the initial elements of the string controlled by str beginning at position pos.

Returns: *this.

basic_string<charT,traits,Allocator>&
    append(const charT* s, size_type n);

Returns: append(basic_string<charT,traits,Allocator>(s, n)).

basic_string<charT,traits,Allocator>&
    append(const charT* s);

Returns: append(basic_string<charT,traits,Allocator>(s)).

Remarks: Uses traits::length().

basic_string<charT,traits,Allocator>&
    append(size_type n, charT c);

Returns: append(basic_string<charT,traits,Allocator>(n, c)).

template<class InputIterator>
    basic_string& append(InputIterator first, InputIterator last);

Returns: append(basic_string<charT,traits,Allocator>(first, last)).

void push_back(charT c)

Effects: Equivalent to append(static_cast<size_type>(1), c).

21.3.6.3 basic_string::assign

basic_string<charT,traits,Allocator>&
    assign(const basic_string<charT,traits>& str);

Returns: assign(str, 0, npos).
basic_string<charT,traits,Allocator>&
assign(const basic_string<charT,traits>&& str);

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. Leaves str in a valid but unspecified state. [Note: A valid implementation is swap(str). — end note ]

Throws: Nothing.

Returns: *this.

basic_string<charT,traits,Allocator>&
assign(const basic_string<charT,traits>& str, size_type pos, size_type n);

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.

The function then replaces the string controlled by *this with a string of length rlen whose elements are a copy of the string controlled by str beginning at position pos.

Returns: *this.

basic_string<charT,traits,Allocator>&
assign(const charT* s, size_type n);

Returns: assign(basic_string<charT,traits,Allocator>(s,n)).

basic_string<charT,traits,Allocator>& assign(const charT* s);

Returns: assign(basic_string<charT, traits, Allocator>(s)).

Remarks: Uses traits::length().

basic_string<charT,traits,Allocator>&
assign(size_type n, charT c);

Returns: assign(basic_string<charT,traits,Allocator>(n,c)).

template<class InputIterator>
basic_string& assign(InputIterator first, InputIterator last);

Returns: assign(basic_string<charT,traits,Allocator>(first,last)).

21.3.6.4 basic_string::insert

basic_string<charT,traits,Allocator>&
insert(size_type pos1, const basic_string<charT,traits,Allocator>& str);

Returns: insert(pos1,str,0,npos).
21.3 Class template `basic_string`  

`basic_string<charT,traits,Allocator>`

`insert(size_type pos1,`  
`   const basic_string<charT,traits,Allocator>& str,`  
`   size_type pos2, size_type n);`

`Requires: pos1 <= size() and pos2 <= str.size()`

`Throws: out_of_range if pos1 > size() or pos2 > str.size().`

`Effects: Determines the effective length `rlen` of the string to insert as the smaller of `n` and `str.size() - pos2`. Then throws length_error if size() >= npos - rlen. Otherwise, the function replaces the string controlled by `*this` with a string of length size() + rlen whose first pos1 elements are a copy of the initial elements of the original string controlled by `*this`, whose next rlen elements are a copy of the elements of the string controlled by `str` beginning at position pos2, and whose remaining elements are a copy of the remaining elements of the original string controlled by `*this`.

`Returns: *this.`

`basic_string<charT,traits,Allocator>`

`insert(size_type pos, const charT* s, size_type n);`

`Returns: insert(pos,basic_string<charT,traits,Allocator>(s,n)).`

`basic_string<charT,traits,Allocator>`

`insert(size_type pos, const charT* s);`

`Returns: insert(pos,basic_string<charT,traits,Allocator>(s)).`

`Remarks: Uses traits::length().`

`basic_string<charT,traits,Allocator>`

`insert(size_type pos, size_type n, charT c);`

`Returns: insert(pos,basic_string<charT,traits,Allocator>(n,c)).`

`iterator insert(iterator p, charT c);`

`Requires: p is a valid iterator on *this.`

`Effects: inserts a copy of c before the character referred to by p.`

`Returns: an iterator which refers to the copy of the inserted character.`

`void insert(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.`

`template<class InputIterator>`

`void insert(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)).`
21.3.6.5 basic_string::erase [string::erase]

    basic_string<charT,traits,Allocator>&
    erase(size_type pos = 0, size_type n = npos);

1    Requires: pos <= size()
2    Throws: out_of_range if pos > size().
3    Effects: Determines the effective length xlen of the string to be removed as the smaller of n and size() - pos.
4    The function then replaces the string controlled by *this with a string of length size() - xlen whose first
5    pos elements are a copy of the initial elements of the original string controlled by *this, and whose remaining
6    elements are a copy of the elements of the original string controlled by *this beginning at position pos + xlen.
7    Returns: *this.

    iterator erase(iterator p);
8    Effects: removes the character referred to by p.
9    Returns: an iterator which points to the element immediately following p prior to the element being erased. If no
10   such element exists, end() is returned.

    iterator erase(iterator first, iterator last);
11   Requires: first and last are valid iterators on *this, defining a range [first, last).
12   Effects: removes the characters in the range [first, last).
13   Returns: an iterator which points to the element pointed to by last prior to the other elements being erased. If
14   no such element exists, end() is returned.

    void pop_back();
15   Requires: !empty()
16   Effects: Equivalent to erase(size() - 1, 1).

21.3.6.6 basic_string::replace [string::replace]

    basic_string<charT,traits,Allocator>&
    replace(size_type pos1, size_type n1, const basic_string<charT,traits,Allocator>& str);
1    Returns: replace(pos1, n1, str, 0, npos).

    basic_string<charT,traits,Allocator>&
    replace(size_type pos1, size_type n1, const basic_string<charT,traits,Allocator>& str,
             size_type pos2, size_type n2);
2    Requires: pos1 <= size() && pos2 <= str.size().
21.3 Class template `basic_string`

**Strings library 572**

3 *Throws:* out_of_range if `pos1 > size()` or `pos2 > str.size()`, or length_error if the length of the resulting string would exceed `max_size()` (see below).

4 *Effects:* Determines the effective length `xlen` of the string to be removed as the smaller of `n1` and `size() - pos1`. Also determines the effective length `rlen` of the string to be inserted as the smaller of `n2` and `str.size() - pos2`. If `size() - zlen >= max_size() - rlen`, throws length_error. Otherwise, the function replaces the string controlled by \*this with a string of length `size() - zlen + rlen` whose first `pos1` elements are a copy of the initial elements of the original string controlled by \*this, whose next `rlen` elements are a copy of the initial elements of the string controlled by `str` beginning at position `pos2`, and whose remaining elements are a copy of the elements of the original string controlled by \*this beginning at position `pos1 + zlen`.

5 *Returns:* \*this.

```
    basic_string<charT,traits,Allocator>&
    replace(size_type pos, size_type n1, const charT* s, size_type n2);
```

6 *Returns:* `replace(pos, n1, basic_string<charT,traits,Allocator>(s, n2)).`

```
    basic_string<charT,traits,Allocator>&
    replace(size_type pos, size_type n1, const charT* s);
```

7 *Returns:* `replace(pos, n1, basic_string<charT,traits,Allocator>(s)).`

8 *Remarks:* Uses `traits::length()`.

```
    basic_string<charT,traits,Allocator>&
    replace(size_type pos, size_type n1, size_type n2, charT c);
```

9 *Returns:* `replace(pos, n1, basic_string<charT,traits,Allocator>(n2, c)).`

```
    basic_string& replace(iterator i1, iterator i2, const basic_string& str);
```

10 *Requires:* The iterators `i1` and `i2` are valid iterators on \*this, defining a range `[i1, i2)`. **Effects:** Replaces the string controlled by \*this with a string of length `size() - (i2 - i1) + str.size()` whose first `begin()` - `i1` elements are a copy of the initial elements of the original string controlled by \*this, whose next `str.size()` elements are a copy of the string controlled by `str`, and whose remaining elements are a copy of the elements of the original string controlled by \*this beginning at position `i2`.

11 *Returns:* \*this.

12 *Remarks:* After the call, the length of the string will be changed by: `str.size() - (i2 - i1)`.

```
    basic_string&
    replace(iterator i1, iterator i2, const charT* s, size_type n);
```

13 *Returns:* `replace(i1, i2, basic_string(s, n)).`

14 *Remarks:* Length change: `n = (i2 - i1)`.

```
    basic_string& replace(iterator i1, iterator i2, const charT* s);
```

Draft
Returns: replace(i1, i2, basic_string(s)).
Remarks: Length change: traits::length(s) - (i2 - i1).
Uses traits::length().

basic_string& replace(iterator i1, iterator i2, size_type n, charT c);
Returns: replace(i1, i2, basic_string(n, c)).
Remarks: Length change: n - (i2 - i1).

template<class InputIterator>
basic_string& replace(iterator i1, iterator i2, InputIterator j1, InputIterator j2);
Returns: replace(i1, i2, basic_string(j1, j2)).
Remarks: Length change: j2 - j1 - (i2 - i1).

21.3.6.7 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.

21.3.6.8 basic_string::swap

void swap(basic_string<charT,traits,Allocator>&& s);
Throws: Nothing.
Postcondition: *this contains the same sequence of characters that was in s, s contains the same sequence of characters that was in *this.
Complexity: constant time.
21.3.7 basic_string string operations

const charT* c_str() const;

1 Returns: A pointer to the initial element of an array of length size() + 1 whose first size() elements equal the corresponding elements of the string controlled by *this and whose last element is a null character specified by charT().

2 Requires: The program shall not alter any of the values stored in the array. Nor shall the program treat the returned value as a valid pointer value after any subsequent call to a non-const member function of the class basic_string that designates the same object as this.

const charT* data() const;

3 Returns: If size() is nonzero, the member returns a pointer to the initial element of an array whose first size() elements equal the corresponding elements of the string controlled by *this. If size() is zero, the member returns a non-null pointer that is copyable and can have zero added to it.

4 Requires: The program shall not alter any of the values stored in the character array. Nor shall the program treat the returned value as a valid pointer value after any subsequent call to a non-const member function of basic_string that designates the same object as this.

allocator_type get_allocator() const;

5 Returns: a copy of the Allocator object used to construct the string.

21.3.7.1 basic_string::find

size_type find(const basic_string<charT,traits,Allocator>& str, size_type pos = 0) const;

1 Effects: Determines the lowest position xpos, if possible, such that both of the following conditions obtain:

— pos <= xpos and xpos + str.size() <= size();
— traits::eq(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().

size_type find(const charT* s, size_type pos, size_type n) const;

4 Returns: find(basic_string<charT,traits,Allocator>(s,n),pos).

size_type find(const charT* s, size_type pos = 0) const;

5 Returns: find(basic_string<charT,traits,Allocator>(s),pos).

6 Remarks: Uses traits::length().

size_type find(charT c, size_type pos = 0) const;

7 Returns: find(basic_string<charT,traits,Allocator>(1,c),pos).

Draft
21.3.7.2  basic_string::rfind

size_type rfind(const basic_string<charT,traits,Allocator>& str,
    size_type pos = npos) const;

Effects: Determines the highest position xpos, if possible, such that both of the following conditions obtain:
- xpos <= pos and xpos + str.size() <= size();
- traits::eq(at(xpos+I), str.at(I)) for all elements 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 rfind(const charT* s, size_type pos, size_type n) const;

Returns: rfind(basic_string<charT,traits,Allocator>(s,n),pos).

size_type rfind(const charT* s, size_type pos = npos) const;

Returns: rfind(basic_string<charT,traits,Allocator>(s),pos).
Remarks: Uses traits::length().

size_type rfind(charT c, size_type pos = npos) const;

Returns: rfind(basic_string<charT,traits,Allocator>(1,c),pos).

21.3.7.3  basic_string::find_first_of

size_type
    find_first_of(const basic_string<charT,traits,Allocator>& str,
        size_type pos = 0) const;

Effects: Determines the lowest position xpos, if possible, such that both of the following conditions obtain:
- pos <= xpos and xpos < size();
- 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().

size_type
    find_first_of(const charT* s, size_type pos, size_type n) const;

Returns: find_first_of(basic_string<charT,traits,Allocator>(s,n),pos).

size_type find_first_of(const charT* s, size_type pos = 0) const;

Returns: find_first_of(basic_string<charT,traits,Allocator>(s),pos).
Remarks: Uses traits::length().
21.3 Class template basic_string

size_type find_first_of(charT c, size_type pos = 0) const;

    Returns: find_first_of(basic_string<charT,traits,Allocator>(1,c),pos).

21.3.7.4 basic_string::find_last_of

size_type
    find_last_of(const basic_string<charT,traits,Allocator>& str, 
                  size_type pos = npos) const;

1    Effects: Determines the highest position xpos, if possible, such that both of the following conditions obtain:
       — xpos <= pos and xpos < size();
       — traits::eq(at(xpos), str.at(I)) for some element 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().

size_type find_last_of(const charT* s, size_type pos, size_type n) const;

4    Returns: find_last_of(basic_string<charT,traits,Allocator>(s,n),pos).

size_type find_last_of(const charT* s, size_type pos = npos) const;

5    Returns: find_last_of(basic_string<charT,traits,Allocator>(s),pos).

6    Remarks: Uses traits::length().

size_type find_last_of(charT c, size_type pos = npos) const;

7    Returns: find_last_of(basic_string<charT,traits,Allocator>(1,c),pos).

21.3.7.5 basic_string::find_first_not_of

size_type
    find_first_not_of(const basic_string<charT,traits,Allocator>& str, 
                       size_type pos = 0) const;

1    Effects: Determines the lowest position xpos, if possible, such that both of the following conditions obtain:
       — pos <= xpos and xpos < size();
       — traits::eq(at(xpos), str.at(I)) for no element 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().

size_type find_first_not_of(const charT* s, size_type pos, size_type n) const;

4    Returns: find_first_not_of(basic_string<charT,traits,Allocator>(s,n),pos).
size_type find_first_not_of(const charT* s, size_type pos = 0) const;
5   \textit{Returns:} find_first_not_of(basic_string<charT,traits,Allocator>(s),pos).
6   \textit{Remarks:} Uses traits::length().

size_type find_first_not_of(charT c, size_type pos = 0) const;
7   \textit{Returns:} find_first_not_of(basic_string<charT,traits,Allocator>(1,c),pos).

21.3.7.6 basic_string::find_last_not_of

size_type
1   find_last_not_of(const basic_string<charT,traits,Allocator>& str, size_type pos = npos) const;
2   \textit{Effects:} Determines the highest position $xpos$, if possible, such that both of the following conditions obtain:
3      \begin{itemize}
4         \item $xpos \leq pos$ and $xpos < \text{size()}$;
5         \item traits::eq(at($xpos$), str.at($I$)) for no element $I$ of the string controlled by $str$.
6      \end{itemize}
7   \textit{Returns:} $xpos$ if the function can determine such a value for $xpos$. Otherwise, returns npos.
8   \textit{Remarks:} Uses traits::eq().

size_type find_last_not_of(const charT* s, size_type pos, size_type n) const;
9   \textit{Returns:} find_last_not_of(basic_string<charT,traits,Allocator>(s,n),pos).

size_type find_last_not_of(const charT* s, size_type pos = npos) const;
10  \textit{Returns:} find_last_not_of(basic_string<charT,traits,Allocator>(s),pos).
11  \textit{Remarks:} Uses traits::length().

size_type find_last_not_of(charT c, size_type pos = npos) const;
12  \textit{Returns:} find_last_not_of(basic_string<charT,traits,Allocator>(1,c),pos).

21.3.7.7 basic_string::substr

basic_string<charT,traits,Allocator>
substr(size_type pos = 0, size_type n = npos) const;
1   \textit{Requires:} \text{pos} \leq \text{size()}
2   \textit{Throws:} out_of_range if \text{pos} > \text{size()}.
3   \textit{Effects:} Determines the effective length $rlen$ of the string to copy as the smaller of $n$ and $\text{size()} - \text{pos}$.
4   \textit{Returns:} basic_string<charT,traits,Allocator>(data()+\text{pos},rlen).

Draft
21.3.7.8 basic_string::compare

```
int compare(const basic_string<charT,traits,Allocator>& str) const

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 65.
```

<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<charT,traits,Allocator>& str) const;

Returns:

    basic_string<charT,traits,Allocator>(*this, pos1, n1).compare(str).

int compare(size_type pos1, size_type n1,
             const basic_string<charT,traits,Allocator>& str,
             size_type pos2, size_type n2) const;

Returns:

    basic_string<charT,traits,Allocator>(*this, pos1, n1).compare(basic_string<charT,traits,Allocator>(str, pos2, n2)).

int compare(const charT *s) const;

Returns: this->compare(basic_string<charT,traits,Allocator>(s)).

int compare(size_type pos, size_type n1,
             const charT *s) const;

Returns:

    basic_string<charT,traits,Allocator>(*this, pos, n1).compare(basic_string<charT,traits,Allocator>(s))

int compare(size_type pos, size_type n1,
             const charT *s, size_type n2) const;

Returns:

    basic_string<charT,traits,Allocator>(*this, pos, n1).compare(basic_string<charT,traits,Allocator>(s, n2))
```
21.3.8 basic_string non-member functions

21.3.8.1 operator+

template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+(const basic_string<charT,traits,Allocator>& lhs, 
const basic_string<charT,traits,Allocator>& rhs);

Returns: basic_string<charT,traits,Allocator>(lhs).append(rhs)

template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>&&
operator+(basic_string<charT,traits,Allocator>&& lhs, 
const basic_string<charT,traits,Allocator>& rhs);

Returns: lhs.append(rhs)

template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>&&
operator+(const basic_string<charT,traits,Allocator>& lhs, 
basic_string<charT,traits,Allocator>&& rhs);

Returns: rhs.insert(0, lhs)

template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>&&
operator+(basic_string<charT,traits,Allocator>&& lhs, 
basic_string<charT,traits,Allocator>&& rhs);

Returns: lhs.append(rhs) [Note: Or equivalently rhs.insert(0, lhs) — end note]

template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+(const charT* lhs, 
const basic_string<charT,traits,Allocator>& rhs);

Returns: basic_string<charT,traits,Allocator>(lhs) + rhs.

Remarks: Uses traits::length().

template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>&
operator+(const charT* lhs, 
basic_string<charT,traits,Allocator>&& rhs);

Returns: rhs.insert(0, lhs).

Remarks: Uses traits::length().

template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+(charT* lhs, 
const basic_string<charT,traits,Allocator>&& rhs);

Draft
21.3 Class template `basic_string`

Returns: `basic_string<charT,traits,Allocator>(1, lhs) + rhs`.

```cpp
template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>&
operator+(charT lhs,  
        basic_string<charT,traits,Allocator>&& rhs);
```

Returns: `rhs.insert(0, 1, lhs)`.

```cpp
template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>&
operator+(const basic_string<charT,traits,Allocator>& lhs,  
          const charT* rhs);
```

Returns: `lhs + basic_string<charT,traits,Allocator>(rhs)`.

Remarks: Uses `traits::length()`.

```cpp
template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>&
operator+(basic_string<charT,traits,Allocator>&& lhs,  
          const charT* rhs);
```

Returns: `lhs.append(rhs)`.

Remarks: Uses `traits::length()`.

```cpp
template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>&
operator+(basic_string<charT,traits,Allocator>&& lhs,  
          charT rhs);
```

Returns: `lhs.append(1, rhs)`.

21.3.8.2 `operator==`

Returns: `lhs.compare(rhs) == 0`.

```cpp
template<class charT, class traits, class Allocator>
bool operator==(const basic_string<charT,traits,Allocator>& lhs,  
                 const basic_string<charT,traits,Allocator>& rhs);
```

Returns: `lhs.compare(rhs) == 0`.

```cpp
template<class charT, class traits, class Allocator>
bool operator==(const charT* lhs,  
                const basic_string<charT,traits,Allocator>& rhs);
```
Returns: basic_string<
>charT,char_traits,Allocator>(lhs) == rhs.

template<class charT, class traits, class Allocator>
bool operator==(const basic_string<charT,traits,Allocator>& lhs,
const charT* rhs);

Returns: lhs == basic_string<charT,traits,Allocator>(rhs).

Remarks: Uses traits::length().

21.3.8.3 operator!=

template<class charT, class traits, class Allocator>
bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
const charT* rhs);

Returns: !(lhs == rhs).

template<class charT, class traits, class Allocator>
bool operator!=(const charT* lhs,
const basic_string<charT,traits,Allocator>& rhs);

Returns: lhs != basic_string<charT,traits,Allocator>(rhs).

Remarks: Uses traits::length().

21.3.8.4 operator<

template<class charT, class traits, class Allocator>
bool operator<(const basic_string<charT,traits,Allocator>& lhs,
const basic_string<charT,traits,Allocator>& rhs);

Returns: lhs.compare(rhs) < 0.

template<class charT, class traits, class Allocator>
bool operator<(const charT* lhs,
const basic_string<charT,traits,Allocator>& rhs);

Returns: basic_string<charT,traits,Allocator>(lhs) < rhs.

template<class charT, class traits, class Allocator>
bool operator<(const basic_string<charT,traits,Allocator>& lhs,
const charT* rhs);

Returns: lhs < basic_string<charT,traits,Allocator>(rhs).
21.3.8.5 `operator>`

```
template<class charT, class traits, class Allocator>
bool operator> (const basic_string<charT,traits,Allocator>& lhs, const basic_string<charT,traits,Allocator>& rhs);
```

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: `basic_string<charT,traits,Allocator>(lhs) > rhs`.

```
template<class charT, class traits, class Allocator>
bool operator> (const basic_string<charT,traits,Allocator>& lhs, const charT* rhs);
```

3. Returns: `lhs > basic_string<charT,traits,Allocator>(rhs)`.

21.3.8.6 `operator<=`

```
template<class charT, class traits, class Allocator>
bool operator<=(const basic_string<charT,traits,Allocator>& lhs, const basic_string<charT,traits,Allocator>& rhs);
```

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: `basic_string<charT,traits,Allocator>(lhs) <= rhs`.

```
template<class charT, class traits, class Allocator>
bool operator<=(const basic_string<charT,traits,Allocator>& lhs, const charT* rhs);
```

3. Returns: `lhs <= basic_string<charT,traits,Allocator>(rhs)`.

21.3.8.7 `operator>=`

```
template<class charT, class traits, class Allocator>
bool operator>=(const basic_string<charT,traits,Allocator>& lhs, const basic_string<charT,traits,Allocator>& rhs);
```

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);
```

Draft
2 template<class charT, class traits, class Allocator>
    bool operator>=(const basic_string<charT, traits, Allocator>& lhs,
                    const charT* rhs);

3 Returns: \( \text{lhs} \geq \text{basic_string}\langle\text{charT, traits, Allocator}\rangle\langle\text{rhs}\rangle \).

21.3.8.8 swap

template<class charT, class traits, class Allocator>
    void swap(basic_string<charT, traits, Allocator>& lhs,
              basic_string<charT, traits, Allocator>& rhs);

template<class charT, class traits, class Allocator>
    void swap(basic_string<charT, traits, Allocator>&& lhs,
              basic_string<charT, traits, Allocator>& rhs);

template<class charT, class traits, class Allocator>
    void swap(basic_string<charT, traits, Allocator>& lhs,
              basic_string<charT, traits, Allocator>&& rhs);

1 Effects: \( \text{lhs} . \text{swap} (\text{rhs}) \);

21.3.8.9 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);

1 Effects: Behaves as a formatted input function (27.6.1.2.1). After constructing a sentry object, if the sentry converts to true, calls \( \text{str} . \text{erase} () \) and then extracts characters from \( \text{is} \) and appends them to \( \text{str} \) as if by calling \( \text{str} . \text{append} (1, c) \). If \( \text{is} . \text{width} () \) is greater than zero, the maximum number \( n \) of characters appended is \( \text{is} . \text{width} () \); otherwise \( n \) is \( \text{str} . \text{max_size} () \). Characters are extracted and appended until any of the following occurs:

- \( n \) characters are stored;
- end-of-file occurs on the input sequence;
- \( \text{isspace}(c, \text{is} . \text{getloc} ()) \) is true for the next available input character \( c \).

2 After the last character (if any) is extracted, \( \text{is} . \text{width} (0) \) is called and the sentry object \( k \) is destroyed.

3 If the function extracts no characters, it calls \( \text{is} . \text{setstate} (\text{ios} : : \text{failbit}) \), which may throw \text{ios} \_\text{base} : : \text{-failure} (27.4.4.3).

4 Returns: \( \text{is} \)

template<class charT, class traits, class Allocator>
    basic_ostream<charT, traits>& operator<<((basic_ostream<charT, traits>&& os,
                          basic_string<charT, traits, Allocator>& str),


const basic_string<

Effects: Behaves as a formatted output function (27.6.2.6.1). After constructing a sentry object, if this object returns true when converted to a value of type bool, determines padding as described in 22.2.2.2, then inserts the resulting sequence of characters 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

Effects: Behaves as an unformatted input function (27.6.1.3), 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) (27.4.4.3)

— str.max_size() characters are stored (in which case, the function calls is.setstate(ios_base::failbit)) (27.4.4.3)

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 (27.4.4.3).

Returns: is.

Returns: getline(is, str, is.widen(\'\n\'))

21.4 Null-terminated sequence utilities

Tables 66, 67, 68, 69, 70, and 71 describe headers <cctype>, <cwctype>, <cstring>, <cwchar>, <cstdlib> (character conversions), and <cuchar>, respectively.
The contents of these headers shall be the same as the Standard C Library headers \(<\texttt{ctype.h}>\), \(<\texttt{wctype.h}>\), \(<\texttt{string.h}>\), \(<\texttt{wchar.h}>\), and \(<\texttt{stdlib.h}>\) and the C Unicode TR header \(<\texttt{uchar.h}>\), respectively, with the following modifications:

1. The headers shall not define the types \texttt{char16_t}, \texttt{char32_t}, and \texttt{wchar_t} (2.11).

2. The function signature \texttt{strchr\(\texttt{const char*} s, \texttt{int} c\)} shall be replaced by the two declarations:
   
   ```
   \texttt{const char* strchr(const char* s, int c);} \\
   \texttt{char* strchr(const char* s, int c);} 
   ```

   both of which shall have the same behavior as the original declaration.

3. The function signature \texttt{strpbrk\(\texttt{const char*} s1, \texttt{const char*} s2\)} shall be replaced by the two declarations:
   
   ```
   \texttt{const char* strpbrk(const char* s1, const char* s2);} \\
   \texttt{char* strpbrk(const char* s1, const char* s2);} 
   ```

   both of which shall have the same behavior as the original declaration.

4. The function signature \texttt{strrchr\(\texttt{const char*} s, \texttt{int} c\)} shall be replaced by the two declarations:
   
   ```
   \texttt{const char* strrchr(const char* s, int c);} \\
   \texttt{char* strrchr(const char* s, int c);} 
   ```

   both of which shall have the same behavior as the original declaration.

5. The function signature \texttt{strstr\(\texttt{const char*} s1, \texttt{const char*} s2\)} shall be replaced by the two declarations:
   
   ```
   \texttt{const char* strstr(const char* s1, const char* s2);} \\
   \texttt{char* strstr(const char* s1, const char* s2);} 
   ```

   both of which shall have the same behavior as the original declaration.

6. The function signature \texttt{memchr\(\texttt{const void*} s, \texttt{int} c, \texttt{size_t} n\)} shall be replaced by the two declarations:
   
   ```
   \texttt{const void* memchr(const void* s, int c, size_t n);} \\
   \texttt{void* memchr(const void* s, int c, size_t n);} 
   ```

   both of which shall have the same behavior as the original declaration.

7. The function signature \texttt{wcschr\(\texttt{const wchar_t*} s, \texttt{wchar_t} c\)} shall be replaced by the two declarations:
   
   ```
   \texttt{const wchar_t* wcschr(const wchar_t* s, wchar_t c);} \\
   \texttt{wchar_t* wcschr(const wchar_t* s, wchar_t c);} 
   ```

   both of which shall have the same behavior as the original declaration.

8. The function signature \texttt{wcspbrk\(\texttt{const wchar_t*} s1, \texttt{const wchar_t*} s2\)} shall be replaced by the two declarations:
   
   ```
   \texttt{const wchar_t* wcspbrk(const wchar_t* s1, const wchar_t* s2);} \\
   \texttt{wchar_t* wcspbrk(const wchar_t* s1, const wchar_t* s2);} 
   ```

   both of which shall have the same behavior as the original declaration.

9. The function signature \texttt{wcsrchr\(\texttt{const wchar_t*} s, \texttt{wchar_t} c\)} shall be replaced by the two declarations:
   
   ```
   \texttt{const wchar_t* wcsrchr(const wchar_t* s, wchar_t c);} \\
   \texttt{wchar_t* wcsrchr(const wchar_t* s, wchar_t c);} 
   ```

   both of which shall have the same behavior as the original declaration.

10. The function signature \texttt{wcstrchr\(\texttt{const wchar_t*} s, \texttt{wchar_t} c\)} shall be replaced by the two declarations:

    ```
    \texttt{const wchar_t* wcstrchr(const wchar_t* s, wchar_t c);} \\
    \texttt{wchar_t* wcstrchr(const wchar_t* s, wchar_t c);} 
    ```

    both of which shall have the same behavior as the original declaration.

11. The function signature \texttt{wcspbrk\(\texttt{const wchar_t*} s1, \texttt{wchar_t*} s2\)} shall be replaced by the two declarations:
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:

\[
\begin{align*}
\text{const wchar_t* wcsstr(const wchar_t* } & \text{s1, const wchar_t* s2);} \\
\text{wchar_t* wcsstr( wchar_t* } & \text{s1, const wchar_t* s2);} \\
\end{align*}
\]

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:

\[
\begin{align*}
\text{const wchar_t* wmemchr(const wchar_t* } & \text{s, wchar_t c, size_t n);} \\
\text{wchar_t* wmemchr( wchar_t* } & \text{s, wchar_t c, size_t n);} \\
\end{align*}
\]

both of which shall have the same behavior as the original declaration.

SEE ALSO: ISO C subclauses 7.3, 7.10.7, 7.10.8, and 7.11. Amendment 1 subclauses 4.4, 4.5, and 4.6.

### Table 66: Header `<cctype>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions:</td>
<td></td>
</tr>
<tr>
<td>isalnum</td>
<td>isblank isdigit</td>
</tr>
<tr>
<td>isalpha</td>
<td>isdigit</td>
</tr>
<tr>
<td>isprint</td>
<td>isprint isupper</td>
</tr>
<tr>
<td>isupper</td>
<td>isupper</td>
</tr>
<tr>
<td>tolower</td>
<td>isalpha</td>
</tr>
<tr>
<td>toupper</td>
<td>iscntrl islower</td>
</tr>
<tr>
<td>isspace</td>
<td>isspace</td>
</tr>
</tbody>
</table>

### Table 67: Header `<cwctype>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro:</td>
<td>WEOF <code>&lt;cwctype&gt;</code></td>
</tr>
<tr>
<td>Types:</td>
<td>wctrans_t wctype_t wint_t <code>&lt;cwctype&gt;</code></td>
</tr>
<tr>
<td>Functions:</td>
<td></td>
</tr>
<tr>
<td>iswalnum</td>
<td>iswcntrl iswgraph</td>
</tr>
<tr>
<td>iswalpha</td>
<td>iswpunct isxdigit</td>
</tr>
<tr>
<td>iswctype</td>
<td>iswlower iswspace</td>
</tr>
<tr>
<td>iswblank</td>
<td>iswdigit iswpunct</td>
</tr>
<tr>
<td></td>
<td>towctrans wctrans</td>
</tr>
<tr>
<td></td>
<td>towupper towlower wctype</td>
</tr>
</tbody>
</table>
### 21.4 Null-terminated sequence utilities

#### Table 68: 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><code>size_t </code>&lt;cstring&gt;`</td>
</tr>
<tr>
<td><strong>Functions:</strong></td>
<td><code>memchr</code> <code>strcat</code> <code>strcspn</code> <code>strncpy</code> <code>strtok</code> <code>memcmp</code> <code>strchr</code> <code>strerror</code> <code>strpbrk</code> <code>strxfrm</code> <code>memcpy</code> <code>strcmp</code> <code>strlen</code> <code>strrchr</code> <code>memmove</code> <code>strcoll</code> <code>strncat</code> <code>strspn</code> <code>memset</code> <code>strcpy</code> <code>strncmp</code> <code>strstr</code></td>
</tr>
</tbody>
</table>

#### Table 69: Header `<cwchar>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Macros:</strong></td>
<td>NULL <code>&lt;cwchar&gt;</code> <code>WCHAR_MAX</code> <code>WCHAR_MIN</code> <code>WEOF </code>&lt;cwchar&gt;`</td>
</tr>
<tr>
<td><strong>Types:</strong></td>
<td><code>mbstate_t</code> <code>wint_t </code>&lt;cwchar&gt;<code> </code>size_t<code> </code>tm`</td>
</tr>
<tr>
<td><strong>Functions:</strong></td>
<td><code>btowc</code> <code>mbrlen</code> <code>vfscanf</code> <code>wscanf</code> <code>wcsply</code> <code>wcsspn</code> <code>wcsxfrm</code> <code>fgetwc</code> <code>mbrtowc</code> <code>vswscanf</code> <code>wcscspn</code> <code>wcsstr</code> <code>wctob</code> <code>fgetws</code> <code>mbsinit</code> <code>vswprintf</code> <code>wcsncat</code> <code>wcsstr</code> <code>wctob</code> <code>fwide</code> <code>putc</code> <code>vfwprintf</code> <code>wcsftime</code> <code>wcstol</code> <code>wmemb</code> <code>fwrite</code> <code>putwc</code> <code>vfprintf</code> <code>wcscpy</code> <code>wcssp</code> <code>wcspnr</code> <code>wcxtbm</code> <code>fwrite</code> <code>putwchar</code> <code>vfwprintf</code> <code>wcsftime</code> <code>wcstod</code> <code>wmemchr</code> <code>fwrite</code> <code>vfwprintf</code> <code>wcscat</code> <code>wcsncpy</code> <code>wcstol</code> <code>wmemchr</code> <code>getwc</code> <code>ungetc</code> <code>vfwidth</code> <code>wcscmp</code> <code>wcsrchr</code> <code>wcstoul</code> <code>wscanf</code> <code>getwchar</code> <code>vfwidth</code> <code>wscoll</code> <code>wcsrtombs</code> <code>wcstoull</code></td>
</tr>
</tbody>
</table>

#### Table 70: Header `<cstdlib>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Macros:</strong></td>
<td>MB_CUR_MAX</td>
</tr>
<tr>
<td><strong>Functions:</strong></td>
<td><code>atof</code> <code>mblen</code> <code>strtol</code> <code>strtoull</code> <code>atoi</code> <code>mbtowc</code> <code>strtold</code> <code>wcstomb</code> <code>atol</code> <code>mbstowcs</code> <code>strtoll</code> <code>wcstombs</code> <code>atoll</code> <code>strtod</code> <code>strtoull</code></td>
</tr>
</tbody>
</table>

#### Table 71: Header `<cuchar>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Macros:</strong></td>
<td><strong>STDC_UTF_16</strong> <strong>STDC_UTF_32</strong></td>
</tr>
<tr>
<td><strong>Functions:</strong></td>
<td><code>mbtouc</code> <code>c16rtomb</code> <code>mbtoc32</code> <code>c32rtomb</code></td>
</tr>
</tbody>
</table>

Draft
Draft
Chapter 22  Localization library

This clause describes components that C++ programs may use to encapsulate (and therefore be more portable when confronting) cultural differences. The locale facility includes internationalization support for character classification and string collation, numeric, monetary, and date/time formatting and parsing, and message retrieval.

The following subclauses describe components for locales themselves, the standard facets, and facilities from the ISO C library, as summarized in Table 72.

Table 72: Localization library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.1 Locales</td>
<td>&lt;locale&gt;</td>
</tr>
<tr>
<td>22.2 Standard locale Categories</td>
<td></td>
</tr>
<tr>
<td>22.3 C library locales</td>
<td>&lt;locale&gt;</td>
</tr>
</tbody>
</table>

22.1 Locales

Header <locale> synopsis

namespace std {
    // 22.1.1, locale:
    class locale;
    template <class Facet> const Facet& use_facet(const locale&);
    template <class Facet> bool has_facet(const locale&) throw();

    // 22.1.3, 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 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);
template <class Codecvt, class Elem = wchar_t> class wstring_convert;
    template <class Codecvt, class Elem = wchar_t,
       class Tr = std::char_traits<Elem>> class wbuffer_convert;

// 22.2.1 and 22.2.1.3, ctype:
    class ctype_base;
    template <class charT> class ctype;
    template <> class ctype<char>;
    // specialization
    template <class charT> class ctype_byname;
    template <> class ctype_byname<char>;
    // specialization
    class codecvt_base;
    template <class internT, class externT, class stateT>
    class codecvt;
    template <class internT, class externT, class stateT>
    class codecvt_byname;

// 22.2.2 and 22.2.3, numeric:
    template <class charT, class InputIterator> class num_get;
    template <class charT, class OutputIterator> class num_put;
    template <class charT> class numpunct;
    template <class charT> class numpunct_byname;

// 22.2.4, collation:
    template <class charT> class collate;
    template <class charT> class collate_byname;

// 22.2.5, date and time:
    class time_base;
    template <class charT, class InputIterator> class time_get;
    template <class charT, class InputIterator> class time_get_byname;
    template <class charT, class OutputIterator> class time_put;
    template <class charT, class OutputIterator> class time_put_byname;

// 22.2.6, money:
    class money_base;
    template <class charT, class InputIterator> class money_get;
    template <class charT, class OutputIterator> class money_put;
    template <class charT, bool Intl> class moneypunct;
    template <class charT, bool Intl> class moneypunct_byname;

// 22.2.7, message retrieval:
    class messages_base;
    template <class charT> class messages;
    template <class charT> class messages_byname;
}

The header `<locale>` defines classes and declares functions that encapsulate and manipulate the information peculiar
to a locale.\textsuperscript{230) 22.1 Class locale

namespace std {
    class locale {
        public:
            // types:
            class facet;
            class id;
            typedef int category;  // values assigned here are for exposition only
            static const category none = 0,
                collate = 0x010, ctype = 0x020,
                monetary = 0x040, numeric = 0x080,
                time = 0x100, messages = 0x200,
                all = collate | ctype | monetary | numeric | time | messages;

            // construct/copy/destroy:
            locale() throw();
            locale(const locale& other) throw();
            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() throw();  // non-virtual
            const locale& operator=(const locale& other) throw();
            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();
        }
    }
}

\textsuperscript{230) In this subclause, the type name struct tm is an incomplete type that is defined in <ctime>.

Class locale implements a type-safe polymorphic set of facets, indexed by facet type. In other words, a facet has a dual role: in one sense, it’s just a class interface; at the same time, it’s an index into a locale’s set of facets.
Access to the facets of a locale is via two function templates, `use_facet<>` and `has_facet<>`.

[Example: An iostream operator<< might be implemented as:]

```cpp
template <class charT, class traits>
basic_ostream<charT,traits>&
operator<< (basic_ostream<charT,traits>& s, Date d)
{
    typename basic_ostream<charT,traits>::sentry cerberos(s);
    if (cerberos) {
        ios_base::iostate err = 0;
        tm tmbuf; d.extract(tmbuf);
        use_facet< time_put<charT,ostreambuf_iterator<charT,traits> > > (s.getloc()).put(s, s, s.fill(), err, &tmbuf, 'x');
        s.setstate(err);
        // might throw
    }
    return s;
}
```

— end example ]

In the call to `use_facet<Facet>(loc)`, the type argument chooses a facet, making available all members of the named type. If `Facet` is not present in a locale, it throws the standard exception `bad_cast`. A C++ program can check if a locale implements a particular facet with the function template `has_facet<Facet>()`. User-defined facets may be installed in a locale, and used identically as may standard facets (22.2.8).

[Note: All locale semantics are accessed via `use_facet<>` and `has_facet<>`, except that:

— A member operator template `operator()(basic_string<C,T,A>&, 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.

— 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 (27.6.1.2)). — end note ]

Once a facet reference is obtained from a locale object by calling `use_facet<>`, that reference remains usable, and the results from member functions of it may be cached and re-used, as long as some locale object refers to that facet.

In successive calls to a locale facet member function on a facet object installed in the same locale, the returned result shall be identical.

A locale constructed from a name string (such as "POSIX"), or from parts of two named locales, has a name; all others do not. Named locales may be compared for equality; an unnamed locale is equal only to (copies of) itself. For an unnamed locale, `locale::name()` returns the string "*".

### 22.1.1 locale types

#### 22.1.1.1 Type locale::category

```cpp
typedef int category;
```

231) Notice that, in the call to put, the stream is implicitly converted to an `ostreambuf_iterator<charT,traits>`. Draft
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

\[(\text{collate} \mid \text{ctype} \mid \text{monetary} \mid \text{numeric} \mid \text{time} \mid \text{messages} \mid \text{all}) == \text{all}\]

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.

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 73:

<table>
<thead>
<tr>
<th>Category</th>
<th>Includes Facets</th>
</tr>
</thead>
<tbody>
<tr>
<td>collate</td>
<td><code>collate&lt;char&gt;</code>, <code>collate&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td>ctype</td>
<td><code>ctype&lt;char&gt;</code>, <code>ctype&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>codecvt&lt;char,char,mbstate_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>codecvt&lt;char16_t,char,mbstate_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>codecvt&lt;char32_t,char,mbstate_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>codecvt&lt;wchar_t,char,mbstate_t&gt;</code></td>
</tr>
<tr>
<td>monetary</td>
<td><code>moneypunct&lt;char&gt;</code>, <code>moneypunct&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>moneypunct&lt;char,true&gt;</code>, <code>moneypunct&lt;wchar_t,true&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>money_get&lt;char&gt;</code>, <code>money_get&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>money_put&lt;char&gt;</code>, <code>money_put&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td>numeric</td>
<td><code>numpunct&lt;char&gt;</code>, <code>numpunct&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>num_get&lt;char&gt;</code>, <code>num_get&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>num_put&lt;char&gt;</code>, <code>num_put&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td>time</td>
<td><code>time_get&lt;char&gt;</code>, <code>time_get&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>time_put&lt;char&gt;</code>, <code>time_put&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td>messages</td>
<td><code>messages&lt;char&gt;</code>, <code>messages&lt;wchar_t&gt;</code></td>
</tr>
</tbody>
</table>

For any locale `loc` either constructed, or returned by `locale::classic()`, and any facet `Facet` shown in Table 73, `has_facet<Facet>(loc)` is true. Each locale member function which takes a `locale::category` argument operates on the corresponding set of facets.

An implementation is required to provide those specializations for facet templates identified as members of a category, and for those shown in Table 74:

The provided implementation of members of facets `num_get<charT>` and `num_put<charT>` calls `use_facet<F>(1)` only for facet \(F\) of types `numpunct<charT>` and `ctype<charT>`, and for locale \(l\) the value obtained by calling member `getloc()` on the `ios_base&` argument to these functions.

In declarations of facets, a template formal 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 (24.1). A template formal parameter with name `C` represents the set of all possible specializations on a
<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;</td>
</tr>
<tr>
<td></td>
<td>numpunct_byname&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>num_get&lt;C, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>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>
A template formal parameter with name International represents the set of all possible specializations on a bool parameter.

22.1.1.1.2 Class locale::facet

```cpp
namespace std {
    class locale::facet {
        protected:
            explicit facet(size_t refs = 0);
            virtual ~facet();

        private:
            facet(const facet&) = delete;  // not defined
            void operator=(const facet&) = delete;  // not defined
    };
}
```

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.

   — For refs == 0, the implementation performs delete static_cast<locale::facet*>(f) (where f is a pointer to the facet) when the last locale object containing the facet is destroyed; for refs == 1, the implementation never destroys the facet.

3 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.

4 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.

22.1.1.3 Class locale::id

```cpp
namespace std {
    class locale::id {
        public:
            id();

        private:
            void operator=(const id&) = delete;  // not defined
    };
}
```
The class locale::id provides identification of a locale facet interface, used as an index for lookup and to encapsulate initialization.

[Note: Because facets are used by iostreams, potentially while static constructors are running, their initialization cannot depend on programmed static initialization. One initialization strategy is for locale to initialize each facet’s id member the first time an instance of the facet is installed into a locale. This depends only on static storage being zero before constructors run (3.6.2). — end note]

### 22.1.1.2 locale constructors and destructor

#### locale() throw();

Default constructor: a snapshot of the current global locale.

**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]

#### locale(const locale& other) throw();

**Effects:** Constructs a locale which is a copy of `other`.

#### const locale& operator=(const locale& other) throw();

**Effects:** Creates a copy of `other`, replacing the current value.

**Returns:** `*this`

#### explicit locale(const char* std_name);

**Effects:** Constructs a locale using standard C locale names, e.g. "POSIX". The resulting locale implements semantics defined to be associated with that name.

**Throws:** runtime_error if the argument is not valid, or is null.

**Remarks:** The set of valid string argument values is "C", "", and any implementation-defined values.

#### explicit locale(const string& std_name);

**Effects:** The same as `locale(std_name.c_str())`.

#### locale(const locale& other, const char* std_name, category);

**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)`.

**Throws:** runtime_error if the argument is not valid, or is null.

**Remarks:** The locale has a name if and only if `other` has a name.
locale(const locale& other, const string& std_name, category cat);

*Effects:* The same as locale(other, std_name.c_str(), cat).

```cpp
template <class Facet> locale(const locale& other, Facet* f);
```

*Effects:* Constructs a locale incorporating all facets from the first argument except that of type Facet, and installs the second argument as the remaining facet. If `f` is null, the resulting object is a copy of `other`.

*Remarks:* The resulting locale has no name.

locale(const locale& other, const locale& one, category cats);

*Effects:* Constructs a locale incorporating all facets from the first argument except those that implement `cats`, which are instead incorporated from the second argument.

*Remarks:* The resulting locale has a name if and only if the first two arguments have names.

```
~locale() throw();
```

A non-virtual destructor that throws no exceptions.

### 22.1.1.3 locale members

```cpp
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
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.

### 22.1.1.4 locale operators

```cpp
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.

```cpp
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,
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:

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 (25.3.1, 23.2.5):
    std::sort(v.begin(), v.end(), loc);
—end example]

22.1.1.5 locale static members

static locale global(const locale& loc);
Sets the global locale to its argument.
Effects: Causes future calls to the constructor locale() to return a copy of the argument. If the argument has a name, does
    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().
Returns: The previous value of locale().

static const locale& classic();
The "C" locale.
Returns: A locale that implements the classic "C" locale semantics, equivalent to the value locale("C").
Remarks: This locale, its facets, and their member functions, do not change with time.

22.1.2 locale globals

template <class Facet> const Facet& use_facet(const locale& loc);
Requires: Facet is a facet class whose definition contains the public static member id as defined in 22.1.1.1.2.
Returns: a reference to the corresponding facet of loc, if present.
599 Localization library

22.1 Locales

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) throw();

 Returns: true if the facet requested is present in loc; otherwise false.

22.1.3 Convenience interfaces [locale.convenience]

22.1.3.1 Character classification [classification]

template <class charT> bool isspace (charT c, const locale& loc);
template <class charT> bool isprint (charT c, const locale& loc);
template <class charT> bool iscntrl (charT c, const locale& loc);
template <class charT> bool isupper (charT c, const locale& loc);
template <class charT> bool islower (charT c, const locale& loc);
template <class charT> bool isalpha (charT c, const locale& loc);
template <class charT> bool isdigit (charT c, const locale& loc);
template <class charT> bool ispunct (charT c, const locale& loc);
template <class charT> bool isxdigit(charT c, const locale& loc);
template <class charT> bool isalnum (charT c, const locale& loc);
template <class charT> bool isgraph (charT c, const locale& loc);

1 Each of these functions isF returns the result of the expression:

   use_facet<ctype<charT>>(loc).is(ctype_base::F, c)

   where F is the ctype_base::mask value corresponding to that function (22.2.1). \(^{232}\)

22.1.3.2 Conversions [conversions]

22.1.3.2.1 Character conversions [conversions.character]

template <class charT> charT toupper(charT c, const locale& loc);

1 Returns: use_facet<ctype<charT>>(loc).toupper(c).

template <class charT> charT tolower(charT c, const locale& loc);

2 Returns: use_facet<ctype<charT>>(loc).tolower(c).

22.1.3.2.2 string conversions [conversions.string]

Class template wstring_convert performs conversions between a wide string and a byte string. It lets you specify a code conversion facet (like class template codecvt) to perform the conversions, without affecting any streams or locales. [Example: Say, for example, you have a code conversion facet called codecvt_utf8 that you want to use 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:

\(^{232}\) 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.
22.1 Locales

wstring_convert<codecvt_utf8<wchar_t>> myconv;
std::string mbstring = myconv.to_bytes(L"Hello\n");
std::cout << mbstring;

— end example

Class template wstring_convert synopsis

namespace std {
    template<class Codecvt,
             class Elem = wchar_t>
    class wstring_convert
    {
        typedef std::basic_string<char> byte_string;
        typedef std::basic_string<Elem> wide_string;
        typedef typename Codecvt::state_type state_type;
        typedef typename wide_string::traits_type::int_type int_type;

        wstring_convert(Codecvt *pcvt = new Codecvt);
        wstring_convert(Codecvt *pcvt, state_type state);
        wstring_convert(const byte_string& byte_err,
                        const wide_string& wide_err = wide_string());
        ~wstring_convert();

        wide_string from_bytes(char byte);
        wide_string from_bytes(const char *ptr);
        wide_string from_bytes(const byte_string& str);
        wide_string from_bytes(const char *first, const char *last);

        byte_string to_bytes(Elem wchar);
        byte_string to_bytes(const _Elem *wptr);
        byte_string to_bytes(const wide_string& wstr);
        byte_string to_bytes(const Elem *first, const Elem *last);

        size_t converted() const;
        state_type state() const;
    
    private:
        byte_string byte_err_string;       // exposition only
        wide_string wide_err_string;       // exposition only
        Codecvt *cvtptr;                   // exposition only
        state_type cvtstate;               // exposition only
        size_t cvtcount;                   // exposition only
    
    
}

The class template describes an object that controls conversions between wide string objects of class std::basic_string<Elem>
and byte string objects of class std::basic_string<char> (also known as std::string). 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 ob-
ject of class Codecvt<Elem, char, std::mbstate_t>, which meets the requirements of the standard code-conversion
facet std::codecvt<Elem, char, std::mbstate_t>.

4 An object of this class template stores:
   — byte_err_string — a byte string to display on errors
   — wide_err_string — a wide string to display on errors
   — cvtptr — a pointer to the allocated conversion object (which is freed when the wstring_convert object is
     destroyed)
   — cvtstate — a conversion state object
   — cvtcount — a conversion count

typedef std::basic_string<char> byte_string;

5 The type shall be a synonym for std::basic_string<char>

size_t converted() const;

6 Returns: cvtcount.

wide_string from_bytes(char byte);
wide_string from_bytes(const char *ptr);
wide_string from_bytes(const byte_string& str);
wide_string from_bytes(const char *first, const char *last);

7 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.

8 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.

9 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;

10 returns cvtstate.

typedef typename Codecvt::state_type state_type;
22.1  Locales  

The type shall be a synonym for Codecvt::state_type.

```cpp
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 nul-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.

```cpp
typedef std::basic_string<Elem> wide_string;
```

The type shall be a synonym for std::basic_string<Elem>.

```cpp
wstring_convert(Codecvt *pcvt = new Codecvt);
wstring_convert(Codecvt *pcvt, state_type state);  
const byte_string& byte_err, 
    const wide_string& wide_err = wide_string());
```

**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.

```cpp
~wstring_convert();
```

**Effects:** The destructor shall delete cvtptr.

### 22.1.3.2.3 Buffer conversions  

[conversions.buffer]

**Class template** `wbuffer_convert` looks like a wide stream buffer, but performs all its I/O through an underlying byte stream buffer that you specify when you construct it. Like class template `wstring_convert`, it lets you specify a code conversion facet to perform the conversions, without affecting any streams or locales.

**Class template** `wbuffer_convert` synopsis

```cpp
namespace std {
    template<class Codecvt,
```
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<Elem, char, std::mbstate_t>`, 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
- `cvt.ptr` — a pointer to the allocated conversion object (which is freed when the `wbuffer_convert` object is destroyed)
- `cvtstate` — a conversion state object

`state_type state() const;`  
**Returns:** `cvtstate`.

`std::streambuf *rdbuf() const;`  
**Returns:** `bufptr`.

`std::streambuf *rdbuf(std::streambuf *bytebuf);`  
**Effects:** stores `bytebuf` in `bufptr`.

**Returns:** the previous value of `bufptr`.  

Draft
typedef typename Codecvt::state_type state_type;

The type shall be a synonym for Codecvt::state_type.

wbuffer_convert(std::streambuf *bytebuf = 0,
                Codecvt *pcvt = new Codecvt, state_type state = state_type());

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.

22.2 Standard locale categories

22.2.1 The ctype category

namespace std {
    class ctype_base {
        public:
            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 alnum = alpha | digit;
            static const mask graph = alnum | punct;
    };
}
The type `mask` is a bitmask type (17.3.2.1.2).

### 22.2.1.1 Class template ctype

```cpp
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 toupper(charT c) const;
    const charT* toupper(charT* low, const charT* high) const;
    charT tolower(charT c) const;
    const charT* tolower(charT* low, const charT* high) const;
    charT widen(char c) const;
    const char* widen(const char* low, const char* high, charT* to) const;
    char narrow(charT c, char dfault) const;
    const charT* narrow(const charT* low, const charT*, char dfault, char* to) const;

    static locale::id id;

    protected:
    ~ctype();

    virtual bool do_is(mask m, charT c) const;
    virtual const charT* do_is(const charT* low, const charT* high, mask* vec) const;
    virtual const charT* do_scan_is(mask m,
                                    const charT* low, const charT* high) const;
    virtual const charT* do_scan_not(mask m,
                                       const charT* low, const charT* high) const;
    virtual charT do_toupper(charT) const;
    virtual const charT* do_toupper(charT* low, const charT* high) const;
    virtual charT do_tolower(charT) const;
    virtual const charT* do_tolower(charT* low, const charT* high) const;
    virtual charT do_widen(char) const;
    virtual const char* do_widen(const char* low, const char* high, charT* dest) const;
    virtual char do_narrow(charT, char dfault) const;
    virtual const charT* do_narrow(const charT* low, const charT* high, char dfault, char* dest) const;
};
```

Draft
22.2 Standard locale categories

Class ctype encapsulates the C library <ctype> features. istream members are required to use \texttt{ctype\tag*{}} for character classing during input parsing.

The specializations required in Table 73 (22.1.1.1.1), namely \texttt{ctype\tag*{char}}</cctype> and \texttt{ctype\tag*{wchar_t}}, implement character classing appropriate to the implementation’s native character set.

22.2.1.1 ctype members

\begin{verbatim}
bool is(mask m, charT c) const;
const charT* is(const charT* low, const charT* high, mask* vec) const;
\end{verbatim}

1 \textit{Returns}: do\_is\tag*{(m, c)} or do\_is\tag*{(low, high, vec)}

\begin{verbatim}
const charT* scan_is(mask m, const charT* low, const charT* high) const;
\end{verbatim}

2 \textit{Returns}: do\_scan\_is\tag*{(m, low, high)}

\begin{verbatim}
const charT* scan_not(mask m, const charT* low, const charT* high) const;
\end{verbatim}

3 \textit{Returns}: do\_scan\_not\tag*{(m, low, high)}

\begin{verbatim}
charT toupper(charT c) const;
const charT* toupper(charT* low, const charT* high) const;
\end{verbatim}

4 \textit{Returns}: do\_toupper\tag*{(c)} or do\_toupper\tag*{(low, high)}

\begin{verbatim}
charT toler(charT c) const;
const charT* toler(charT* low, const charT* high) const;
\end{verbatim}

5 \textit{Returns}: do\_tolower\tag*{(c)} or do\_tolower\tag*{(low, high)}

\begin{verbatim}
charT widen(char c) const;
const char* widen(const char* low, const char* high, char* to) const;
\end{verbatim}

6 \textit{Returns}: do\_widen\tag*{(c)} or do\_widen\tag*{(low, high, to)}

\begin{verbatim}
narrow(charT c, char default) const;
const charT* narrow(const charT* low, const charT*, char default, char* to) const;
\end{verbatim}

7 \textit{Returns}: do\_narrow\tag*{(c, default)} or do\_narrow\tag*{(low, high, default, to)}

22.2.1.2 ctype virtual functions

\begin{verbatim}
bool do_is(mask m, charT c) const;
const charT* do_is(const charT* low, const charT* high, mask* vec) const;
\end{verbatim}
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 $\ast p$ where $(\text{low} \leq p < \text{high})$, and places it into $\text{vec}[p-\text{low}]$.

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`.

```cpp
def do_scan_is(mask m, const charT* low, const charT* high) const;
```

Effects: Locates a character in a buffer that conforms to a classification $m$.

Returns: The smallest pointer $p$ in the range $[\text{low}, \text{high})$ such that $\text{is}(m, \ast p)$ would return `true`; otherwise, returns `high`.

```cpp
def 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 $[\text{low}, \text{high})$ such that $\text{is}(m, \ast p)$ would return `false`; otherwise, returns `high`.

```cpp
charT do_toupper(charT c) const;
def do_toupper(charT* low, const charT* high) const;
```

Effects: Converts a character or characters to upper case. The second form replaces each character $\ast p$ in the range $[\text{low}, \text{high})$ 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`.

```cpp
charT do_tolower(charT c) const;
def do_tolower(charT* low, const charT* high) const;
```

Effects: Converts a character or characters to lower case. The second form replaces each character $\ast p$ in the range $[\text{low}, \text{high})$ 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`.

```cpp
charT do_widen(char c) const;
def do_widen(const char* low, const char* high, charT* dest) const;
```

Effects: Applies the simplest reasonable transformation from a char value or sequence of char values to the corresponding charT value or values. The only characters for which unique transformations are required are those in the basic source character set (2.2).

For any named ctype category with a `ctype<charT>` facet `ctc` and valid `ctype_base::mask` value $M$, $(\text{ctc}. \text{is}(M, c) \lor \text{is}(M, \text{do_widen}(c))$ is true.

---

233) The char argument of `do_widen` is intended to accept values derived from character literals for conversion to the locale’s encoding.

234) In other words, the transformed character is not a member of any character classification that $c$ is not also a member of.
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 \( high \).

```cpp
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(2.2) the transformation is such that

\[
\text{do_widen(do_narrow(c,0)) == c}
\]

For any named \( ctype \) category with a \( ctype<\text{char}> \) facet \( ctc \) however, and \( \text{ctype_base::mask} \) value \( M \),

\[
(\text{is}(M, c) || !\text{ctc}.is(M, \text{do_narrow}(c, dfault))) \]

is true (unless \( \text{do_narrow} \) returns \( dfault \)). In addition, for any digit character \( c \), the expression \( (\text{do_narrow}(c, dfault) - '0') \) evaluates to the digit value of the character. The second form transforms each character \( *p \) in the range \([low, high)\), placing the result (or \( dfault \) if no simple transformation is readily available) in \( dest[p-low] \).

Returns: The first form returns the transformed value; or \( dfault \) if no mapping is readily available. The second form returns \( high \).

### 22.2.1.2 Class template \texttt{ctype\_byname}

```cpp
namespace std {
    template <class charT>
    class ctype\_byname : public ctype<\texttt{charT}> {
    public:
        typedef typename ctype<\texttt{charT}>::mask mask;
        explicit ctype\_byname(const char*, size_t refs = 0);
        explicit ctype\_byname(const string&, size_t refs = 0);
    protected:
        ~ctype\_byname(); // virtual
    };
}
```

### 22.2.1.3 \texttt{ctype} specializations

```cpp
namespace std {
    template <> class ctype<\texttt{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) const;
char narrow(char c, char default) const;
const char* narrow(const char* low, const char* high, char default,
    char* to) const;

static locale::id id;
static const size_t table_size = IMPLEMENTATION_DEFINED;

protected:
const mask* table() const throw();
static const mask* classic_table() throw();

`ctype(); // virtual

virtual char do_toupper(char c) const;
virtual const char* do_toupper(char* low, const char* high) const;
virtual char do_tolower(char c) const;
virtual const char* do_tolower(char* low, const char* high) const;

virtual char do_widen(char c) const;
virtual const char* do_widen(const char* low,
    const char* high,
    char* to) const;
virtual char do_narrow(char c, char default) const;
virtual const char* do_narrow(const char* low,
    const char* high,
    char default, char* to) const;

};

A specialization ctype<char> is provided so that the member functions on type char can be implemented inline.\(^{235}\)
The implementation-defined value of member table_size is at least 256.

22.2.1.3.1 ctype<char> destructor [facet.ctype.char.dtor]

\(^{235}\) 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>.

Draft
```
"ctype();
Effects: If the constructor's first argument was nonzero, and its second argument was true, does delete [] table().

22.2.1.3.2 ctype<char> members
In the following member descriptions, for unsigned char values v where (v >= table_size), table()[v] is assumed to have an implementation-defined 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);
Precondition: tbl either 0 or an array of at least table_size elements.
Effects: Passes its refs argument to its base class constructor.

bool is(mask m, char c) const;
const char* is(const char* low, const char* high, mask* vec) const;
Effects: The second form, for all *p in the range [low,high), assigns into vec[p-low] the value table()[(unsigned char)*p].
Returns: The first form returns table()[(unsigned char)c] & m; the second form returns high.

const char* scan_is(mask m, const char* low, const char* high) const;
Returns: The smallest p in the range [low,high) such that
table()[(unsigned char) *p] & m
is true.

const char* scan_not(mask m, const char* low, const char* high) const;
Returns: The smallest p in the range [low,high) such that
table()[(unsigned char) *p] & m
is false.

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.
```
Localization library

22.2 Standard locale categories

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 default) const;
const char* narrow(const char* low, const char* high, char default, char* to) const;

Returns: do_narrow(c, default) or do_narrow(low, high, default, to), respectively.

const mask* table() const throw();

Returns: The first constructor argument, if it was non-zero, otherwise classic_table().

22.2.1.3.3 ctype<char> static members

static const mask* classic_table() throw();

Returns: A pointer to the initial element of an array of size table_size which represents the classifications of characters in the "C" locale.

22.2.1.3.4 ctype<char> 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 default) const;
virtual const char* do_narrow(const char* low, const char* high, char default, char* to) const;

These functions are described identically as those members of the same name in the ctype class template (22.2.1.1).

22.2.1.4 Class template codecvt

namespace std {
    class codecvt_base {
    public:
        enum result { ok, partial, error, noconv };
    };
    template <class internT, class externT, class stateT>
    class codecvt : public locale::facet, public codecvt_base {

    Draft
The class \codev\text{<intern_T, extern_T, state_T>} is for use when converting from one codeset to another, such as from wide characters to multibyte characters or between wide character encodings such as Unicode and EUC.

The \text{state_T} argument selects the pair of codesets being mapped between.

The specializations required in Table 73 (22.1.1.1.1) convert the implementation-defined native character set. \codev\text{<char, char, mbstate_t>} implements a degenerate conversion; it does not convert at all. The specialization \codev\text{<char16_t, char, mbstate_t>} converts between the UTF-16 and UTF-8 encodings schemes, and the specialization \codev\text{<char32_t, char, mbstate_t>} converts between the UTF-32 and UTF-8 encodings schemes.
Localization library 22.2 Standard locale categories

t<char,mbstate_t> converts between the native character sets for narrow and wide characters. Specializations on mbstate_t perform conversion between encodings known to the library implementor. Other encodings can be converted by specializing on a user-defined stateT type. The stateT object can contain any state that is useful to communicate to or from the specialized do_in or do_out members.

22.2.1.4.1 codecvt members

result out(stateT& state,
const internT* from, const internT* from_end, const internT*& from_next,
externT* to, externT* to_limit, externT*& to_next) const;

1 Returns: do_out(state, from, from_end, from_next, to, to_limit, to_next)

result unshift(stateT& state,
externT* to, externT* to_limit, externT*& to_next) const;

2 Returns: do_unshift(state, to, to_limit, to_next)

result in(stateT& state,
const externT* from, const externT* from_end, const externT*& from_next,
internT* to, internT* to_limit, internT*& to_next) const;

3 Returns: do_in(state, from, from_end, from_next, to, to_limit, to_next)

int encoding() const throw();

4 Returns: do_encoding()

bool always_noconv() const throw();

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 throw();

7 Returns: do_max_length()

22.2.1.4.2 codecvt virtual functions

result do_out(stateT& state,
const internT* from, const internT* from_end, const internT*& from_next,
externT* to, externT* to_limit, externT*& to_next) const;

result do_in(stateT& state,
const externT* from, const externT* from_end, const externT*& from_next,
internT* to, internT* to_limit, internT*& to_next) const;

1 Preconditions: (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.

Draft
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 \((to\_limit - to)\) destination elements.

Stops if it encounters a character it cannot convert. It always leaves the \(from\_next\) and \(to\_next\) pointers pointing one beyond the last element successfully converted. If returns \(\text{noconv}\), \(\text{internT}\) and \(\text{externT}\) 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 \([to, to\_limit)\).

A \codecvt facet that is used by basic_filebuf (27.8) shall have the property that if

\[
\text{do\_out}(state, from, from\_end, from\_next, to, to\_limit, to\_next)
\]

would return \(\text{ok}\), where \(from \neq from\_end\), then

\[
\text{do\_out}(state, from, from + 1, from\_next, to, to\_end, to\_next)
\]

shall also return \(\text{ok}\), and that if

\[
\text{do\_in}(state, from, from\_end, from\_next, to, to\_limit, to\_next)
\]

would return \(\text{ok}\), where \(to \neq to\_end\), then

\[
\text{do\_in}(state, from, from\_end, from\_next, to, to + 1, to\_next)
\]

shall also return \(\text{ok}\).

Remarks: Its operations on \(state\) are unspecified. \([\text{Note:} \text{This argument can be used, for example, to maintain shift state, to specify conversion options (such as count only), or to identify a cache of seek offsets.} \text{-- end note}]\)

Returns: An enumeration value, as summarized in Table 75:

<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>(\text{internT}) and (\text{externT}) are the same type, and input sequence is identical to converted sequence</td>
</tr>
</tbody>
</table>

A return value of \(\text{partial}\), 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.

\[
\text{result do\_unshift(stateT\& state,}
\text{ externT* to, externT* to\_limit, externT*\& to\_next) const;}
\]

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.
6 **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.

7 **Effects:** Places characters starting at \(to\) that should be appended to terminate a sequence when the current \(\text{state}\) is given by \(\text{state}\). \(^{237}\) Stores no more than \((\text{to\_limit}-\text{to})\) destination elements, and leaves the \(\text{to\_next}\) pointer pointing one beyond the last element successfully stored. \(\text{codecvt}<\text{char, char, mbstate\_t}>\) stores no characters.

8 **Returns:** An enumeration value, as summarized in Table 76:

<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 ((\text{to_limit}-\text{to})) destination elements was needed to terminate a sequence given the value of (\text{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 (\text{state_type})</td>
</tr>
</tbody>
</table>

\(\text{codecvt}<\text{char, char, mbstate\_t}>\), returns \(\text{noconv}\).

9 \(\text{int do\_encoding()}\) const throw();

| Returns: | -1 if the encoding of the \(\text{extern\_T}\) sequence is state-dependent; else the constant number of \(\text{extern\_T}\) characters needed to produce an internal character; or 0 if this number is not a constant\(^{238}\). |

10 \(\text{bool do\_always\_noconv()}\) const throw();

| Returns: | true if \(\text{do\_in()}\) and \(\text{do\_out()}\) return \(\text{noconv}\) for all valid argument values. \(\text{codecvt}<\text{char, char, mbstate\_t}>\) returns true. |

11 \(\text{int do\_length(\text{state\_t} state, const \text{extern\_T* from, const extern\_T* from\_end, size\_t max})}\) const;

| Preconditions: | \((\text{from}<=\text{from\_end})\) 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. |

12 **Effects:** The effect on the \(\text{state}\) argument is “as if” it called \(\text{do\_in(state, from, from\_end, from, to, to+max, to)}\) for \(to\) pointing to a buffer of at least \(\text{max}\) elements.

13 **Returns:** \((\text{from\_next-\ from})\) where \(\text{from\_next}\) is the largest value in the range \([\text{from, from\_end})\] such that the sequence of values in the range \([\text{from, from\_next})\] represents \(\text{max}\) or fewer valid complete characters of type \(\text{intern\_T}\). The specialization \(\text{codecvt}<\text{char, char, mbstate\_t}>\), returns the lesser of \(\text{max}\) and \((\text{from\_end-\ from})\).

\(\text{int do\_max\_length()}\) const throw();

\(^{237}\) Typically these will be characters to return the state to \(\text{state\_T()}\)

\(^{238}\) If \(\text{encoding()}\) yields -1, then more than \(\text{max\_length()}\) \(\text{extern\_T}\) elements may be consumed when producing a single \(\text{intern\_T}\) character, and additional \(\text{extern\_T}\) elements may appear at the end of a sequence after those that yield the final \(\text{intern\_T}\) character.
Returns: The maximum value that \texttt{do\_length(state, from, from\_end, 1)} can return for any valid range \([from, from\_end)\) and state\(T\) value \texttt{state}. The specialization \texttt{codecvt<char, char, mbstate_t>::do\_max\_length()} returns \(1\).

### 22.2.1.5 Class template codecvt\_byname

```cpp
namespace std {
    template <class internT, class externT, class stateT>
    class codecvt\_byname : public codecvt<internT, externT, stateT> {
        public:
            explicit codecvt\_byname(const char*, size\_t refs = 0);
            explicit codecvt\_byname(const string&, size\_t refs = 0);
        protected:
            codecvt\_byname(); // virtual
        }
    }
}
```

### 22.2.2 The numeric category

The classes \texttt{num\_get<>} and \texttt{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.\(^{239}\)

All specifications of member functions for \texttt{num\_put} and \texttt{num\_get} in the subclauses of 22.2.2 only apply to the specializations required in Tables 73 and 74 (22.1.1.1.1), namely \texttt{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 \texttt{ios\_base\&} argument for formatting specifications (22.2), 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 \texttt{num\_get<>} and \texttt{num\_put<>} member functions for formatting and parsing numeric values (27.6.1.2.1, 27.6.2.6.1).

### 22.2.2.1 Class template num\_get

```cpp
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 \texttt{\_in}, iter\_type \texttt{\_end}, ios\_base\&,  
                          ios\_base::iostate\& \texttt{\_err}, bool\& \texttt{\_v}) const;
            iter\_type get(iter\_type \texttt{\_in}, iter\_type \texttt{\_end}, ios\_base\&,  
                          ios\_base::iostate\& \texttt{\_err}, long\& \texttt{\_v}) const;
            iter\_type get(iter\_type \texttt{\_in}, iter\_type \texttt{\_end}, ios\_base\&,
                          ios\_base::iostate\& \texttt{\_err}, bool\& \texttt{\_v}) const;
        }
    }
```

\(^{239}\) Parsing "-1" correctly into (e.g.) an \texttt{unsigned short} requires that the corresponding member \texttt{get()} at least extract the sign before delegating.
The facet `num_get` is used to parse numeric values from an input sequence such as an istream.

### 22.2.2.1.1 num_get members

The `num_get` facet is a member of the `std::locale` class, which is used to parse numeric values from an input sequence such as an istream. It provides a variety of methods to parse different types of numeric values, such as integers, long integers, floating-point numbers, and long double numbers. The methods `get` are used to parse values for the specified types, while the methods `do_get` provide a virtual implementation for these methods, allowing for customization based on the specific requirements of the local context.

```cpp
protected:
  "num_get(); // virtual
  virtual iter_type do_get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, bool& v) const;
  virtual iter_type do_get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, long& v) const;
  virtual iter_type do_get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, long long& v) const;
  virtual iter_type do_get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, unsigned short& v) const;
  virtual iter_type do_get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, unsigned int& v) const;
  virtual iter_type do_get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, unsigned long& v) const;
  virtual iter_type do_get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, unsigned long long& v) const;
  virtual iter_type do_get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, float& v) const;
  virtual iter_type do_get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, double& v) const;
  virtual iter_type do_get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, long double& v) const;
  virtual iter_type do_get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, void*& v) const;
};
```
22.2 Standard locale categories

22.2.2.1.2 num_get virtual functions

iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, long long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned short& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned int& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned long long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, float& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, double& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, long double& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, void*& val) const;

Returns: do_get(in, end, str, err, val).

Draft
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()`. If an error occurs, `val` is unchanged; otherwise it is set to the resulting value.

The details of this operation occur in three stages

— Stage 1: Determine a conversion specifier
— Stage 2: Extract characters from `in` and determine a corresponding char value for the format expected by the conversion specification determined in stage 1.
— Stage 3: Store results

The details of the stages are presented below.

Stage 1: The function initializes local variables via

```cpp
fmtflags flags = str.flags();
fmtflags basefield = (flags & ios_base::basefield);
fmtflags uppercase = (flags & ios_base::uppercase);
fmtflags boolalpha = (flags & ios_base::boolalpha);
```

For conversion to an integral type, the function determines the integral conversion specifier as indicated in Table 77. 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 78.

Stage 2: If `in == end` then stage 2 terminates. Otherwise a `charT` is taken from `in` and local variables are initialized as if by

```cpp
char_type ct = *in;
char c = src[find(atoms, atoms + sizeof(src) - 1, ct) - atoms];
if ( ct == use_facet<numpunct<charT> >(loc).decimal_point() )
    c = '.';
bool discard =
    ( ct == use_facet<numpunct<charT> >(loc).thousands_sep()
    &&
    use_facet<numpunct<charT> >(loc).grouping().length() != 0 );
```

Draft
Table 78: 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>

where the values src and atoms are defined as if by:

```c
static const char src[] = "0123456789abcdefABCDEF+-";
char_type atoms[sizeof(src)];
use_facet<ctype<charT> >(loc).widen(src, src + sizeof(src), atoms);
```

for this value of loc.

If discard is true, then if '.' has not yet been accumulated, then the position of the character is remembered, but the character is otherwise ignored. Otherwise, if '.' has already been accumulated, the character is discarded and Stage 2 terminates.

If 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 result of stage 2 processing can be one of

- A sequence of chars has been accumulated in stage 2 that is converted (according to the rules of scanf) to a value of the type of val. This value is stored in val and ios_base::goodbit is stored in err.
- The sequence of chars accumulated in stage 2 would have caused scanf to report an input failure. ios_base::failbit is assigned to err.

Digit grouping is checked. That is, the positions of discarded separators is examined for consistency with use_facet<numpunct<charT> >(loc).grouping(). If they are not consistent then ios_base::failbit is assigned to err.

In any case, if stage 2 processing was terminated by the test for in==end then err |=ios_base::eofbit is performed.

```c
iter_type do_get(iter_type in, iter_type end, ios_base& str,
                 ios_base::iostate& err, bool& val) const;
```

**Effects:** If (str.flags()&ios_base::boolalpha)==0 then input proceeds as it would for a long except that if a value is being stored into val, the value is determined according to the following: If the value to be stored is 0 then false is stored. If the value is 1 then true is stored. Otherwise err |=ios_base::failbit is performed and no value is stored.
Otherwise target sequences are determined “as if” by calling the members `falsename()` and `truename()` of the facet obtained by `use_facet<numpunct<charT>>>(str.getloc())`. Successive characters in the range `[in, end)` (see 23.1.1) are obtained and matched against corresponding positions in the target sequences only as necessary to identify a unique match. The input iterator `in` is compared to `end` only when necessary to obtain a character. If and only if a target sequence is uniquely matched, `val` is set to the corresponding value.

The `in` iterator is always left pointing one position beyond the last character successfully matched. If `val` is set, then `err` is set to `str.goodbit` or to `str.eofbit` if, when seeking another character to match, it is found that (`in == end`). If `val` is not set, then `err` is set to `str.failbit` or to (`str.failbit`|`str.eofbit`) if the reason for the failure was that (`in == end`). [Example: For targets true: "a" and false: "abb", the input sequence "a" yields `val == true` and `err == str.eofbit`; the input sequence "abc" yields `err == str.failbit`, with `in` ending at the 'c' element. For targets true: "1" and false: "0", the input sequence "1" yields `val == true` and `err == str.goodbit`. For empty targets (""), any input sequence yields `err == str.failbit`. — end example]

Returns: `in`.

### 22.2.2.2 Class template `num_put`

```cpp
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;
        iter_type put(iter_type s, ios_base& f, char_type fill, double v) const;
        iter_type put(iter_type s, ios_base& f, char_type fill, long double v) const;
        iter_type put(iter_type s, ios_base& f, char_type fill, const void* v) const;

        static locale::id id;

    protected:
        ~num_put(); // virtual
        virtual iter_type do_put(iter_type s, ios_base& f, char_type fill, bool v) const;
        virtual iter_type do_put(iter_type s, ios_base& f, char_type fill, double v) const;
    }
}
```
The facet `num_put` is used to format numeric values to a character sequence such as an ostream.

### 22.2.2.2.1 num_put members

```cpp
iter_type put(iter_type out, ios_base& str, char_type fill, bool val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, long long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, unsigned long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, double val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, long double val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, const void* val) const;
```

Returns: `do_put(out, str, fill, val)`.

### 22.2.2.2 num_put virtual functions

```cpp
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;
```

Draft
Effects: Writes characters to the sequence out, formatting val as desired. In the following description, a local variable initialized with

\[ \text{locale loc = str.getloc();} \]

The details of this operation occur in several stages:

— Stage 1: Determine a printf conversion specifier spec and determining the characters that would be printed by printf(27.8.2) given this conversion specifier for

\[ \text{printf(spec, val)} \]

assuming that the current locale is the "C" locale.

— Stage 2: Adjust the representation by converting each char determined by stage 1 to a charT using a conversion and values returned by members of use_facet&lt; numpunct&lt;charT&gt; &gt;(str.getloc())

— Stage 3: Determine where padding is required.

— Stage 4: Insert the sequence into the out.

Detailed descriptions of each stage follow.

Returns: out.

Stage 1: The first action of stage 1 is to determine a conversion specifier. The tables that describe this determination use the following local variables

\[
\begin{align*}
\text{fmtflags flags} &= \text{str.flags()} ; \\
\text{fmtflags basefield} &= (\text{flags} & (\text{ios_base::basefield})) ; \\
\text{fmtflags uppercase} &= (\text{flags} & (\text{ios_base::uppercase})) ; \\
\text{fmtflags floatfield} &= (\text{flags} & (\text{ios_base::floatfield})) ; \\
\text{fmtflags showpos} &= (\text{flags} & (\text{ios_base::showpos})) ; \\
\text{fmtflags showbase} &= (\text{flags} & (\text{ios_base::showbase})) ;
\end{align*}
\]

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 79.

For conversion from a floating-point type, the function determines the floating-point conversion specifier as indicated in Table 80:

For conversions from an integral or floating type a length modifier is added to the conversion specifier as indicated in Table 81

Draft
### Table 79: 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; !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 80: 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 81: Length Modifier

<table>
<thead>
<tr>
<th>type</th>
<th>length modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>1</td>
</tr>
<tr>
<td>long long</td>
<td>ll</td>
</tr>
<tr>
<td>unsigned long</td>
<td>1</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>ll</td>
</tr>
<tr>
<td>long double</td>
<td>L</td>
</tr>
<tr>
<td>otherwise</td>
<td>none</td>
</tr>
</tbody>
</table>
The conversion specifier has the following optional additional qualifiers prepended as indicated in Table 82.

<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>

For conversion from a floating-point type, `str.precision()` is specified in the conversion specification.

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

```c
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 22.2.3.1.2.

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\(^{240}\) is determined according to Table 83.

<table>
<thead>
<tr>
<th>State</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjustfield == ios_base::left</td>
<td>pad after</td>
</tr>
<tr>
<td>adjustfield == ios_base::right</td>
<td>pad before</td>
</tr>
<tr>
<td>adjustfield == internal and a sign occurs in the representation</td>
<td>pad after the sign</td>
</tr>
<tr>
<td>adjustfield == internal and representation after stage 1 began with 0x of 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()`.

\(^{240}\) The conversion specification `#o` generates a leading 0 which is not a padding character.
Stage 4: The sequence of charT's at the end of stage 3 are output via

```cpp
*out++ = c
```

iter_type do_put(iter_type out, ios_base& str, char_type fill, bool val) const;

6 Returns: If `str.flags() & ios_base::boolalpha) == 0` returns `do_put(out, str, fill, (int)val)`, otherwise obtains a string s as if by

```cpp
string_type s =
val ? use_facet<ctype<charT>>>(loc).truename()
  : use_facet<ctype<charT>>>(loc).falsename();
```

and then inserts each character c of s into out via `*out++ = c` and returns out.

22.2.3 The numeric punctuation facet

22.2.3.1 Class template numpunct

```cpp
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;
    string_type truename() const;
    string_type falsename() const;

    static locale::id id;

    protected:
      "numpunct(); //virtual
      virtual char_type do_decimal_point() const;
      virtual char_type do_thousands_sep() const;
      virtual string do_grouping() const;
      virtual string_type do_truename() const; //for bool
      virtual string_type do_falsename() const; //for bool
    }
  }
}
```

Draft
numpunct<> specifies numeric punctuation. The specializations required in Table 73 (22.1.1.1.1), namely numpunct< wchar_t> and numpunct< char>, provide classic "C" numeric formats, i.e. they contain information equivalent to that contained in the "C" locale or their wide character counterparts as if obtained by a call to widen.

The syntax for number formats is as follows, where digit represents the radix set specified by the fmtflags argument value, and thousands-sep and decimal-point are the results of corresponding numpunct<charT> members. Integer values have the format:

integer ::= [sign] units
sign ::= plusminus
plusminus ::= '+' | '-'
units ::= digits [thousands-sep units]
digits ::= digit [digits]

and floating-point values have:

floatval ::= [sign] units [decimal-point [digits]] [e [sign] digits] |
[sign] decimal-point digits [e [sign] digits]
e ::= 'e' | 'E'

where the number of digits between thousands-seps is as specified by do_grouping(). For parsing, if the digits portion contains no thousands-separators, no grouping constraint is applied.

22.2.3.1.1 numpunct members

char_type decimal_point() const;
1    Returns: do_decimal_point()

char_type thousands_sep() const;
2    Returns: do_thousands_sep()

string grouping() const;
3    Returns: do_grouping()

string_type truename() const;

string_type false_name() const;

4    Returns: do_truename() or do_falsename(), respectively.

22.2.3.1.2 numpunct virtual functions

char_type do_decimal_point() const;
1    Returns: A character for use as the decimal radix separator. The required specializations return \',\', or L\',\'.

char_type do_thousands_sep() const;
2    Returns: A character for use as the digit group separator. The required specializations return \',\', or L\',\'.

string do_grouping() const;

Draft
22.2 Standard locale categories

Returns: A basic_string<char> vec used as a vector of integer values, in which each element vec[i] represents the number of digits\(^{241}\) in the group at position i, starting with position 0 as the rightmost group. If vec.size() \(\leq i\), the number is the same as group \((i-1)\); if \(i<0\) \(\|\) vec[i]<=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".

22.2.3.2 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();" // virtual
                ;
    }
}

22.2.4 The collate category

22.2.4.1 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);

            int compare(const charT* low1, const charT* high1, const charT* low2, const charT* high2) const;
            string_type transform(const charT* low, const charT* high) const;
            long hash(const charT* low, const charT* high) const;
}

\(^{241}\)Thus, the string "\003" specifies groups of 3 digits each, and "3" probably indicates groups of 51 (!) digits each, because 51 is the ASCII value of "3".
The class `collate<charT>` provides features for use in the collation (comparison) and hashing of strings. A locale member function template, `operator()`, uses the collate facet to allow a locale to act directly as the predicate argument for standard algorithms (clause 25) and containers operating on strings. The specializations required in Table 73 (22.1.1.1.1), namely `collate<char>` and `collate<wchar_t>`, apply lexicographic ordering (25.3.8).

Each function compares a string of characters `*p` in the range `[low, high)`.

### 22.2.4.1.1 `collate` members

```cpp
int compare(const charT* low1, const charT* high1,
            const charT* low2, const charT* high2) const;
```

**Returns:** `do_compare(low1, high1, low2, high2)`

```cpp
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.\(^{242}\)

### 22.2.4.1.2 `collate` virtual functions

```cpp
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 73 (22.1.1.1.1), namely `collate<char>` and `collate<wchar_t>`, implement a lexicographical comparison (25.3.8).

```cpp
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.\(^{242}\)

---

\(^{242}\) This function is useful when one string is being compared to many other strings.
22.2 Standard locale categories

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]

22.2.4.2 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(); // virtual
    }
}
```

22.2.5 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 22.2.5 only apply to the specializations required in Tables 73 and 74 (22.1.1.1.1). Their members use their `ios_base&`, `ios_base::iostate&`, and `fill` arguments as described in (22.2), and the `ctype<>` facet, to determine formatting details.

22.2.5.1 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;
    }
}
```
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.\footnote{In other words, user confirmation is required for reliable parsing of user-entered dates and times, but machine-generated formats can be parsed reliably. This allows parsers to be aggressive about interpreting user variations on standard formats.}

If the end iterator is reached during parsing by any of the get() member functions, the member sets ios_base::eofbit in err.

### 22.2.5.1.1 time_get members

- **elapsed_time**
  - **date_order()** const;
  - **Returns:** do_date_order()

- **iter_type** get_time(iter_type s, iter_type end, ios_base& str, 
  ios_base::iostate& err, tm* t) const;
  - **Returns:** do_get_time(s, end, str, err, t)

- **iter_type** get_date(iter_type s, iter_type end, ios_base& str, 
  ios_base::iostate& err, tm* t) const;
Returns: do_get_date(s, end, str, err, t)

iter_type get_weekday(iter_type s, iter_type end, ios_base& str,
    ios_base::iostate& err, tm* t) const;

Returns: do_get_weekday(s, end, str, err, t) or do_get_monthname(s, end, str, err, t)

iter_type get_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)

Requires: [fmt,end] 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:

- The expression fmt == end evaluates to true.
- The expression err == ios_base::goodbit evaluates to false.
- The expression s == end evaluates to true, in which case the function evaluates err = ios_base::eofbit | ios_base::failbit.
- The next element of fmt is equal to ‘%’, optionally followed by a modifier character, followed by a conversion specifier character, format, together forming a conversion specification valid for the ISO/IEC 9945 function strftime. If the number of elements in the range [fmt,end) is not sufficient to unambiguously determine whether the conversion specification is complete and valid, the function evaluates err = ios_base::failbit. Otherwise, the function evaluates s = do_get(s, end, f, err, t, format, modifier), where the value of modifier is ‘\0’ when the optional modifier is absent from the conversion specification. If err == ios_base::goodbit holds after the evaluation of the expression, the function increments fmt to point just past the end of the conversion specification and continues looping.

- The expression isspace(*fmt, f.getloc()) evaluates to true, in which case the function first increments fmt until fmt == end || !isspace(*fmt, f.getloc()) evaluates to true, then advances s until s == end || !isspace(*s) is true, and finally resumes looping.

- The next character read from s matches the element pointed to by fmt in a case-insensitive comparison, in which case the function evaluates ++fmt, ++s and continues looping. Otherwise, the function evaluates err = ios_base::failbit.
\[ Note: \] The function uses the ctype<charT> facet installed in \( f' \)’s locale to determine valid whitespace characters. It is unspecified by what means the function performs case-insensitive comparison or whether multi-character sequences are considered while doing so.

Returns: \( s \)

### 22.2.5.1.2 time_get virtual functions

**dateorder do_date_order() const;**

*Returns:* An enumeration value indicating the preferred order of components for those date formats that are composed of day, month, and year.\(^{244}\) Returns no_order if the date format specified by \( \text{'x'} \) contains other variable components (e.g. Julian day, week number, week day).

**iter_type do_get_time(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm* t) const;**

*Effects:* Reads characters starting at \( s \) until it has extracted those struct tm members, and remaining format characters, used by time_put<>::put to produce the format specified by \( "%H:%M:%S" \), or until it encounters an error or end of sequence.

*Returns:* An iterator pointing immediately beyond the last character recognized as possibly part of a valid time.

**iter_type do_get_date(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm* t) const;**

*Effects:* Reads characters starting at \( s \) until it has extracted those struct tm members, and remaining format characters, used by time_put<>::put to produce one of the following formats, or until it encounters an error. The format depends on the value returned by date_order() as in Table 84:

<table>
<thead>
<tr>
<th>date_order()</th>
<th>format</th>
</tr>
</thead>
<tbody>
<tr>
<td>no_order</td>
<td>&quot;%m%d%y&quot;</td>
</tr>
<tr>
<td>dmy</td>
<td>&quot;%d%m%y&quot;</td>
</tr>
<tr>
<td>mdy</td>
<td>&quot;%m%d%y&quot;</td>
</tr>
<tr>
<td>ymd</td>
<td>&quot;%y%m%d&quot;</td>
</tr>
<tr>
<td>ydm</td>
<td>&quot;%y%d%m&quot;</td>
</tr>
</tbody>
</table>

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.

\(^{244}\) 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 the (perhaps abbreviated) name of a weekday or month. If it finds an abbreviation that is followed by characters that could match a full name, it continues reading until it matches the full name or fails. It sets the appropriate struct tm member accordingly.

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid name.

```cpp
iter_type do_get_year(iter_type s, iter_type end, ios_base& str,
         ios_base::iostate& err, tm* t) const;
```

Effects: Reads characters starting at s until it has extracted an unambiguous year identifier. It is implementation-defined whether two-digit year numbers are accepted, and (if so) what century they are assumed to lie in. Sets the t->tm_year member accordingly.

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid year identifier.

```cpp
iter_type do_get(iter_type s, iter_type end, ios_base& f,
         ios_base::iostate& err, tm* t, char format, char modifier) const;
```

Requires: [fmt,end) shall be a valid range and t shall be dereferenceable.

Effects: The function starts by evaluating err = ios_base::goodbit. It then reads characters starting at s until it encounters an error, or until it has extracted and assigned those struct tm members, and any remaining format characters, corresponding to a conversion directive appropriate for the ISO/IEC 9945 function strftime, formed by concatenating ‘%’, the modifier character, when non-NUL, and the format character. When the concatenation fails to yield a complete valid directive the function leaves the object pointed to by t unchanged and evaluate:

```cpp
err |= ios_base::failbit; When s == end evaluates to true after reading a character the function evaluates
err |= ios_base::eofbit.
```

For complex conversion directives such as %c, %x, or %X, or directives that involve the optional modifiers E or O, when the function is unable to unambiguously determine some or all struct tm members from the input sequence [s,end), it evaluates err |= ios_base::eofbit. In such cases the values of those struct tm members are unspecified and may be outside their valid range.

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.

Returns: An iterator pointing immediately beyond the last character recognized as possibly part of a valid input sequence for the given format and modifier.

22.2.5.2 Class template time_getbyname

```cpp
namespace std {
    template <class charT, class InputIterator = istreambuf_iterator<charT> >
    class time_getbyname : public time_get<charT, InputIterator> {
        public:
            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);
        }

    [locale.time.getbyname]
```
protected:
~time_get_byname();    // virtual
};

22.2.5.3 Class template time_put

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);

    // the following is implemented in terms of other member functions.
    iter_type put(iter_type s, ios_base& f, char_type fill, const tm* tmb,
                   const charT* pattern, const charT* pat_end) const;
    iter_type put(iter_type s, ios_base& f, char_type fill,
                  const tm* tmb, char format, char modifier = 0) const;

    static locale::id id;

protected:
~time_put();    // virtual
    virtual iter_type do_put(iter_type s, ios_base& str, char_type, const tm* t,
                              char format, char modifier) const;
};
}

22.2.5.3.1 time_put members

iter_type put(iter_type s, ios_base& str, char_type fill, const tm* t,
              const charT* pattern, const charT* pat_end) const;
iter_type put(iter_type s, ios_base& str, char_type fill, const tm* t,
              char format, char modifier = 0) const;

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).

Draft

245) Although the C programming language defines no modifiers, most vendors do.
22.2 Standard locale categories

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.

22.2.5.3.2 time_put virtual functions

iter_type do_put(iter_type s, ios_base& f, 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().\(^{246}\) except that the sequence of characters produced for those specifiers that are described as depending on the C locale are instead implementation-defined.\(^{247}\)

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]

22.2.5.4 Class template time_put_byname

namespace std {

template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
class time_put_byname : public time_put<charT, OutputIterator>
{
public:
    typedef charT char_type;
    typedef OutputIterator iter_type;

    explicit time_put_byname(const char* name = 0);
    explicit time_put_byname(const string& name, size_t refs = 0);

protected:
    ~time_put_byname(); // virtual
};
}

22.2.6 The monetary category

These templates handle monetary formats. A template parameter indicates whether local or international monetary formats are to be used.

All specifications of member functions for money_put and money_get in the subclauses of 22.2.6 only apply to the specializations required in Tables 73 and 74 (22.1.1.1.1). Their members use their ios_base, ios_base::iostate,
and \texttt{fill} arguments as described in (22.2), and the \texttt{moneypunct<>} and \texttt{ctype<>} facets, to determine formatting details.

\subsection*{22.2.6.1 Class template \texttt{money\_get}}

\begin{verbatim}
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
        virtual iter_type do_get(iter_type, iter_type, bool, ios_base& str, ios_base::iostate& err, long double& units) const;
        virtual iter_type do_get(iter_type, iter_type, bool, ios_base& str, ios_base::iostate& err, string_type& digits) const;
    };
}
\end{verbatim}

\subsubsection*{22.2.6.1.1 \texttt{money\_get} members}

\begin{verbatim}
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(s, iter_type end, bool intl, ios_base& f,
    ios_base::iostate& err, string_type& digits) const;
\end{verbatim}

\textit{Returns:} \texttt{do\_get(s, end, Intl, F, Err, Quant)}

\subsubsection*{22.2.6.1.2 \texttt{money\_get} virtual functions}

\begin{verbatim}
iter_type do_get(iter_type s, iter_type end, bool Intl,
    ios_base& str, ios_base::iostate& Err,
    long double& units) const;
iter_type do_get(iter_type s, iter_type end, bool Intl,
    ios_base& str, ios_base::iostate& Err,
\end{verbatim}

Draft
Effects: Reads characters from s to parse and construct a monetary value according to the format specified by a moneypunct<charT,Intl> facet reference mp and the character mapping specified by a ctype<charT> facet reference ct obtained from the locale returned by str.getloc() and str.flags(). If a valid sequence is recognized, does not change err; otherwise, sets err to (err | str.failbit), or (err | str.failbit | str.eofbit) if no more characters are available, and does not change units or digits. Uses the pattern returned by mp.neg_format() to parse all values. The result is returned as an integral value stored in units or as a sequence of digits possibly preceded by a minus sign (as produced by ct.widen(c) where c is '-' or in the range from '0' through '9', inclusive) stored in digits. [Example: The sequence $1,056.23 in a common United States locale would yield, for units, 105623, or, for digits, "105623". —end example] If mp.grouping() indicates that no thousands separators are permitted, any such characters are not read, and parsing is terminated at the point where they first appear. Otherwise, thousands separators are optional; if present, they are checked for correct placement only after all format components have been read.

Where space or none appears in the format pattern, except at the end, optional white space (as recognized by ct.is) is consumed after any required space. If (str.flags() & str.showbase) is false, the currency symbol is optional and is consumed only if other characters are needed to complete the format; otherwise, the currency symbol is required.

If the first character (if any) in the string pos returned by mp.positive_sign() or the string neg returned by mp.negative_sign() is recognized in the position indicated by sign in the format pattern, it is consumed and any remaining characters in the string are required after all the other format components. [Example: If showbase is off, then for a neg value of "()" and a currency symbol of "L", in "(100 L)" the "L" is consumed; but if neg is "-", the "L" in "-100 L" is not consumed. —end example] If pos or neg is empty, the sign component is optional, and if no sign is detected, the result is given the sign that corresponds to the source of the empty string. Otherwise, the character in the indicated position must match the first character of pos or neg, and the result is given the corresponding sign. If the first character of pos is equal to the first character of neg, or if both strings are empty, the result is given a positive sign.

Digits in the numeric monetary component are extracted and placed in digits, or into a character buffer buf1 for conversion to produce a value for units, in the order in which they appear, preceded by a minus sign if and only if the result is negative. The value units is produced as if by

```
for (int i = 0; i < n; ++i)
    buf2[i] = src[find(atoms, atoms+sizeof(src), buf1[i]) - atoms];
buf2[n] = 0;
sscanf(buf2, "%Lf", &units);
```

where n is the number of characters placed in buf1, buf2 is a character buffer, and the values src and atoms are defined as if by

```c
static const char src[] = "0123456789-";
charT atoms[sizeof(src)];
cwiden(src, src + sizeof(src) - 1, atoms);
```

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid monetary quantity.

248) The semantics here are different from ct.narrow.


22.2.6.2 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
            virtual iter_type
do_put(iter_type s, bool intl, ios_base& f,
                  char_type fill, long double units) const;
            virtual iter_type
do_put(iter_type s, bool intl, ios_base& f,
                  char_type fill, const string_type& digits) const;
        }
    }
}
```

22.2.6.2.1 money_put members

```
iter_type put(iter_type s, bool intl, ios_base& f,
             char_type fill, long double quant) const;
iter_type put(iter_type s, bool intl, ios_base& f,
             char_type fill, const string_type& quant) const;
```

1. **Returns:** do_put(s, intl, f, loc, quant)

22.2.6.2.2 money_put virtual functions

```
iter_type do_put(iter_type s, bool intl, ios_base& str,
                 char_type fill, long double units) const;
iter_type do_put(iter_type s, bool intl, ios_base& str,
                 char_type fill, const string_type& digits) const;
```

1. **Effects:** Writes characters to s according to the format specified by a 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

\[
\text{ct.widen}(\text{buf1}, \text{buf1} + \text{sprintf}(\text{buf1}, "\%.0Lf", \text{units}), \text{buf2})
\]

for character buffers `buf1` and `buf2`. If the first character in `digits` or `buf2` is equal to `ct.widen('-')`, then the pattern used for formatting is the result of `mp.neg_format()`; otherwise the pattern is the result of `mp.pos_format()`. Digit characters are written, interspersed with any thousands separators and decimal point specified by the format, in the order they appear (after the optional leading minus sign) in `digits` or `buf2`. In `digits`, only the optional leading minus sign and the immediately subsequent digit characters (as classified according to `ct`) are used; any trailing characters (including digits appearing after a non-digit character) are ignored. Calls `str.width(0)`.

2 **Remarks:** The currency symbol is generated if and only if `(str.flags() & str.showbase)` is nonzero. If the number of characters generated for the specified format is less than the value returned by `str.width()` on entry to the function, then copies of `fill` are inserted as necessary to pad to the specified width. For the value `af` equal to `(str.flags() & str.adjustfield)`, if `(af == str.internal)` is true, the fill characters are placed where none or space appears in the formatting pattern; otherwise if `(af == str.left)` is true, they are placed after the other characters; otherwise, they are placed before the other characters. [Note: It is possible, with some combinations of format patterns and flag values, to produce output that cannot be parsed using `num_get<>::get`. — end note]

3 **Returns:** An iterator pointing immediately after the last character produced.

### 22.2.6.3 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;
    }
}
```
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\(^{249}\) In the `field` member of a `pattern` object, each value `symbol`, `sign`, `value`, and either `space` or `none` appears exactly once. The value `none`, if present, is not first; the value `space`, if present, is neither first nor last.

Where `none` or `space` appears, white space is permitted in the format, except where `none` appears at the end, in which case no white space is permitted. The value `space` indicates that at least one space is required at that position. Where `symbol` appears, the sequence of characters returned by `curr_symbol()` is permitted, and can be required. Where `sign` appears, the first (if any) of the sequence of characters returned by `positive_sign()` or `negative_sign()` (respectively as the monetary value is non-negative or negative) is required. Any remaining characters of the sign sequence are required after all other format components. Where `value` appears, the absolute numeric monetary value is required.

The format of the numeric monetary value is a decimal number:

\[
\text{value ::= units \ [ decimal-point \ [ digits \] \] \ |} \\
\text{decimal-point digits}
\]

if `frac_digits()` returns a positive value, or

\[
\text{value ::= units}
\]

otherwise. The symbol `decimal-point` indicates the character returned by `decimal_point()`. The other symbols are defined as follows:

\[
\text{units ::= digits \ [ thousands-sep units \]}
\text{digits ::= adigit \ [ digits \]}
\]

\(^{249}\) An array of `char`, rather than an array of `part`, is specified for `pattern::field` purely for efficiency.
In the syntax specification, the symbol `digit` 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()`.

### 22.2.6.3.1 moneypunct members

```cpp
charT decimal_point() const;
charT thousands_sep() const;
string grouping() const;
string_type curr_symbol() const;
string_type positive_sign() const;
string_type negative_sign() const;
int frac_digits() const;
pattern pos_format() const;
pattern neg_format() const;
```

Each of these functions `F` returns the result of calling the corresponding virtual member function `do_`F`()`.

### 22.2.6.3.2 moneypunct virtual functions

```cpp
charT do_decimal_point() const;

Returns: The radix separator to use in case `do_frac_digits()` is greater than zero.\(^{250}\)
```

```cpp
charT do_thousands_sep() const;

Returns: The digit group separator to use in case `do_grouping()` specifies a digit grouping pattern.\(^{251}\)
```

```cpp
string do_grouping() const;

Returns: A pattern defined identically as, but not necessarily equal to, the result of `numpunct<>::do_grouping()`.\(^{252}\)
```

```cpp
string_type do_curr_symbol() const;

Returns: A string to use as the currency identifier symbol.\(^{253}\)
```

```cpp
string_type do_positive_sign() const;
string_type do_negative_sign() const;

Returns: `do_positive_sign()` returns the string to use to indicate a positive monetary value; `do_negative_sign()` returns the string to use to indicate a negative value.\(^{254}\)
```

---

\(^{250}\) In common U.S. locales this is `'.'`.

\(^{251}\) In common U.S. locales this is `,`.

\(^{252}\) To specify grouping by 3s, the value is "\003" (not "3").

\(^{253}\) For international specializations (second template parameter `true`) this is always four characters long, usually three letters and a space.

\(^{254}\) This is usually the empty string.

Draft
int do_frac_digits() const;

Returns: The number of digits after the decimal radix separator, if any.\textsuperscript{255}

pattern do_pos_format() const;
pattern do_neg_format() const;

Returns: The specializations required in Table 74 (22.1.1.1.1), namely \texttt{moneypunct<char>}, \texttt{moneypunct<wchar_t>}, \texttt{moneypunct<char,true>}, and \texttt{moneypunct<wchar_t,true>}, return an object of type \texttt{pattern} initialized to \{ symbol, sign, none, value \}.\textsuperscript{256}

\textbf{22.2.6.4 Class template \texttt{moneypunct\_byname}}

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();  // virtual
    };
}

\textbf{22.2.7 The message retrieval category}

Class \texttt{messages<charT>} implements retrieval of strings from message catalogs.

\textbf{22.2.7.1 Class template \texttt{messages}}

namespace std {
    class messages_base {
    public:
        typedef int 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;
    }
}

\textsuperscript{255} In common U.S. locales, this is 2.

\textsuperscript{256} Note that the international symbol returned by \texttt{do\_curr\_sym()} usually contains a space, itself; for example, "USD ".

Draft
22.2 Standard locale categories

```cpp
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
    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;
};
```

Values of type `messages_base::catalog` usable as arguments to members `get` and `close` can be obtained only by calling member `open`.

### 22.2.7.1.1 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;
```

1. **Returns:** `do_get(cat, set,msgid, dfault)`.

```
void close(catalog cat) const;
```

3. **Effects:** Calls `do_close(cat)`.

### 22.2.7.1.2 messages virtual functions

```
catalog do_open(const basic_string<char>& name,
    const locale& loc) const;
```

1. **Returns:** A value that may be passed to `get()` to retrieve a message, from the message catalog identified by the string `name` according to an implementation-defined mapping. The result can be used until it is passed to `close()`.

2. Returns a value less than 0 if no such catalog can be opened.

3. **Remarks:** The locale argument `loc` is used for character set code conversion when retrieving messages, if needed.

```
string_type do_get(catalog cat, int set, intmsgid,
    const string_type& dfault) const;
```

4. **Requires:** A catalog `cat` obtained from `open()` and not yet closed.
Returns: A message identified by arguments \textit{set}, \textit{msgid}, and \textit{default}, according to an implementation-defined mapping. If no such message can be found, returns \textit{default}.

void do_close(catalog \textit{cat}) \textit{const};

Requires: A catalog \textit{cat} obtained from open() and not yet closed.

Effects: Releases unspecified resources associated with \textit{cat}.

Remarks: The limit on such resources, if any, is implementation-defined.

\subsection*{22.2.7.2 Class template \texttt{messages\_byname}}

namespace std {
    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 \textit{refs} = 0);
        explicit messages\_byname(const string& \textit{refs} = 0);
        protected:
            ^messages\_byname(); // virtual
    }
}

\subsection*{22.2.8 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 \texttt{locale\_\_\_\_facet} a class containing a static member: \texttt{static \texttt{locale\_\_\_\_id}} \texttt{id}.

\begin{itemize}
    \item \textbf{[Note:} The locale member function templates verify its type and storage class. \textit{— end note}]
    \item \textbf{[Note:} This paragraph is intentionally empty. \textit{— end note}]
    \item \textbf{[Example:} Traditional global localization is still easy:

        \begin{verbatim}
        \#include <iostream>
        \#include <locale>
        \n        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());
        }
        \end{verbatim}
    \end{itemize}
5 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 ]

6 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.

7 Example: Here is an example of the use of locales in a library interface.

```cpp
// include <iosfwd>
#include <locale>

class Date {
...
public:
    Date(unsigned day, unsigned month, unsigned year);
    std::string asString(const std::locale& = std::locale());
};
```

8 This example illustrates two architectural uses of class locale.

9 The first is as a default argument in `Date::asString()`, where the default is the global (presumably user-preferred) locale.

10 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.

```
// include "Date"    // includes <ctime>
```
#include <sstream>
std::string Date::asString(const std::locale& l)
{
    using namespace std;
    ostringstream s; s.imbue(l);
    s << *this; return s.str();
}

std::istream& operator>>(std::istream& s, Date& d)
{
    using namespace std;
    istream::sentry cerberos(s);
    if (cerberos) {
        ios_base::iostate err = goodbit;
        struct tm t;
        use_facet< time_get<char> >(s.getloc()).get_date(s, 0, s, err, &t);
        if (!err) d = Date(t.tm_day, t.tm_mon + 1, t.tm_year + 1900);
        s.setstate(err);
    }
    return s;
}

--- end example ---

A locale object may be extended with a new facet simply by constructing it with an instance of a class derived from locale::facet. The only member a C++ program must define is the static member id, which identifies your class interface as a new facet.

[Example: Classifying Japanese characters:

```cpp
// file: <jctype>
#include <locale>
namespace My {
    using namespace std;
    class JCtype : public locale::facet {
    public:
        static locale::id id; // required for use as a new locale facet
        bool is_kanji (wchar_t c) const;
        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;
}

The new facet is used exactly like the built-in facets. — end example }

[ Example: Replacing an existing facet is even easier. Here we do not define a member id because we are reusing the numpunct<

    charT> facet interface:

    // file: my_bool.C
    #include <iostream>
    #include <locale>
    #include <string>
    namespace My {
        using namespace std;
typedef numpunct_byname<char> cnumpunct;
class BoolNames : public cnumpunct {
    protected:
        string do_truename() const { return "Oui Oui!"; }
        string do_falsename() const { return "Mais Non!"; }
    BoolNames() {}
    public:
        BoolNames(const char* name) : cnumpunct(name) {}
    }
    }

    int main(int argc, char** argv)
    {
        using namespace std;
        // make the user’s preferred locale, except for...
        locale loc(locale(""), new My::BoolNames(""));
cout.imbue(loc);
cout << boolalpha << "Any arguments today? " << (argc > 1) << endl;
return 0;
}

— end example ]

22.3 C Library Locales

Header <clocale> (Table 85):

The contents are the same as the Standard C library header <locale.h>.

SEE ALSO: ISO C clause 7.4.

Draft
Table 85: Header `<locale>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td>LC_ALL LC_COLLATE LC_TYPE</td>
</tr>
<tr>
<td></td>
<td>LC_MONETARY LC_NUMERIC 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 setlocale</td>
</tr>
</tbody>
</table>
Draft
Chapter 23  Containers library

This clause describes components that C++ programs may use to organize collections of information.

The following subclauses describe container requirements, and components for sequence containers and associative containers, as summarized in Table 86:

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.1 Requirements</td>
<td>&lt;array&gt;</td>
</tr>
<tr>
<td>23.2 Sequence containers</td>
<td>&lt;deque&gt;</td>
</tr>
<tr>
<td>23.2 Sequence containers</td>
<td>&lt;list&gt;</td>
</tr>
<tr>
<td>23.2 Sequence containers</td>
<td>&lt;queue&gt;</td>
</tr>
<tr>
<td>23.2 Sequence containers</td>
<td>&lt;stack&gt;</td>
</tr>
<tr>
<td>23.2 Sequence containers</td>
<td>&lt;vector&gt;</td>
</tr>
<tr>
<td>23.3 Associative containers</td>
<td>&lt;map&gt;</td>
</tr>
<tr>
<td>23.3 Associative containers</td>
<td>&lt;set&gt;</td>
</tr>
<tr>
<td>23.3.5 bitset</td>
<td>&lt;bitset&gt;</td>
</tr>
<tr>
<td>23.4 Unordered associative containers</td>
<td>&lt;unordered_map&gt;</td>
</tr>
<tr>
<td>23.4 Unordered associative containers</td>
<td>&lt;unordered_set&gt;</td>
</tr>
</tbody>
</table>

23.1 Container requirements

Containers are objects that store other objects. They control allocation and deallocation of these objects through constructors, destructors, insert and erase operations.

All of the complexity requirements in this clause are stated solely in terms of the number of operations on the contained objects. [Example: the copy constructor of type vector <vector<int> > has linear complexity, even though the complexity of copying each contained vector<int> is itself linear. — end example ]

Objects stored in these components shall be MoveConstructible and MoveAssignable. If the copy constructor of a container is used, objects stored in that container shall be CopyConstructible. If the copy assignment operator of a sequence container is used, objects stored in that container shall be CopyConstructible and CopyAssignable. If the copy assignment operator of an associative container is used, objects stored in that container shall be CopyConstructible.
In Tables 87 and 88, \( X \) denotes a container class containing objects of type \( T \), \( a \) and \( b \) denote values of type \( X \), \( u \) denotes an identifier, \( r \) denotes an lvalue or a const rvalue of type \( X \), and \( rv \) denotes a non-const rvalue of type \( X \).

**Table 87: Container requirements**

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>operational semantics</th>
<th>assertion/note pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X::value_-type )</td>
<td>( T )</td>
<td></td>
<td>( T ) is CopyConstructible</td>
<td>compile time</td>
</tr>
<tr>
<td>( X::reference )</td>
<td>lvalue of ( T )</td>
<td></td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>( X::const_-reference )</td>
<td>const lvalue of ( T )</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 except output iterator, 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 except output iterator</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>( X::difference_type )</td>
<td>signed integral 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 integral type</td>
<td>size_type can represent any non-negative value of difference_type</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>( X u; )</td>
<td></td>
<td>post: ( u.size() == 0 )</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>( X(); )</td>
<td></td>
<td>( X().size() == 0 )</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>( X(a); )</td>
<td></td>
<td>( a == X(a) )</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>( X u(a); )</td>
<td></td>
<td>post: ( u == a )</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>( X u = a; )</td>
<td></td>
<td>Equivalent to: ( X u; u = a; )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( X u(rv); )</td>
<td></td>
<td>u shall be equal to the value that ( rv ) had before this construction</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>( X u = rv; )</td>
<td></td>
<td>Equivalent to: ( X u; u = rv; )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( a = rv; )</td>
<td>( X &amp; )</td>
<td>a shall be equal to the value that ( rv ) had before this construction</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>( (&amp;a)\rightarrow X(); )</td>
<td>void</td>
<td>note: the destructor is applied to every element of ( a ); all the memory is deallocated.</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>expression</td>
<td>return type</td>
<td>operational semantics</td>
<td>assertion/note</td>
<td>complexity</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>-----------------------</td>
<td>----------------</td>
<td>------------</td>
</tr>
<tr>
<td>a.begin();</td>
<td>iterator;</td>
<td>suitable for constant a</td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>a.end();</td>
<td>iterator;</td>
<td>suitable for constant a</td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>a.cbegin();</td>
<td>iterator</td>
<td>suitable for constant a</td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>a.cend();</td>
<td>iterator</td>
<td>suitable for constant a</td>
<td></td>
<td>constant</td>
</tr>
</tbody>
</table>
| a == b     | convertible to bool | == is an equivalence relation. a.size() == b.size() &
|            |             | equal(a.begin(), a.end(), b.begin(), b.end()) |                | linear     |
| a != b     | convertible to bool | Equivalent to: !(a == b) |                | linear     |
| a.swap(b); | void        | swap(a,b) | (Note A)   |
| r = a X&   | post: r == a | (Note A) | (Note A) |
| a.size()   | size_type   | a.end() – a.begin() |                | linear     |
| a.max_size() | size_type | size() of the largest possible container |                | (Note A) |
| a.empty()  | convertible to bool | a.size() == 0 |                | constant   |
| a < b      | convertible to bool | lexicographical_compare( a.begin(), a.end(), b.begin(), b.end()) | pre: < is defined for values of T. < is a total ordering relationship. | linear     |
| a > b      | convertible to bool | b < a |                | linear     |
| a <= b     | convertible to bool | !(a > b) |                | linear     |
| a >= b     | convertible to bool | !(a < b) |                | linear     |

Notes: the algorithms swap(), equal() and lexicographical_compare() are defined in clause 25. Those entries marked “(Note A)” should have constant complexity.
The member function `size()` returns the number of elements in the container. Its semantics is defined by the rules of constructors, inserts, and erases.

`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();`

In the expressions

```
    i == j
    i != j
    i < j
    i <= j
    i > j
    i >= j
    i - j
```

where `i` and `j` denote objects of a container’s iterator type, either or both may be replaced by an object of the container’s `const_iterator` type referring to the same element with no change in semantics.

Copy constructors for all container types defined in this clause copy an allocator argument from their respective first parameters. All other constructors for these container types take an `Allocator&` argument (20.1.2), an allocator whose value type is the same as the container’s value type. A copy of this argument is used for any memory allocation performed, by these constructors and by all member functions, during the lifetime of each container object. In all container types defined in this clause, the member `get_allocator()` returns a copy of the Allocator object used to construct the container.

If the iterator type of a container belongs to the bidirectional or random access iterator categories (24.1), the container is called *reversible* and satisfies the additional requirements in Table 88.

---

Table 88: Reversible container requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>assertion/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::reverse_iterator</td>
<td>iterator type pointing to T</td>
<td>reverse_iterator&lt;iterator&gt;</td>
<td>compile</td>
</tr>
<tr>
<td>X::const_reverse_iterator</td>
<td>iterator type pointing to const T</td>
<td>reverse_iterator&lt;const_reverse_iterator&gt;</td>
<td>compile</td>
</tr>
<tr>
<td>a.rbegin()</td>
<td>reverse_iterator; const_reverse_iterator for constant a</td>
<td>reverse_iterator(end())</td>
<td>constant</td>
</tr>
<tr>
<td>a.rend()</td>
<td>reverse_iterator; const_reverse_iterator for constant a</td>
<td>reverse_iterator(begin())</td>
<td>constant</td>
</tr>
<tr>
<td>a.crbegin();</td>
<td>const_reverse_iterator</td>
<td>const_cast&lt;X const&amp;&gt;(a).rbegin();</td>
<td>constant</td>
</tr>
</tbody>
</table>

---

257) As specified in 20.1.2, paragraphs 4-5, the semantics described in this clause applies only to the case where allocators compare equal.
23.1 Container requirements

Unless otherwise specified (see 23.2.2.3 and 23.2.5.4) all container types defined in this clause meet the following additional requirements:

- if an exception is thrown by an `insert()` function while inserting a single element, that function has no effects.
- if an exception is thrown by a `push_back()` or `push_front()` function, that function has no effects.
- no `erase()`, `pop_back()` or `pop_front()` function throws an exception.
- no copy constructor or assignment operator of a returned iterator throws an exception.
- no `swap()` function throws an exception unless that exception is thrown by the copy constructor or assignment operator of the container’s Compare object (if any; see 23.1.2).
- no `swap()` function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped.

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.

Objects passed to member functions of a container as rvalue references shall not be elements of that container. No diagnostic required.

23.1.1 Sequence containers

A sequence container organizes a finite set of objects, all of the same type, into a strictly linear arrangement. The library provides three basic kinds of sequence containers: `vector`, `list`, and `deque`. It 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).

`vector`, `list`, and `deque` offer the programmer different complexity trade-offs and should be used accordingly. `vector` is the type of sequence container that should be used by default. `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.

In Tables 89 and 90, `X` denotes a sequence container class, `a` denotes a value of `X`, `i` and `j` denote iterators satisfying input iterator requirements and refer to elements implicitly convertible to `value_type`, `[i, j)` denotes a valid range, `n` denotes a value of `X::size_type`, `p` denotes a valid `const` iterator to `a`, `q` denotes 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; `arga` denotes a function parameter pack with the pattern `Arga&&`.

The complexities of the expressions are sequence dependent.
### 23.1 Container requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X(n, t) )</td>
<td></td>
<td>Requires: ( T ) shall be CopyConstructible. post: ( \text{size}() == n ) constructs a sequence container with ( n ) copies of ( t )</td>
</tr>
<tr>
<td>( X a(n, t) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( X(i, j) )</td>
<td>iterator</td>
<td>Requires: If the iterator’s dereference operation returns an lvalue or a const rvalue, ( T ) shall be CopyConstructible. Each iterator in the range ( [i, j) ) shall be dereferenced exactly once. post: ( \text{size}() == \text{distance between } i \text{ and } j ) constructs a sequence container equal to the range ( [i, j) )</td>
</tr>
<tr>
<td>( X a(i, j) )</td>
<td>iterator</td>
<td></td>
</tr>
<tr>
<td>( a.\text{emplace}(p, \text{args}); )</td>
<td>iterator</td>
<td>Inserts an object of type ( T ) constructed with ( T(\text{std::forward&lt;Args&gt;(args)...)}) ).</td>
</tr>
<tr>
<td>( a.\text{insert}(p,t) )</td>
<td>iterator</td>
<td>Requires: ( T ) shall be CopyConstructible and CopyAssignable. inserts a copy of ( t ) before ( p ).</td>
</tr>
<tr>
<td>( a.\text{insert}(p,rv) )</td>
<td>iterator</td>
<td>requires: ( T ) shall be CopyConstructible and CopyAssignable. inserts a copy of ( rv ) before ( p ).</td>
</tr>
<tr>
<td>( a.\text{insert}(p,n,t) )</td>
<td>void</td>
<td>Requires: ( T ) shall be CopyConstructible and CopyAssignable. inserts ( n ) copies of ( t ) before ( p ).</td>
</tr>
<tr>
<td>( a.\text{insert}(p,i,j) )</td>
<td>void</td>
<td>Requires: If the iterator’s dereference operation returns an lvalue or a const rvalue, ( T ) shall be CopyConstructible. Each iterator in the range ( [i, j) ) shall be dereferenced exactly once. pre: ( i ) and ( j ) are not iterators into ( a ). inserts copies of elements in ( [i, j) ) before ( p ).</td>
</tr>
<tr>
<td>( a.\text{erase}(q) )</td>
<td>iterator</td>
<td>erases the element pointed to by ( q )</td>
</tr>
<tr>
<td>( a.\text{erase}(q1,q2) )</td>
<td>iterator</td>
<td>erases the elements in the range ( [q1, q2) ).</td>
</tr>
<tr>
<td>( a.\text{clear}() )</td>
<td>void</td>
<td>( \text{erase}() ), ( \text{end}() ) post: ( \text{size}() == 0 )</td>
</tr>
<tr>
<td>( a.\text{assign}(i,j) )</td>
<td>void</td>
<td>Requires: If the iterator’s dereference operation returns an lvalue or a const rvalue, ( T ) shall be CopyConstructible and CopyAssignable. Each iterator in the range ( [i, j) ) shall be dereferenced exactly once. pre: ( i, j ) are not iterators into ( a ). Replaces elements in ( a ) with a copy of ( [i, j) ).</td>
</tr>
<tr>
<td>( a.\text{assign}(n,t) )</td>
<td>void</td>
<td>Requires: ( T ) shall be CopyConstructible and CopyAssignable. pre: ( t ) is not a reference into ( a ). Replaces elements in ( a ) with ( n ) copies of ( t ).</td>
</tr>
</tbody>
</table>

Draft
iterator and const_iterator types for sequence containers shall be at least of the forward iterator category.

The iterator returned from a.insert(p,t) points to the copy of t inserted 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:

- If the constructor

  template <class InputIterator>
  X(InputIterator first, InputIterator last,
      const allocator_type& alloc = allocator_type())

  is called with a type InputIterator that does not qualify as an input iterator, then the constructor will behave as if the overloaded constructor:

  X(size_type, const value_type& = value_type(),
     const allocator_type& = allocator_type())

  were called instead, with the arguments static_cast<size_type>(first), last and alloc, respectively.

- If the member functions of the forms:

  template <class InputIterator> // such as insert()
  rt fx1(iterator p, InputIterator first, InputIterator last);

  template <class InputIterator> // such as append(), assign()
  rt fx2(InputIterator first, InputIterator last);

  template <class InputIterator> // such as replace()
  rt fx3(iterator i1, iterator i2, InputIterator first, InputIterator last);

  are called with a type InputIterator that does not qualify as an input iterator, then these functions will behave as if the overloaded member functions:

  rt fx1(iterator, size_type, const value_type&);

  rt fx2(size_type, const value_type&);

  rt fx3(iterator, iterator, size_type, const value_type&);

  were called instead, with the same arguments.

In the previous paragraph the alternative binding will fail if first is not implicitly convertible to X::size_type or if last is not implicitly convertible to X::value_type.

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 90 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>assertion/note</th>
<th>container</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.front()</td>
<td>reference; const_reference for constant a</td>
<td>*a.begin()</td>
<td>vector, list, deque, basic_string</td>
</tr>
<tr>
<td>a.back()</td>
<td>reference; const_reference for constant a</td>
<td>{ iterator tmp = a.end(); --tmp; return *tmp; }</td>
<td>vector, list, deque, basic_string</td>
</tr>
<tr>
<td>a.push_front(t)</td>
<td>void</td>
<td>a.insert(a.begin(), t). Requires: T shall be CopyConstructible.</td>
<td>list, deque</td>
</tr>
<tr>
<td>a.push_front(rv)</td>
<td>void</td>
<td>a.insert(a.begin(), rv)</td>
<td>list, deque</td>
</tr>
<tr>
<td>a.push_back(t)</td>
<td>void</td>
<td>a.insert(a.end(), t). Requires: T shall be CopyConstructible.</td>
<td>vector, list, deque, basic_string</td>
</tr>
<tr>
<td>a.push_back(rv)</td>
<td>void</td>
<td>a.insert(a.end(), rv)</td>
<td>vector, list, deque, basic_string</td>
</tr>
<tr>
<td>a.push_front(args)</td>
<td>void</td>
<td>aemplace(a.begin(), std::forward&lt;Args&gt;(args)...</td>
<td>list, deque</td>
</tr>
<tr>
<td>a.push_back(args)</td>
<td>void</td>
<td>aemplace(a.end(), std::forward&lt;Args&gt;(args)...</td>
<td>list, deque, vector</td>
</tr>
<tr>
<td>a.pop_front()</td>
<td>void</td>
<td>a.erase(a.begin())</td>
<td>list, deque</td>
</tr>
<tr>
<td>a.pop_back()</td>
<td>void</td>
<td>{ iterator tmp = a.end(); --tmp; a.erase(tmp); }</td>
<td>vector, list, deque, basic_string</td>
</tr>
<tr>
<td>a[n]</td>
<td>reference; const_reference for constant a</td>
<td>*(a.begin() + n)</td>
<td>vector, deque, basic_string</td>
</tr>
<tr>
<td>a.at(n)</td>
<td>reference; const_reference for constant a</td>
<td>*(a.begin() + n)</td>
<td>vector, deque</td>
</tr>
</tbody>
</table>

The member function `at()` provides bounds-checked access to container elements. `at()` throws `out_of_range` if `n >= a.size()`.

### 23.1.2 Associative containers

Associative containers provide an ability for fast retrieval of data based on keys. The library provides four basic kinds...
of associative containers: `set`, `multiset`, `map` and `multimap`.

2 Each associative container is parameterized on `Key` and an ordering relation `Compare` that induces a strict weak ordering (25.3) on elements of `Key`. In addition, `map` and `multimap` associate an arbitrary type `T` with the `Key`. The object of type `Compare` is called the *comparison object* of a container. This comparison object may be a pointer to function or an object of a type with an appropriate function call operator.

3 The phrase “equivalence of keys” means the equivalence relation imposed by the comparison and not the `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.

4 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` and `erase` preserve the relative ordering of equivalent elements.

5 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>`. Keys in an associative container are immutable.

6 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.

7 In Table 91, `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, `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`, `[q1,q2]` denotes a valid range of `const iterators` in `a`, `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`.

Table 91: 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><code>X::key_type</code></td>
<td>Key</td>
<td>Key is CopyConstructible and CopyAssignable</td>
<td>compile time</td>
</tr>
<tr>
<td><code>X::key_compare</code></td>
<td>Compare</td>
<td>defaults to <code>less&lt;Key&gt;</code></td>
<td>compile time</td>
</tr>
<tr>
<td><code>X::value_compare</code></td>
<td>a binary predicate type</td>
<td>is the same as <code>key_compare</code> for <code>set</code> and <code>multiset</code>; is an ordering relation on pairs induced by the first component (<em>i.e.</em> <code>Key</code>) for <code>map</code> and <code>multimap</code>.</td>
<td>compile time</td>
</tr>
<tr>
<td><code>X(c)</code></td>
<td>constructs an empty container uses <code>c</code> as a comparison object</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td><code>X(a(c);</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>X()</code></td>
<td>constructs an empty container uses <code>Compare()</code> as a comparison object</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td><code>X a;</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>expression</td>
<td>return type</td>
<td>assertion/note</td>
<td>complexity</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
<td>----------------</td>
<td>-----------------</td>
</tr>
<tr>
<td><code>X(i,j,c)</code></td>
<td></td>
<td>constructs an empty container and inserts elements from the range <code>[i, j)</code> into it; uses <code>c</code> as a comparison object.</td>
<td><code>N \log N</code> in general (<code>N</code> is the distance from <code>i</code> to <code>j</code>); linear if <code>[i, j)</code> is sorted with <code>value_compare()</code></td>
</tr>
<tr>
<td><code>X a(i,j,c);</code></td>
<td></td>
<td>same as above, but uses <code>Compare()</code> as a comparison object</td>
<td>same as above</td>
</tr>
<tr>
<td><code>X(i,j)</code></td>
<td></td>
<td>returns the comparison object out of which <code>a</code> was constructed.</td>
<td>constant</td>
</tr>
<tr>
<td><code>X a(i,j);</code></td>
<td></td>
<td>returns an object of <code>value_compare</code> constructed out of the comparison object</td>
<td>constant</td>
</tr>
<tr>
<td><code>a.key_-</code></td>
<td><code>X::key_-</code></td>
<td>inserts a <code>T</code> object <code>t</code> constructed with <code>std::forward&lt;Args&gt;(args)</code>... if and only if there is no element in the container with key equivalent to the key of <code>t</code>. The <code>bool</code> component of the returned pair is 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 <code>t</code>.</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>comp()</code></td>
<td><code>compare</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a.value_-</code></td>
<td><code>X::value_-</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>comp()</code></td>
<td><code>compare</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a_uniq.</code></td>
<td><code>pair&lt;iterator, bool&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>emplace(args)</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a_eq.</code></td>
<td><code>iterator</code></td>
<td>inserts a <code>T</code> object <code>t</code> constructed with <code>std::forward&lt;Args&gt;(args)</code>... and returns the iterator pointing to the newly inserted element.</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>emplace(args)</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a.</code></td>
<td><code>iterator</code></td>
<td>equivalent to <code>a.emplace(std::forward&lt;Args&gt;(args))</code>. Return value is an iterator pointing to the element with the key equivalent to the newly inserted element. The <code>const_iterator p</code> is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.</td>
<td>logarithmic in general, but amortized constant if the element is inserted right after <code>r</code></td>
</tr>
<tr>
<td><code>emplace(p, args)</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>expression</td>
<td>return type</td>
<td>assertion/note pre/post-condition</td>
<td>complexity</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------</td>
<td>---------------------------------------------------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>a uniq. insert(t)</td>
<td>pair&lt;iterator, bool&gt;</td>
<td>inserts t if and only if there is no element in the container with key equivalent to the key of t. The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of t.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>a_eq.insert(t)</td>
<td>iterator</td>
<td>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>inserts t if and only if there is no element with key equivalent to the key of t in containers with unique keys; always inserts t in containers with equivalent keys. always returns the iterator pointing to the element with key equivalent to the key of t. t is inserted as close as possible to the position just prior to p.</td>
<td>logarithmic in general, but amortized constant if t is inserted right before p.</td>
</tr>
<tr>
<td>a.insert(i, j)</td>
<td>void</td>
<td>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(size()) + N ) ( (N ) is the distance from i to j)</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(size()) + \text{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>
</tbody>
</table>
### Container requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>assertion/note</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.erase(q1, q2)</td>
<td>iterator</td>
<td>erases all the elements in the range [q1, q2]. Returns q2.</td>
<td>( \log(size()) + N ) where ( N ) is the distance from q1 to q2.</td>
</tr>
<tr>
<td>a.clear()</td>
<td>void</td>
<td>erase(a.begin(), a.end()) post: size() == 0</td>
<td>linear in size().</td>
</tr>
<tr>
<td>a.find(k)</td>
<td>iterator;</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></td>
<td>const_iterator for constant a.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a.count(k)</td>
<td>size_type</td>
<td>returns the number of elements with key equivalent to k</td>
<td>( \log(size()) + \text{count}(k) )</td>
</tr>
<tr>
<td>a.lower_bound(k)</td>
<td>iterator;</td>
<td>returns an iterator pointing to the first element with key not less than k, or a.end() if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td></td>
<td>const_iterator for constant a.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a.upper_bound(k)</td>
<td>iterator;</td>
<td>returns an iterator pointing to the first element with key greater than k, or a.end() if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td></td>
<td>const_iterator for constant a.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a.equal_range(k)</td>
<td>pair&lt;iterator, iterator&gt;; pair&lt;const_iterator, const_iterator&gt; for constant a.</td>
<td>equivalent to make_&lt;pair&lt;a.lower_bound(k), a.upper_bound(k)&gt;</td>
<td>logarithmic</td>
</tr>
</tbody>
</table>

8 The insert 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.

9 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,

\[
\text{value_comp}(*j, *i) = \text{false}
\]

10 For associative containers with unique keys the stronger condition holds,

\[
\text{value_comp}(*i, *j) \neq \text{false}.
\]

11 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

Draft
containers library

23.1 Container requirements

23.1.3 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 (23.1), except that the expressions in table 92 are not required to be valid, where a and b denote values of a type X, and X is an unordered associative container class:

Table 92: Container requirements that are not required for unordered associative containers

<table>
<thead>
<tr>
<th>unsupported expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>a == b</td>
</tr>
<tr>
<td>a != b</td>
</tr>
<tr>
<td>a &lt; b</td>
</tr>
<tr>
<td>a &gt; b</td>
</tr>
<tr>
<td>a &lt;= b</td>
</tr>
<tr>
<td>a &gt;= b</td>
</tr>
</tbody>
</table>

Each unordered associative container is parameterized by Key, by a function object Hash that acts as a hash function for 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.

A hash function is a function object that takes a single argument of type Key and returns a value of type std::size_t.

Two values k1 and k2 of type Key are considered equal if the container’s equality function object returns true when passed those values. If k1 and k2 are equal, the hash function shall return the same value for both.

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.

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>.

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.

In table 93: 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 \( q_1 \) are valid dereferenceable \texttt{const} iterators to \( a \), \([q_1, q_2)\) is a valid range in \( a \), \( r \) and \( r_1 \) are valid dereferenceable \texttt{const} iterators to \( a \), \([r_1, r_2)\) is a valid range in \( a \), \( t \) is a value of type \( X::\text{value}{}_{-}{}_{-}\text{type} \), \( k \) is a value of type \( \text{key}{}_{-}{}_{-}\text{type} \), \( hf \) is a possibly \texttt{const} value of type \( \text{hasher} \), \( eq \) is a possibly \texttt{const} value of type \( \text{key}{}_{-}{}_{-}\text{equal} \), \( n \) is a value of type \( \text{size}{}_{-}{}_{-}\text{type} \), and \( z \) is a value of type \( \text{float} \).

Table 93: Unordered associative container requirements (in addition to container)

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>assertion/note pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X::\text{key}{}<em>{-}{}</em>{-}\text{type} )</td>
<td>Key</td>
<td>Key shall be \texttt{CopyAssignable} and \texttt{CopyConstructible}</td>
<td>compile time</td>
</tr>
<tr>
<td>( X::\text{hasher} )</td>
<td>Hash</td>
<td>Hash shall be a unary function object type such that the expression ( hf(k) ) has type ( \text{std}::\text{size}{}<em>{-}{}</em>{-}\text{t} ).</td>
<td>compile time</td>
</tr>
<tr>
<td>( X::\text{key}{}<em>{-}{}</em>{-}\text{equal} )</td>
<td>Pred</td>
<td>Pred shall be a binary predicate that takes two arguments of type ( \text{Key} ). Pred is an equivalence relation.</td>
<td>compile time</td>
</tr>
<tr>
<td>( X::\text{local}{}<em>{-}{}</em>{-}\text{iterator} )</td>
<td>An iterator type whose category, value type, difference type, and pointer and reference types are the same as ( X::\text{iterator} )'s.</td>
<td>A \texttt{local}{}<em>{-}{}</em>{-}\text{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::\text{const}{}<em>{-}{}</em>{-}\text{local}{}<em>{-}{}</em>{-}\text{iterator} )</td>
<td>An iterator type whose category, value type, difference type, and pointer and reference types are the same as ( X::\text{const}{}<em>{-}{}</em>{-}\text{iterator} )'s.</td>
<td>A \texttt{const}{}<em>{-}{}</em>{-}\text{local}{}<em>{-}{}</em>{-}\text{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>Constructs an empty container with at least ( n ) buckets, using ( hf ) as the hash function and ( eq ) as the key equality predicate.</td>
<td>( \mathcal{O}(n) )</td>
</tr>
<tr>
<td>( X(n, hf, eq) )</td>
<td>( X )</td>
<td>Constructs an empty container with at least ( n ) buckets, using ( hf ) as the hash function and ( \text{key}{}<em>{-}{}</em>{-}\text{equal}() ) as the key equality predicate.</td>
<td>( \mathcal{O}(n) )</td>
</tr>
<tr>
<td>( X(n, hf) )</td>
<td>( X )</td>
<td>Constructs an empty container with at least ( n ) buckets, using ( \text{hasher}() ) as the hash function and ( \text{key}{}<em>{-}{}</em>{-}\text{equal}() ) as the key equality predicate.</td>
<td>( \mathcal{O}(n) )</td>
</tr>
<tr>
<td>( X(n) )</td>
<td>( X )</td>
<td>Constructs an empty container with at least ( n ) buckets, using ( \text{hasher}() ) as the hash function and ( \text{key}{}<em>{-}{}</em>{-}\text{equal}() ) as the key equality predicate.</td>
<td>( \mathcal{O}(n) )</td>
</tr>
<tr>
<td>expression</td>
<td>return type</td>
<td>assertion/note pre/post-condition</td>
<td>complexity</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>-----------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>X()</td>
<td>X</td>
<td>Constructs an empty container with an unspecified number of buckets, using hasher() as the hash function and key_equal as the key equality predicate.</td>
<td>constant</td>
</tr>
<tr>
<td>X a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(i, j, n, hf, eq)</td>
<td>X</td>
<td>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 $\Theta(N)$ (N is distance(i, j)), worst case $\Theta(N^2)$</td>
</tr>
<tr>
<td>X a(i, j, n, hf, eq)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(i, j, n, hf)</td>
<td>X</td>
<td>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 $\Theta(N)$ (N is distance(i, j)), worst case $\Theta(N^2)$</td>
</tr>
<tr>
<td>X a(i, j, n, hf)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(i, j, n)</td>
<td>X</td>
<td>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>X a(i, j, n)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(i, j)</td>
<td>X</td>
<td>Constructs an empty container with an unspecified number of buckets, using hasher() as the hash function and key_equal as the key equality predicate, and inserts elements from [i, j) into it.</td>
<td>Average case $\Theta(N)$ (N is distance(i, j)), worst case $\Theta(N^2)$</td>
</tr>
<tr>
<td>X a(i, j)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(b)</td>
<td>X</td>
<td>Copy constructor. In addition to the contained elements, copies the hash function, predicate, and maximum load factor.</td>
<td>Average case linear in b.size(), worst case quadratic.</td>
</tr>
<tr>
<td>X a(b)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a = b</td>
<td>X</td>
<td>Copy assignment operator. In addition to the contained elements, copies the hash function, predicate, and maximum load factor.</td>
<td>Average case linear in b.size(), worst case quadratic.</td>
</tr>
<tr>
<td>b.hash_function()</td>
<td>hasher</td>
<td>Returns b’s hash function.</td>
<td>constant</td>
</tr>
<tr>
<td>b.key_eq()</td>
<td>key_equal</td>
<td>Returns b’s key equality predicate.</td>
<td>constant</td>
</tr>
</tbody>
</table>
### 23.1 Container requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>assertion/note pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_uniq_.emplace(args)</td>
<td>pair&lt;iterator, bool&gt;</td>
<td>inserts a <code>T</code> object <code>t</code> constructed with <code>std::forward&lt;Args&gt;(args)</code>... if and only if there is no element in the container with key equivalent to the key of <code>t</code>. The 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 <code>t</code>.</td>
<td>Average case ( \Theta(1) ), worst case ( \Theta(a_uniq_.size()) ).</td>
</tr>
<tr>
<td>a_eq.emplace(args)</td>
<td>iterator</td>
<td>inserts a <code>T</code> object <code>t</code> constructed with <code>std::forward&lt;Args&gt;(args)</code>... and returns the iterator pointing to the newly inserted element.</td>
<td>Average case ( \Theta(1) ), worst case ( \Theta(a_eq_.size()) ).</td>
</tr>
<tr>
<td>a_.emplace(p, args)</td>
<td>iterator</td>
<td>equivalent to a_.emplace(<code>std::forward&lt;Args&gt;(args)</code>...). Return value is an iterator pointing to the element with the key equivalent to the newly inserted element. The const iterator <code>p</code> is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.</td>
<td>Average case ( \Theta(1) ), worst case ( \Theta(a_.size()) ).</td>
</tr>
<tr>
<td>a_uniq.insert(t)</td>
<td>pair&lt;iterator, bool&gt;</td>
<td>Inserts <code>t</code> if and only if there is no element in the container with key equivalent to the key of <code>t</code>. The 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 <code>t</code>.</td>
<td>Average case ( \Theta(1) ), worst case ( \Theta(a_uniq_.size()) ).</td>
</tr>
<tr>
<td>a_eq.insert(t)</td>
<td>iterator</td>
<td>Inserts <code>t</code>, and returns an iterator pointing to the newly inserted element.</td>
<td>Average case ( \Theta(1) ), worst case ( \Theta(a_eq_.size()) ).</td>
</tr>
<tr>
<td>a.insert(q, t)</td>
<td>iterator</td>
<td>Equivalent to a.insert(t). 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 ( \Theta(1) ), worst case ( \Theta(a_.size()) ).</td>
</tr>
<tr>
<td>expression</td>
<td>return type</td>
<td>assertion/note pre/post-condition</td>
<td>complexity</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>a.insert(r, t)</td>
<td>const_iterator</td>
<td>Equivalent to a.insert(t). Return value is an iterator pointing to the element with the key equivalent to that of t. The iterator r 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.insert(i, j)</td>
<td>void</td>
<td>Pre: i and j are not iterators in a. Equivalent to a.insert(t) for each element in ([i, j)).</td>
<td>Average case ( \Theta(N) ), where ( N ) is distance(i, j). Worst case ( \Theta(N \cdot a.size()) ).</td>
</tr>
<tr>
<td>a.erase(k)</td>
<td>size_type</td>
<td>Erases all elements with key equivalent to k. Returns the number of elements erased.</td>
<td>Average case ( \Theta(a.count(k)) ). Worst case ( \Theta(a.size()) ).</td>
</tr>
<tr>
<td>a.erase(q)</td>
<td>iterator</td>
<td>Erases the element pointed to by q. Return value is the iterator immediately following q prior to the erasure.</td>
<td>Average case ( \Theta(1) ), worst case ( \Theta(a.size()) ).</td>
</tr>
<tr>
<td>a.erase(r)</td>
<td>const_iterator</td>
<td>Erases the element pointed to by r. Return value is the iterator immediately following r prior to the erasure.</td>
<td>Average case ( O(1) ), worst case ( O(a.size()) ).</td>
</tr>
<tr>
<td>a.erase(q1, q2)</td>
<td>iterator</td>
<td>Erases all elements in the range ([q1, q2)). Return value is the iterator immediately following the erased elements prior to the erasure.</td>
<td>Average case linear in distance(q1, q2), worst case ( \Theta(a.size()) ).</td>
</tr>
<tr>
<td>a.erase(r1, r2)</td>
<td>const_iterator</td>
<td>Erases all elements in the range ([r1, r2)). Return value is the iterator immediately following the erased elements prior to the erasure.</td>
<td>Average case linear in distance(r1, r2), worst case ( O(a.size()) ).</td>
</tr>
<tr>
<td>a.clear()</td>
<td>void</td>
<td>Erases all elements in the container. Post: a.size() == 0.</td>
<td>Linear.</td>
</tr>
<tr>
<td>b.find(k)</td>
<td>iterator; const_iterator for const b.</td>
<td>Returns an iterator pointing to an element with key equivalent to k, or b.end() if no such element exists.</td>
<td>Average case ( \Theta(1) ), worst case ( \Theta(b.size()) ).</td>
</tr>
<tr>
<td>b.count(k)</td>
<td>size_type</td>
<td>Returns the number of elements with key equivalent to k.</td>
<td>Average case ( \Theta(1) ), worst case ( \Theta(b.size()) ).</td>
</tr>
</tbody>
</table>
### Container requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>assertion/note pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>b.equal_range(k)</td>
<td>pair&lt;iterator, iterator&gt;,</td>
<td>Returns a range containing all</td>
<td>Average case</td>
</tr>
<tr>
<td></td>
<td>pair&lt;const_iterator, const_iterator&gt;for const b.</td>
<td>elements with keys equivalent to k.</td>
<td>(\mathcal{O}(b.\text{count}(k))).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns make_pair(b.end(), b.end()) if no such elements exist.</td>
<td>(\mathcal{O}(b.\text{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>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.\text{bucket}_\text{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.\text{bucket}_\text{count}())). Returns the number of elements in the (n)th bucket.</td>
<td>(\mathcal{O}(b.\text{bucket}_\text{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.\text{bucket}_\text{count}())). Note: ([b.\text{begin}(n), b.\text{end}(n))) is a valid range containing all of the elements in the (n)th bucket.</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.\text{bucket}_\text{count}())).</td>
<td>Constant</td>
</tr>
<tr>
<td>b.load_factor()</td>
<td>float</td>
<td>Returns the average number of elements per bucket.</td>
<td>Constant</td>
</tr>
<tr>
<td>b.max_load_factor()</td>
<td>float</td>
<td>Returns a positive number that the container attempts to keep the load factor less than or equal to. The container automatically increases the number of buckets as necessary to keep the load factor below this number.</td>
<td>Constant</td>
</tr>
<tr>
<td>a.max_load_factor(z)</td>
<td>void</td>
<td>Pre: (z) shall be positive. Changes the container's maximum load load factor, using (z) as a hint.</td>
<td>Constant</td>
</tr>
<tr>
<td>a.rehash(n)</td>
<td>void</td>
<td>Post: a.\text{bucket}<em>\text{count}() &gt; a.\text{size}() / a.\text{max}</em>\text{load}<em>\text{factor}() and a.\text{bucket}</em>\text{count}() &gt;= n.</td>
<td>Average case linear in a.\text{size}(), worst case quadratic.</td>
</tr>
</tbody>
</table>

Draft
Unordered associative containers are not required to support the expressions \( a == b \) or \( a != b \). \[^{Note}\] This is because the container requirements define operator equality in terms of equality of ranges. Since the elements of an unordered associative container appear in an arbitrary order, range equality is not a useful operation. \[^{- end note -}\]

The iterator types \texttt{iterator} and \texttt{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 \texttt{iterator} and \texttt{const_iterator} are const iterators.

The insert 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.

The insert members shall not affect the validity of iterators if \((N+n) < z \cdot B\), where \(N\) is the number of elements in the container prior to the insert operation, \(n\) is the number of elements inserted, \(B\) is the container’s bucket count, and \(z\) is the container’s maximum load factor.

23.1.3.1 Exception safety guarantees \[^{[unord.req.except]}\]

1. For unordered associative containers, no \texttt{clear()} function throws an exception. No \texttt{erase()} function throws an exception unless that exception is thrown by the container’s Hash or Pred object (if any).
2. For unordered associative containers, if an exception is thrown by any operation other than the container’s hash function from within an \texttt{insert()} function inserting a single element, the \texttt{insert()} function has no effect.
3. For unordered associative containers, no \texttt{swap} function throws an exception unless that exception is thrown by the copy constructor or copy assignment operator of the container’s Hash or Pred object (if any).
4. For unordered associative containers, if an exception is thrown from within a \texttt{rehash()} function other than by the container’s hash function or comparison function, the \texttt{rehash()} function has no effect.

23.2 Sequence containers \[^{[sequences]}\]

Header \texttt{<array> synopsis}

\begin{verbatim}
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);
}
\end{verbatim}
bool operator>=
    (const array<T,N>& x, const array<T,N>& y);
void swap(array<T,N>& x, array<T,N>& y);

template <class T>
    class tuple_size; // forward declaration

template <int I, class T>
    struct tuple_element; // forward declaration

template <class T, size_t N>
    struct tuple_size<array<T, N> >;

template <int I, class T, size_t N>
    struct tuple_element<I, array<T, N> >;

T& get(array<T, N>&);
const T& get(const array<T, N>&);

} // namespace std

Header <deque> synopsis

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);

void swap(deque<T,Allocator>& x, deque<T,Allocator>& y);
void swap(deque<T,Allocator>&& x, deque<T,Allocator>& y);
void swap(deque<T,Allocator>& x, deque<T,Allocator>&& y);
}

Header <list> synopsis

namespace std {
    template <class T, class Allocator = allocator<T> >
        class list;

} // namespace std
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);

namespace std {
    template <class T, class Container = deque<T> > class queue;
    template <class T, class Container>
    bool operator==(const queue<T, Container>& x, const queue<T, Container>& y);
    template <class T, class Container>
    bool operator< (const queue<T, Container>& x, const queue<T, Container>& y);
    template <class T, class Container>
    bool operator!=(const queue<T, Container>& x, const queue<T, Container>& y);
    template <class T, class Container>
    bool operator> (const queue<T, Container>& x, const queue<T, Container>& y);
    template <class T, class Container>
    bool operator>=(const queue<T, Container>& x, const queue<T, Container>& y);
    template <class T, class Container>
    bool operator<=(const queue<T, Container>& x, const queue<T, Container>& y);
    template <class T, class Allocator>
    void swap(queue<T,Allocator>& x, queue<T,Allocator>& y);
    template <class T, class Allocator>
    void swap(queue<T,Allocator>&& x, queue<T,Allocator>& y);
    template <class T, class Allocator>
    void swap(queue<T,Allocator>& x, queue<T,Allocator>&& y);
    template <class T, class Container = vector<T>>,</p>

Draft
23.2 Sequence containers

Containers library

```cpp
class Compare = less<typename Container::value_type >
class priority_queue;
template <class T, class Allocator>
void swap(priority_queue<T,Allocator>& x, priority_queue<T,Allocator>& y);
template <class T, class Allocator>
void swap(priority_queue<T,Allocator>&& x, priority_queue<T,Allocator>& y);
template <class T, class Allocator>
void swap(priority_queue<T,Allocator>& x, priority_queue<T,Allocator>&& y);
}
```

**Header <stack> synopsis**

```cpp
namespace std {
template <class T, class Container = deque<T> > class stack;
template <class T, class Container>
bool operator==(const stack<T, Container>& x, const stack<T, Container>& y);
template <class T, class Container>
bool operator<(const stack<T, Container>& x, const stack<T, Container>& y);
template <class T, class Container>
bool operator!=(const stack<T, Container>& x, const stack<T, Container>& y);
template <class T, class Container>
bool operator>(const stack<T, Container>& x, const stack<T, Container>& y);
template <class T, class Container>
bool operator>=(const stack<T, Container>& x, const stack<T, Container>& y);
template <class T, class Container>
bool operator<=(const stack<T, Container>& x, const stack<T, Container>& y);
template <class T, class Allocator>
void swap(stack<T,Allocator>& x, stack<T,Allocator>& y);
template <class T, class Allocator>
void swap(stack<T,Allocator>&& x, stack<T,Allocator>& y);
template <class T, class Allocator>
void swap(stack<T,Allocator>& x, stack<T,Allocator>&& y);
}
```

**Header <vector> synopsis**

```cpp
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);
template <class T, class Allocator>
void swap(vector<T,Allocator>&& x, vector<T,Allocator>& y);
template <class T, class Allocator>
void swap(vector<T,Allocator>& x, vector<T,Allocator>&& y);
}
```

Draft
```cpp
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);

template <class T, class Allocator>
void swap(vector<T, Allocator>&& x, vector<T, Allocator>& y);

template <class T, class Allocator>
void swap(vector<T, Allocator>& x, vector<T, Allocator>&& y);

23.2.1 Class template array

The header <array> defines a class template for storing fixed-size sequences of objects. An array supports random access iterators. An instance of array<T, N> stores N elements of type T, so that size() == N is an invariant. The elements of an array are stored contiguously, meaning that if a is an array<T, N> then it obeys the identity &a[n] == &a[0] + n for all 0 <= n < N.

An array is an aggregate (8.5.1) that can be initialized with the syntax

```cpp
array a = { initializer-list };  
```

where initializer-list is a comma separated list of up to N elements whose types are convertible to T.

Unless otherwise specified, all array operations are as described in 23.1. Descriptions are provided here only for operations on array that are not described in that clause or 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 std::reverse_iterator<iterator> reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

        T elems[N];  // Exposition only
    }
}
```

Draft
// No explicit construct/copy/destroy for aggregate type

void assign(const T& u);
void swap(array<T, N>&);

// iterators:
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// capacity:
constexpr size_type size() const;
constexpr size_type max_size() const;
bool empty() const;

// 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;

T * data();
const T * data() const;
}
functions (12.1, 12.4, and 12.8) to conform to the container requirements table in 23.1.

### 23.2.1.2 array specialized algorithms

```cpp
template <class T, size_t N> void swap(array<T,N>& x, array<T,N>& y);
```

**Effects**

```cpp
swap_ranges(x.begin(), x.end(), y.begin());
```

### 23.2.1.3 array::size

```cpp
template <class T, size_t N> size_type array<T,N>::size();
```

**Returns:**

```
N
```

### 23.2.1.4 array::data

```cpp
T *data();
const T *data() const;
```

**Returns:**

```
elems.
```

### 23.2.1.5 Zero sized arrays

1. array shall provide support for the special case \( N = 0 \).
2. In the case that \( N = 0 \), \( \text{begin()} == \text{end()} == \text{unique value} \). The return value of \( \text{data()} \) is unspecified.
3. The effect of calling \( \text{front()} \) or \( \text{back()} \) for a zero-sized array is implementation defined.

### 23.2.1.6 Tuple interface to class template array

```cpp
tuple_size<array<T, N> >::value
```

**Return type:** integral constant expression.

```cpp
tuple_element<I, array<T, N> >::type
```

**Requires:** \( 0 <= I < N \). The program is ill-formed if \( I \) is out of bounds.

**Value:** The type \( T \).

```cpp
template <int I, class T, size_t N> T& get(array<T,N>& a);
```

**Requires:** \( 0 <= I < N \). The program is ill-formed if \( I \) is out of bounds.

**Returns:** A reference to the \( I \)th element of \( a \), where indexing is zero-based.

```cpp
template <int I, class T, size_t N> const T& get(const array<T,N>& a);
```
23.2 Sequence containers

6 Requires: 0 <= 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.

23.2.2 Class template deque

A deque is a sequence container that, like a vector (23.2.5), 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.

A deque satisfies all of the requirements of a container and of a reversible container (given in tables in 23.1) and of a sequence container, including the optional sequence container requirements (23.1.1). 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 typename Allocator::reference             reference;
            typedef typename Allocator::const_reference       const_reference;
            typedef implementation-defined                     iterator; // See 23.1
            typedef implementation-defined                     const_iterator; // See 23.1
            typedef implementation-defined                     size_type; // See 23.1
            typedef implementation-defined                     difference_type; // See 23.1
            typedef T                                          value_type;
            typedef Allocator                                  allocator_type;
            typedef typename Allocator::pointer               pointer;
            typedef typename Allocator::const_pointer         const_pointer;
            typedef std::reverse_iterator<iterator>           reverse_iterator;
            typedef std::reverse_iterator<const_iterator>     const_reverse_iterator;

            // 23.2.2.1 construct/copy/destroy:
            explicit deque(const Allocator& = Allocator());
            explicit deque(size_type n);
            deque(size_type n, const T& value,
                const Allocator& = Allocator());
            template <class InputIterator>
                deque(InputIterator first, InputIterator last,
                    const Allocator& = Allocator());
            deque(const deque<T,Allocator>& x);
            deque(T,Allocator>& x);
            ~deque();
            deque<T,Allocator>& operator=(const deque<T,Allocator>& x);
            deque operator=(const deque<T,Allocator>& x);
            template <class InputIterator>
                void assign(InputIterator first, InputIterator last);
            void assign(size_type n, const T& t);

            Draft
allocator_type get_allocator() const;

// iterators:
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// 23.2.2.2 capacity:
size_type size() const;
size_type max_size() const;
void resize(size_type sz);
void resize(size_type sz, T c);
bool empty() const;

// 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;

// 23.2.2.3 modifiers:
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
template <class... Args> void push_front(Args&&... args);
template <class... Args> void push_back(Args&&... args);
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);
void insert(const_iterator position, T&& x);
template <class InputIterator>
void insert (const_iterator position,
InputIterator first, InputIterator last);
23.2 Sequence containers

void pop_front();
void pop_back();

iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void swap(deque<T,Allocator>&&);
void clear();
};

template <class T, class Allocator>
bool operator==(const deque<T,Allocator>& x,
const deque<T,Allocator>& y);

// specialized algorithms:

23.2.2.1 deque constructors, copy, and assignment

extric deque(const Allocator& = Allocator());

Effects: Constructs an empty deque, using the specified allocator.

Complexity: Constant.

extric deque(size_type n);

Effects: Constructs a deque with n default constructed elements.

Requires: T shall be DefaultConstructible.

Complexity: Linear in n.
deque(size_type n, const T& value,
    const Allocator& = Allocator());

Effects: Constructs a deque with n copies of value, using the specified allocator.
Requires: T shall be CopyConstructible.
Complexity: Linear in n.

template <class InputIterator>
    deque(InputIterator first, InputIterator last,
    const Allocator& = Allocator());

Effects: Constructs a deque equal to the the range [first, last), using the specified allocator.
Complexity: distance(first, last).

template <class InputIterator>
    void assign(InputIterator first, InputIterator last);

Effects:
    erase(begin(), end());
    insert(begin(), first, last);

void assign(size_type n, const T& t);

Effects:
    erase(begin(), end());
    insert(begin(), n, t);

23.2.2.2  deque capacity

void resize(size_type sz);

Effects: If sz < size(), equivalent to erase(begin() + sz, end()); If size() < sz, appends sz - size() default constructed elements to the sequence.
Requires: T shall be DefaultConstructible.

void resize(size_type sz, T c);

Effects:
    if (sz > size())
        insert(end(), sz-size(), c);
    else if (sz < size())
        erase(begin()+sz, end());
    else
        // do nothing

Requires: T shall be CopyConstructible.
23.2.2.3 deque modifiers

```
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
void insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
void insert(const_iterator position, InputIterator first, InputIterator last);

template <class... Args> void push_front(Args&&... args);

template <class... Args> void push_back(Args&&... args);

template <class... Args> iterator emplace(const_iterator position, Args&&... args);
```

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.

Remarks: If an exception is thrown other than by the copy constructor or assignment operator of T there are no effects.

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 the copy constructor of T.

```
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
```

Effects: An erase in the middle of the deque invalidates all the iterators and references to elements of the deque. An erase at either end of the deque invalidates only the iterators and the references to the erased elements.

Complexity: The number of calls to the destructor is the same as the number of elements erased, but the number of the calls to the assignment operator is at most equal to the minimum of the number of elements before the erased elements and the number of elements after the erased elements.

Throws: Nothing unless an exception is thrown by the copy constructor or assignment operator of T.

23.2.2.4 deque specialized algorithms

```
template <class T, class Allocator>
void swap(deque<T,Allocator>& x, deque<T,Allocator>& y);
template <class T, class Allocator>
void swap(deque<T,Allocator>&& x, deque<T,Allocator>&& y);
template <class T, class Allocator>
void swap(deque<T,Allocator>& x, deque<T,Allocator>&& y);
```

Effects:

```
x.swap(y);
```
### 23.2.3 Class template `list`

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 (23.2.5) and deques (23.2.2), 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 and of a reversible container (given in two tables in 23.1) and of a sequence container, including most of the the optional sequence container requirements (23.1.1). The exceptions are the operator[] and at member functions, which are not provided. Descriptions are provided here only for operations on list that are not described in one of these tables or for operations where there is additional semantic information.

```cpp
namespace std {
    template <class T, class Allocator = allocator<T> >
    class list {
    public:
        // types:
        typedef typename Allocator::reference       reference;
        typedef typename Allocator::const_reference const_reference;
        typedef implementation-defined iterator;    // See 23.1
        typedef implementation-defined const_iterator; // See 23.1
        typedef implementation-defined size_type;      // See 23.1
        typedef implementation-defined difference_type; // See 23.1
        typedef T value_type;
        typedef Allocator allocator_type;
        typedef typename Allocator::pointer         pointer;
        typedef typename Allocator::const_pointer    const_pointer;
        typedef std::reverse_iterator<iterator>      reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

        // 23.2.3.1 construct/copy/destroy:
        explicit list(const Allocator& = Allocator());
        explicit list(size_type n);
        list(size_type n, const T& value,
                const Allocator& = Allocator());
        template <class InputIterator>
        list(InputIterator first, InputIterator last,
                const Allocator& = Allocator());
        list(const list<T,Allocator>& x);
        list(list&& x);
        ~list();
        list<T,Allocator>&& operator=(const list<T,Allocator>&& x);
        list<T,Allocator>& operator=(list<T,Allocator>&& x);
        template <class InputIterator>
        void assign(InputIterator first, InputIterator last);
        void assign(size_type n, const T& t);
        allocator_type get_allocator() const;

        // iterators:
        iterator begin();
        "list();
        list<T,Allocator>&& operator=(const list<T,Allocator>&& x);
        list<T,Allocator>& operator=(list<T,Allocator>&& x);
        template <class InputIterator>
        void assign(InputIterator first, InputIterator last);
        void assign(size_type n, const T& t);
        allocator_type get_allocator() const;
    }
}
```

258) These member functions are only provided by containers whose iterators are random access iterators.
23.2 Sequence containers

Containers library

const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// 23.2.3.2 capacity:
bool empty() const;
size_type size() const;
size_type max_size() const;
void resize(size_type sz);
void resize(size_type sz, T c);

// element access:
reference front();
const_reference front() const;
reference back();
const_reference back() const;

// 23.2.3.3 modifiers:
void push_front(const T& x);
void push_front(T&& x);
template <class... Args> void push_front(Args&&... args);
void pop_front();
void push_back(const T& x);
void push_back(T&& x);
template <class... Args> void push_back(Args&&... args);
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);
void insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
void insert(const_iterator position, InputIterator first, InputIterator last);

iterator erase(const_iterator position);
iterator erase(const_iterator position, const_iterator last);
void swap(list<T,Allocator>&&);
void clear();

// 23.2.3.4 list operations:

Draft
void splice(const_iterator position, list<T,Allocator>&& x);
void splice(const_iterator position, list<T,Allocator>&& x, const_iterator i);
void splice(const_iterator position, list<T,Allocator>&& 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<T,Allocator>&& x);
template <class Compare> void merge(list<T,Allocator>&& x, Compare comp);

void sort();
template <class Compare> void sort(Compare comp);

void reverse();
};

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);
template <class T, class Allocator>
void swap(list<T,Allocator>&& x, list<T,Allocator>& y);
template <class T, class Allocator>
void swap(list<T,Allocator>& x, list<T,Allocator>&& y);
}

23.2.3.1 list constructors, copy, and assignment

explicit list(const Allocator& = Allocator());

1 Effects: Constructs an empty list, using the specified allocator.
2 Complexity: Constant.

Draft
explicit list(size_type n);

Effects: Constructs a list with \( n \) default constructed elements.

Requires: \( T \) shall be DefaultConstructible.

Complexity: Linear in \( n \).

\[
\text{list(size_type } n, \text{ const T& value,} \\
\text{const Allocator& } = \text{ Allocator());}
\]

Effects: Constructs a list with \( n \) copies of \( \text{value} \), using the specified allocator.

Requires: \( T \) shall be CopyConstructible.

Complexity: Linear in \( n \).

\[
\text{template <class InputIterator> } \\
\text{list(InputIterator first, InputIterator last,} \\
\text{const Allocator& } = \text{ Allocator());}
\]

Effects: Constructs a list equal to the range \([\text{first}, \text{last})\).

Complexity: Linear in distance(\( \text{first}, \text{last}) \).

\[
\text{template <class InputIterator> } \\
\text{void assign(InputIterator first, InputIterator last);} \\
\]

Effects: Replaces the contents of the list with the range \([\text{first}, \text{last})\).

erase(begin(), end());
insert(begin(), n, \text{t});

void assign(size_type n, const T& t);

Effects: Replaces the contents of the list with \( n \) copies of \( t \).

23.2.3.2 list capacity

\[
\text{void resize(size_type sz);} \\
\]

Effects: If \( \text{sz < size()} \), equivalent to \text{list<T>::iterator it = begin(); advance(it, sz); erase(it,} \\
\text{end());} . If \text{size()} < sz, appends \text{sz - size()} default constructed elements to the sequence.

Requires: \( T \) shall be DefaultConstructible.

\[
\text{void resize(size_type sz, } T); \\
\]

Effects:

\[
\text{if (sz > size())} \\
\text{insert(end(), sz-size(), c);} \\
\text{else if (sz < size())} \\
\text{iterator i = begin();} \\
\text{advance(i, sz);} \\
\]

Draft
erase(i, end());
} else {
    // do nothing
}

Requires: T shall be CopyConstructible.

23.2.3.3 list modifiers

iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
void insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
    void insert(const_iterator position, InputIterator first, InputIterator last);

void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
template <class... Args> void push_front(Args&&... args);
template <class... Args> void push_back(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);

Remarks: Does not affect the validity of iterators and references. If an exception is thrown there are no effects.

Complexity: Insertion of a single element into a list takes constant time and exactly one call to the copy constructor or move constructor of T. Insertion of multiple elements into a list is linear in the number of elements inserted, and the number of calls to the copy constructor or move constructor of T is exactly equal to the number of elements inserted.

iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);

void pop_front();
void pop_back();
void clear();

Effects: Invalidates only the iterators and references to the erased elements.

Throws: Nothing.

Complexity: Erasing a single element is a constant time operation with a single call to the destructor of T. Erasing a range in a list is linear time in the size of the range and the number of calls to the destructor of type T is exactly equal to the size of the range.

23.2.3.4 list operations

Since lists allow fast insertion and erasing from the middle of a list, certain operations are provided specifically for
23.2 Sequence containers

Containers library

list provides three splice operations that destructively move elements from one list to another.

```cpp
void splice(const_iterator position, list<T,Allocator>&& x);

Requires: &x != this.

Effects: Inserts the contents of x before position and x becomes empty. Pointers and references to the moved elements of x now refer to those same elements but as members of *this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this, not into x.

Throws: Nothing

Complexity: Constant time.
```

```cpp
void splice(const_iterator position, list<T,Allocator>&& x, 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.

Throws: Nothing

Requires: i is a valid dereferenceable iterator of x.

Complexity: Constant time.
```

```cpp
void splice(const_iterator position, list<T,Allocator>&& x, iterator first, 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.

Throws: Nothing unless an exception is thrown by *i == value or pred(*i) != false.

Remarks: Stable.

Complexity: Exactly size() applications of the corresponding predicate.
```

---

259) As specified in 20.1.2, the requirements in this clause apply only to lists whose allocators compare equal.
Containers library

23.2 Sequence containers

```cpp
void unique();  
template <class BinaryPredicate> void unique(BinaryPredicate binary_pred);
```

**Effects:** Eliminates all but the first element from every consecutive group of equal elements referred to by the iterator \(i\) in the range \([\text{first} + 1, \text{last})\) for which \(*i == *(i - 1)\) (for the version of `unique` with no arguments) or \(\text{pred(*i, *(i - 1))}\) (for the version of `unique` with a predicate argument) holds.

**Throws:** Nothing unless an exception is thrown by \(*i == *(i - 1)\) or \(\text{pred(*i, *(i - 1))}\)

**Complexity:** If the range \([\text{first}, \text{last})\) is not empty, exactly \((\text{last} - \text{first}) - 1\) applications of the corresponding predicate, otherwise no applications of the predicate.

```cpp
void merge(list<T,Allocator>&& x);  
template <class Compare> void merge(list<T,Allocator>&& x, Compare comp);
```

**Requires:** `comp` shall define a strict weak ordering (25.3), and both the list and the argument list shall be sorted according to this ordering.

**Effects:** If \((\&x == \text{this})\) does nothing; otherwise, merges the two sorted ranges \([\text{begin()}, \text{end})\) and \([x.\text{begin()}, x.\text{end})\). The result is a range in which the elements will be sorted in non-decreasing order according to the ordering defined by `comp`; that is, for every iterator \(i\) in the range other than the first, the condition `comp(*i, *(i - 1))` will be false.

**Remarks:** Stable. If \((\&x != \text{this})\) the range \([x.\text{begin()}, x.\text{end})\) is empty after the merge.

**Complexity:** At most `size() + x.size() - 1` applications of `comp` if \((\&x != \text{this})\); otherwise, no applications of `comp` are performed. If an exception is thrown other than by a comparison there are no effects.

```cpp
void reverse();
```

**Effects:** Reverses the order of the elements in the list.

**Throws:** Nothing.

**Complexity:** Linear time.

```cpp
void sort();  
template <class Compare> void sort(Compare comp);
```

**Requires:** `operator<` (for the first version) or `comp` (for the second version) defines a strict weak ordering (25.3).

**Effects:** Sorts the list according to the `operator<` or a `Compare` function object.

**Remarks:** Stable.

**Complexity:** Approximately \(N \log(N)\) comparisons, where \(N == \text{size()}\).

### 23.2.3.5 list specialized algorithms

```cpp
template <class T, class Allocator>  
void swap(list<T,Allocator>& x, list<T,Allocator>& y);
```

```cpp
template <class T, class Allocator>  
void swap(list<T,Allocator>&& x, list<T,Allocator>& y);
```

```cpp
template <class T, class Allocator>  
void swap(list<T,Allocator>& x, list<T,Allocator>& y);
```

```cpp
template <class T, class Allocator>  
void swap(list<T,Allocator>&& x, list<T,Allocator>& y);
```

Draft
void swap(list<T, Allocator>& x, list<T, Allocator>&& y);

Effects:
    x.swap(y);

23.2.4 Container adaptors

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.

23.2.4.1 Class template queue

Any sequence container supporting operations front(), back(), push_back() and pop_front() can be used to instantiate queue. In particular, list (23.2.3) and deque (23.2.2) can be used.

23.2.4.1.1 queue definition

```cpp
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());
        queue(queue&& q) : c(std::move(q.c)) {} // 
        queue& operator=(queue&& q) { c = std::move(q.c); return *this; }
        bool empty() const { return c.empty(); }
        size_type size() const { return c.size(); }
        reference front() { return c.front(); }
        const_reference front() const { return c.front(); }
        reference back() { return c.back(); }
        const_reference back() const { return c.back(); }
        void push(const value_type& x) { c.push_back(x); }
        void push(value_type&& x) { c.push_back(std::move(x)); }
        void pop() { c.pop_front(); }
        void swap(queue&& q) { c.swap(q.c); }
    };
}
```

Draft
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);

template <class T, class Container>
void swap(queue<T, Container>&& x, queue<T, Container>& y);

template <class T, class Container>
void swap(queue<T, Container>& x, queue<T, Container>&& y);

23.2.4.1.2 queue operators

template <class T, class Container>
bool operator==(const queue<T, Container>& x,
   const queue<T, Container>& y);

// Returns: x.c == y.c.

template <class T, class Container>
bool operator!=(const queue<T, Container>& x,
   const queue<T, Container>& y);

// Returns: x.c != y.c.

template <class T, class Container>
bool operator<(const queue<T, Container>& x,
   const queue<T, Container>& y);

// Returns: x.c < y.c.

template <class T, class Container>
bool operator<=(const queue<T, Container>& x,
   const queue<T, Container>& y);

// Returns: x.c <= y.c.

template <class T, class Container>

Draft
23.2 Sequence containers

Containers library 690

bool operator> (const queue<T, Container>& x,
const queue<T, Container>& y);

Returns: x.c > y.c.

template <class T, class Container>
bool operator>=(const queue<T, Container>& x,
const queue<T, Container>& y);

Returns: x.c >= y.c.

23.2.4.1.3 queue specialized algorithms

[queue.special]

template <class T, class Container>
void swap(queue<T, Container>& x, queue<T, Container>& y);
template <class T, class Container>
void swap(queue<T, Container>&& x, queue<T, Container>& y);
template <class T, class Container>
void swap(queue<T, Container>& x, queue<T, Container>&& y);

Effects: x.swap(y).

23.2.4.2 Class template priority_queue

[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 (23.2.5) and deque (23.2.2) 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 (25.3).

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,

Draft
const Compare& x,
const Container& y);
template <class InputIterator>
priority_queue(InputIterator first, InputIterator last,
const Compare& x = Compare(),
Container&& y = Container());
priority_queue(priority_queue&&);
priority_queue& operator=(priority_queue&&);

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);
void pop();
void swap(priority_queue&&);

// no equality is provided

template <class T, class Container, Compare>
void swap(priority_queue<T, Container, Compare>& x, priority_queue<T, Container, Compare>& y);

Draft
23.2.4.2.2 \textbf{priority\_queue members} \hfill [priqueue.members]

void push(const value\_type& x);

\hspace{1em} \textit{Effects:}
\hspace{1em} c.push\_back(x);
\hspace{1em} push\_heap(c.begin(), c.end(), comp);

void push(value\_type&& x);

\hspace{1em} \textit{Effects:}
\hspace{1em} c.push\_back(std::move(x));
\hspace{1em} push\_heap(c.begin(), c.end(), comp);

void pop();

\hspace{1em} \textit{Effects:}
\hspace{1em} pop\_heap(c.begin(), c.end(), comp);
\hspace{1em} c.pop\_back();

23.2.4.2.3 \textbf{priority\_queue specialized algorithms} \hfill [priqueue.special]

\begin{verbatim}
template <class T, class Container, Compare>
void swap(priority_queue<T, Container, Compare>& x, priority_queue<T, Container, Compare>& y);
template <class T, class Container, Compare>
void swap(priority_queue<T, Container, Compare>&& x, priority_queue<T, Container, Compare>& y);
template <class T, class Container, Compare>
void swap(priority_queue<T, Container, Compare>& x, priority_queue<T, Container, Compare>&& y);
\end{verbatim}

\hspace{1em} \textit{Effects: }x.swap(y).

23.2.4.3 \textbf{Class template stack} \hfill [stack]

Any sequence container supporting operations \texttt{back()}, \texttt{push\_back()} and \texttt{pop\_back()} can be used to instantiate stack. In particular, \texttt{vector (23.2.5)}, \texttt{list (23.2.3)} and \texttt{deque (23.2.2)} can be used.

23.2.4.3.1 \textbf{stack definition} \hfill [stack.defn]

\begin{verbatim}
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;
    
    stack() : container\_type() {
      // Constructor
    }
  }
}
\end{verbatim}

Draft
Containers library 23.2 Sequence containers

Container c;

public:
  explicit stack(const Container&);
  explicit stack(Container&& = Container());

  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)); }
  void pop() { c.pop_back(); }
  void swap(stack&& s) { c.swap(s.c); }
};

23.2.4.3.2 stack operators

template <class T, class Container>
  bool operator==(const stack<T, Container>& x,
                  const stack<T, Container>& y);

  template <class T, class Container>
  bool operator<(const stack<T, Container>& x,
                 const stack<T, Container>& y);

  template <class T, class Container>
  bool operator!=(const stack<T, Container>& x,
                  const stack<T, Container>& y);

  template <class T, class Container>
  bool operator>(const stack<T, Container>& x,
                 const stack<T, Container>& y);

  template <class T, class Container>
  bool operator>=(const stack<T, Container>& x,
                const stack<T, Container>& y);

  template <class T, class Container>
  bool operator<=(const stack<T, Container>& x,
                const stack<T, Container>& y);

  template <class T, class Allocator>
    void swap(stack<T, Allocator>& x, stack<T, Allocator>& y);

  template <class T, class Allocator>
    void swap(stack<T, Allocator>&& x, stack<T, Allocator>& y);
  template <class T, class Allocator>
    void swap(stack<T, Allocator>& x, stack<T, Allocator>&& y);

1

Returns: x.c == y.c.
23.2 Sequence containers

Containers library  694

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.

23.2.4.3.3 stack specialized algorithms

[stack.special]

template <class T, class Container>
void swap(stack<T, Container>& x, stack<T, Container>& y);

effects: x.swap(y).

23.2.5 Class template vector

A vector is a sequence container that supports random access iterators. In addition, it 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. The elements of a vector are stored contiguously, meaning that if v is a vector<T, Allocator> where T is some type other than bool, then it obeys the identity &v[n] == &v[0] + n for all 0 <= n < v.size().
the push_front and pop_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 typename Allocator::reference reference;
        typedef typename Allocator::const_reference const_reference;
        typedef implementation_defined iterator;       // See 23.1
        typedef implementation_defined const_iterator; // See 23.1
        typedef implementation_defined size_type;       // See 23.1
        typedef implementation_defined difference_type;// See 23.1
        typedef T value_type;
        typedef Allocator allocator_type;
        typedef typename Allocator::pointer pointer;
        typedef typename Allocator::const_pointer const_pointer;
        typedef std::reverse_iterator<iterator> reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

        // 23.2.5.1 construct/copy/destroy:
        explicit vector(const Allocator& = Allocator());
        explicit vector(size_type n);
        vector(size_type n, const T& value, 
               const Allocator& = Allocator());
        template <class InputIterator>
        vector(InputIterator first, InputIterator last, 
               const Allocator& = Allocator());
        vector(const vector<T,Allocator>& x);
        vector(vector&&);
        ~vector();
        vector<T,Allocator>& operator=(const vector<T,Allocator>& x);
        vector<T,Allocator>& operator=(vector<T,Allocator>&& x);
        template <class InputIterator>
        void assign(InputIterator first, InputIterator last);
        void assign(size_type n, const T& u);
        allocator_type get_allocator() const;

        // iterators:
        iterator begin();
        const_iterator begin() const;
        iterator end();
        const_iterator end() const;
        reverse_iterator rbegin();
        const_reverse_iterator rbegin() const;
        reverse_iterator rend();
        const_reverse_iterator rend() const;
    };
const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// 23.2.5.2 capacity:
size_type size() const;
size_type max_size() const;
void resize(size_type sz);
void resize(size_type sz, T c);
size_type capacity() const;
bool empty() const;
void reserve(size_type n);

// 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;

// 23.2.5.3 data access
pointer data();
const_pointer data() const;

// 23.2.5.4 modifiers:
void push_back(const T& x);
void push_back(T&& x);
template <class... Args> void push_back(Args&&... args);
void pop_back();

template <class... Args> iterator emplace(const_iterator position, Args&&... args);
iterator insert(const_iterator position, const T& x);
void insert(const_iterator position, size_type n, const T& x);
void insert(const_iterator position, size_type n, T&& x);
template <class InputIterator>

    void insert(const_iterator position,
               InputIterator first, InputIterator last);
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void swap(vector<T, Allocator>&&);
void clear();
};

template <class T, class Allocator>
bool operator==(const vector<T, Allocator>& x,
               const vector<T, Allocator>& y);

Draft
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);

// specialized algorithms:
template <class T, class Allocator>
void swap(vector<T,Allocator>& x, vector<T,Allocator>& y);

23.2.5.1 vector constructors, copy, and assignment

vector(const Allocator& = Allocator());

Effects: Constructs an empty vector, using the specified allocator.
Complexity: Constant.

explicit vector(size_type n);

Effects: Constructs a vector with n default constructed elements.
Requires: T shall be DefaultConstructible.
Complexity: Linear in n.

explicit 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 CopyConstructible.
Complexity: Linear in n.

template <class InputIterator>
vector(InputIterator first, InputIterator last,
const Allocator& = Allocator());

Draft
23.2 Sequence containers

Effects: Constructs a vector equal to the range \([\text{first}, \text{last})\), using the specified allocator.

Complexity: Makes only \(N\) calls to the copy constructor of \(T\) (where \(N\) is the distance between \(\text{first}\) and \(\text{last}\)) 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.

```cpp
template <class InputIterator>
void assign(InputIterator first, InputIterator last);
```

Effects:
- \(\text{erase(}\text{begin(), end())}\);
- \(\text{insert(}\text{begin(), first, last))}\);

```cpp
void assign(size_type n, const T& t);
```

Effects:
- \(\text{erase(}\text{begin(), end())}\);
- \(\text{insert(}\text{begin(), n, t)}\);

23.2.5.2 vector capacity

```cpp
size_type capacity() const;
```

Returns: The total number of elements that the vector can hold without requiring reallocation.

```cpp
void reserve(size_type n);
```

Requires: If \(\text{value_type}\) has a move constructor, that constructor shall not throw any exceptions.

Effects: A directive that informs a vector of a planned change in size, so that it can manage the storage allocation accordingly. After \(\text{reserve()}\), \(\text{capacity()}\) is greater or equal to the argument of \(\text{reserve}\) if reallocation happens; and equal to the previous value of \(\text{capacity()}\) otherwise. Reallocation happens at this point if and only if the current capacity is less than the argument of \(\text{reserve()}\). If an exception is thrown, 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: \(\text{length_error}\) if \(n > \text{max_size()}\).

Remarks: Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. It is guaranteed that no reallocation takes place during insertions that happen after a call to \(\text{reserve()}\) until the time when an insertion would make the size of the vector greater than the value of \(\text{capacity()}\).

```cpp
void swap(vector<T, Allocator>&& x);
```

Effects: Exchanges the contents and \(\text{capacity()}\) of \(*\text{this}\) with that of \(x\).

Complexity: Constant time.

```cpp
void resize(size_type sz);
```

\(\text{reserve()}\) uses \(\text{Allocator::allocate()}\) which may throw an appropriate exception.
Effects: If \( sz < \text{size()} \), equivalent to \( \text{erase(begin() + sz, end());} \). If \( \text{size()} < sz \), appends \( sz - \text{size()} \) default constructed elements to the sequence.

Requires: \( T \) shall be DefaultConstructible.

```cpp
void resize(size_type sz, T c = T());
```

Effects:

```cpp
if (sz > size())
    insert(end(), sz-size(), c);
else if (sz < size())
    erase(begin()+sz, end());
else
    // do nothing
```

Requires: If value_type has a move constructor, that constructor shall not throw any exceptions.

### 23.2.5.3 vector data

- **pointe**r `data()`;
- **const_pointer** `data()` `const`;

Returns: A pointer such that \([\text{data()}, \text{data()} + \text{size()}]\) is a valid range. For a non-empty vector, \( \text{data()} == \&\text{front()} \).

**Complexity:** Constant time.

**Throws:** Nothing.

### 23.2.5.4 vector modifiers

```cpp
void push_back(const T& x);
void push_back(T&& x);
```

Requires: If value_type has a move constructor, that constructor shall not throw any exceptions.

```cpp
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
void insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
    void insert(const_iterator position, InputIterator first, InputIterator last);
```

```cpp
template <class... Args> void push_back(Args&&... args);
```

```cpp
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
```

Requires: If value_type has a move constructor, that constructor shall not throw any exceptions.

**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 or assignment operator of \( T \) or by any InputIterator operation there are no effects.
23.2 Sequence containers

 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);

 Effects: Invalidates iterators and references at or after the point of the erase.

 Complexity: The destructor of T is called the number of times equal to the number of the elements erased, but the move assignment operator of T is called the number of times equal to the number of elements in the vector after the erased elements.

 Throws: Nothing unless an exception is thrown by the copy constructor or assignment operator of T.

23.2.5.5 vector specialized algorithms

 template <class T, class Allocator>
 void swap(vector<T,Allocator>& x, vector<T,Allocator>& y);
 template <class T, class Allocator>
 void swap(vector<T,Allocator>&& x, vector<T,Allocator>& y);
 template <class T, class Allocator>
 void swap(vector<T,Allocator>& x, vector<T,Allocator>&& y);

 Effects: x.swap(y);

23.2.6 Class vector<bool>

 To optimize space allocation, a specialization of vector for bool elements is provided:

 namespace std {
  template <class Allocator> class vector<bool, Allocator> {
   public:
   // types:
   typedef bool const_reference;
   typedef implementation-defined iterator; // See 23.1
   typedef implementation-defined const_iterator; // See 23.1
   typedef implementation-defined size_type; // See 23.1
   typedef implementation-defined difference_type; // See 23.1
   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();
   };

   //...
public:

~reference();
operator bool() const;
reference& operator=(const bool x);
reference& operator=(const reference& x);
void flip();  // flips the bit
};

// construct/copy/destroy:
explicit vector(const Allocator& = Allocator());
explicit vector(size_type n, const bool& value = bool(),
     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();
vector<bool,Allocator>& operator=(const vector<bool,Allocator>& x);
vector<bool,Allocator>& operator=(vector<bool,Allocator>&& x);
template <class InputIterator>
    void assign(InputIterator first, InputIterator last);
void assign(size_type n, const bool& t);
allocator_type get_allocator() const;

// iterators:
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    const_iterator cbegin() const;
    const_iterator cend() const;
    const_reverse_iterator crbegin() const;
    const_reverse_iterator crend() const;

// capacity:
    size_type size() const;
    size_type max_size() const;
    void resize(size_type sz, bool c = false);
    size_type capacity() const;
    bool empty() const;
    void reserve(size_type n);

// element access:
    reference operator[](size_type n);
23.3 Associative containers

Headers `<map>` and `<set>`:

**Header `<map>` synopsis**

```cpp
namespace std {
    template <class Key, class T, class Compare = less<Key>,
             class Allocator = allocator<pair<const Key, T> > >
    class map {
```

---

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.

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();
```

**Effects:** Replaces each element in the container with its complement. It is unspecified whether the function has any effect on allocated but unused bits.

---

23.3 Associative containers

Headers `<map>` and `<set>`:

**Header `<map>` synopsis**

```cpp
namespace std {
    template <class Key, class T, class Compare = less<Key>,
             class Allocator = allocator<pair<const Key, T> > >
    class map {
```
class map;

    template <class Key, class T, class Compare, class Allocator>
    bool operator==(const map<Key,T,Compare,Allocator>& x,
                 const map<Key,T,Compare,Allocator>& y);

    template <class Key, class T, class Compare, class Allocator>
    bool operator<(const map<Key,T,Compare,Allocator>& x,
                  const map<Key,T,Compare,Allocator>& y);

    template <class Key, class T, class Compare, class Allocator>
    bool operator!=(const map<Key,T,Compare,Allocator>& x,
                  const map<Key,T,Compare,Allocator>& y);

    template <class Key, class T, class Compare, class Allocator>
    bool operator>(const map<Key,T,Compare,Allocator>& x,
                  const map<Key,T,Compare,Allocator>& y);

    template <class Key, class T, class Compare, class Allocator>
    bool operator>=(const map<Key,T,Compare,Allocator>& x,
                  const map<Key,T,Compare,Allocator>& y);

    template <class Key, class T, class Compare, class Allocator>
    bool operator<=(const map<Key,T,Compare,Allocator>& x,
                  const map<Key,T,Compare,Allocator>& y);

    template <class Key, class T, class Compare, class Allocator>
    void swap(map<Key,T,Compare,Allocator>& x,
              map<Key,T,Compare,Allocator>& y);

    template <class Key, class T, class Compare, class Allocator>
    void swap(map<Key,T,Compare,Allocator>& x,
              map<Key,T,Compare,Allocator>& y);

    template <class Key, class T, class Compare, class Allocator>
    void swap(map<Key,T,Compare,Allocator>& x,
              map<Key,T,Compare,Allocator>& y);

    template <class Key, class T, class Compare = less<Key>,
                class Allocator = allocator<pair<const Key, T>>>
    class multimap;

    template <class Key, class T, class Compare, class Allocator>
    bool operator==(const multimap<Key,T,Compare,Allocator>& x,
                   const multimap<Key,T,Compare,Allocator>& y);

    template <class Key, class T, class Compare, class Allocator>
    bool operator<(const multimap<Key,T,Compare,Allocator>& x,
                   const multimap<Key,T,Compare,Allocator>& y);

    template <class Key, class T, class Compare, class Allocator>
    bool operator!=(const multimap<Key,T,Compare,Allocator>& x,
                   const multimap<Key,T,Compare,Allocator>& y);

    template <class Key, class T, class Compare, class Allocator>
    bool operator>(const multimap<Key,T,Compare,Allocator>& x,
                   const multimap<Key,T,Compare,Allocator>& y);

    template <class Key, class T, class Compare, class Allocator>
    bool operator>=(const multimap<Key,T,Compare,Allocator>& x,
                   const multimap<Key,T,Compare,Allocator>& y);

    template <class Key, class T, class Compare, class Allocator>
    bool operator<=(const multimap<Key,T,Compare,Allocator>& x,
                   const multimap<Key,T,Compare,Allocator>& y);

    template <class Key, class T, class Compare, class Allocator>
    void swap(multimap<Key,T,Compare,Allocator>& x,
              multimap<Key,T,Compare,Allocator>& y);

    template <class Key, class T, class Compare, class Allocator>
    void swap(multimap<Key,T,Compare,Allocator>& x,
              multimap<Key,T,Compare,Allocator>& y);

    template <class Key, class T, class Compare, class Allocator>
    void swap(multimap<Key,T,Compare,Allocator>& x,
              multimap<Key,T,Compare,Allocator>& y);

Draft
template <class Key, class T, class Compare, class Allocator>
void swap(multimap<Key,T,Compare,Allocator>& x,
multimap<Key,T,Compare,Allocator>& y);

Header <set> synopsis

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);
    template <class Key, class Compare, class Allocator>
    void swap(set<Key,Compare,Allocator>& x,
              set<Key,Compare,Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    void swap(set<Key,T,Compare,Allocator>& x,
              set<Key,T,Compare,Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    void swap(set<Key,T,Compare,Allocator>&& x,
              set<Key,T,Compare,Allocator>& 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>
    bool operator<=(const multiset<Key,Compare,Allocator>& x,
                   const multiset<Key,Compare,Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    void swap(multiset<Key,T,Compare,Allocator>& x,
              multiset<Key,T,Compare,Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
    void swap(multiset<Key,T,Compare,Allocator>& x,
              multiset<Key,T,Compare,Allocator>&& y);
}
23.3.1 Class template map

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 and of a reversible container (23.1) and of an associative container (23.1.2). A map also provides most operations described in (23.1.2) for unique keys. This means that a map supports the \( \text{a\_uniq} \) operations in (23.1.2) but not the \( \text{a\_eq} \) operations. For a map\( \langle \text{Key},T \rangle \) the \text{key\_type} is \text{Key} and the \text{value\_type} is \text{pair<const Key,T>}. Descriptions are provided here only for operations on map that are not described in one of those tables or for operations where there is additional semantic information.

```cpp
namespace std {
    template <class Key, class T, class Compare = less<Key>,
            class Allocator = allocator<pair<const Key, T> > >
    class map {
    public:
        // types:
        typedef Key key_type;
        typedef T mapped_type;
        typedef pair<const Key, T> value_type;
        typedef Compare key_compare;
        typedef Allocator allocator_type;
        typedef typename Allocator::reference reference;
    }
}
```
Associative containers

Containers library

```cpp
typedef typename Allocator::const_reference const_reference;
typedef implementation-defined iterator; // See 23.1
typedef implementation-defined const_iterator; // See 23.1
typedef implementation-defined size_type; // See 23.1
typedef implementation-defined difference_type; // See 23.1
typedef typename Allocator::pointer pointer;
typedef typename Allocator::const_pointer const_pointer;
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

class value_compare
    : public binary_function<value_type, value_type, bool> {
    friend class map;
    protected:
    Compare comp;
    value_compare(Compare c) : comp(c) {}
    public:
    bool operator()(const value_type& x, const value_type& y) const {
        return comp(x.first, y.first);
    }
};

// 23.3.1.1 construct/copy/destroy:
explicit map(const Compare& comp = Compare(),
             const Allocator& = Allocator());
template <class InputIterator>
    map(InputIterator first, InputIterator last,
         const Compare& comp = Compare(), const Allocator& = Allocator());
map(const map<Key, T, Compare, Allocator>& x);
map(map<Key, T, Compare, Allocator>&& x);
~map();
map<Key, T, Compare, Allocator>&
    operator=(const map<Key, T, Compare, Allocator>&& x);
map<Key, T, Compare, Allocator>&
    operator=(map<Key, T, Compare, Allocator>&& x);
allocator_type get_allocator() const;

// iterators:
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
const_iterator cbegin() const;
const_iterator cend() const;
```
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// capacity:
bool empty() const;
size_type size() const;
size_type max_size() const;

// 23.3.1.2 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;

// modifiers:
templates

pair<iterator, bool> insert(const value_type& x);
template<class P> pair<iterator, bool> insert(P&& x);

pair<iterator, iterator> equal_range(const key_type& x);

pair<const_iterator, const_iterator> equal_range(const key_type& x) const;

};

Draft
23.3 Associative containers

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);

// 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);
template <class Key, class T, class Compare, class Allocator>
void swap(map<Key,T,Compare,Allocator&& x,
          map<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
void swap(map<Key,T,Compare,Allocator& x,
          map<Key,T,Compare,Allocator>& y);

23.3.1 map constructors, copy, and assignment

explicit map(const Compare& comp = Compare(),
             const Allocator& = Allocator());

Effects: Constructs an empty map using the specified comparison object and allocator.

Complexity: Constant.

template <class InputIterator>
map(InputIterator first, InputIterator last,
    const Compare& comp = Compare(), const Allocator& = Allocator());

Requires: If the iterator's dereference operator returns an lvalue or a const rvalue pair<key_type, mapped_type>, then both key_type and mapped_type shall be CopyConstructible.

Effects: Constructs an empty map using the specified comparison object and allocator, and inserts elements from the range [first, last).
Completeness: Linear in \(N\) if the range \([first, last)\) is already sorted using \(comp\) and otherwise \(N \log N\), where \(N\) is \(last - first\).

### 23.3.1.2 map element access

\[\text{T& operator[] (const key_type& x);}\]

1. **Effects**: If there is no key equivalent to \(x\) in the map, inserts \(\text{value_type}(x, \text{T}())\) into the map.
2. **Requires**: \(\text{key_type}\) shall be \(\text{CopyConstructible}\) and \(\text{mapped_type}\) shall be \(\text{DefaultConstructible}\).
3. **Returns**: A reference to the \(\text{mapped_type}\) corresponding to \(x\) in \(*\text{this}\).
4. **Complexity**: Logarithmic.

\[\text{T& operator[] (const key_type&& x);}\]

5. **Effects**: If there is no key equivalent to \(x\) in the map, inserts \(\text{value_type}(\text{std::move}(x), \text{T}())\) into the map.
6. **Requires**: \(\text{mapped_type}\) shall be \(\text{DefaultConstructible}\).
7. **Returns**: A reference to the \(\text{mapped_type}\) corresponding to \(x\) in \(*\text{this}\).
8. **Complexity**: Logarithmic.

\[\text{T& at(const key_type& x);}\]

9. **Returns**: A reference to the element whose key is equivalent to \(x\).
10. **Throws**: An exception object of type \(\text{out_of_range}\) if no such element is present.

### 23.3.1.3 map modifiers

\[\text{template <class P> pair<iterator, bool> insert(P&& x);}\]

\[\text{template <class P> pair<\text{const}_\text{iterator}, bool> insert(\text{\text{const}}\_\text{iterator} \text{position, P&& x});}\]

1. **Requires**: \(P\) shall be convertible to \(\text{value_type}\).

If \(P\) is instantiated as a reference type, then the argument \(x\) is copied from. Otherwise \(x\) is considered to be an rvalue as it is converted to \(\text{value_type}\) and inserted into the map. Specifically, in such cases \(\text{CopyConstructible}\) is not required of \(\text{key_type}\) or \(\text{mapped_type}\) unless the conversion from \(P\) specifically requires it (e.g. if \(P\) is a tuple<\text{const key_type}, mapped_type>, then \text{key_type} must be \text{CopyConstructible}). The signature taking \text{InputIterator} parameters does not require \text{CopyConstructible} of either \text{key_type} or \text{mapped_type} if the dereferenced \text{InputIterator} returns a non-const rvalue \text{pair<key_type,mapped_type>}. Otherwise \text{CopyConstructible} is required for both \text{key_type} and \text{mapped_type}.

### 23.3.1.4 map operations

Draft
The `find`, `lower_bound`, `upper_bound` and `equal_range` member functions each have two versions, one const and the other non-const. In each case the behavior of the two functions is identical except that the const version returns a `const_iterator` and the non-const version an `iterator`.

### 23.3.1.5 map specialized algorithms

```cpp
template <class Key, class T, class Compare, class Allocator>
void swap(map<Key, T, Compare, Allocator>& x, map<Key, T, Compare, Allocator>& y);
```

Effects:

\[ x.\text{swap}(y); \]

### 23.3.2 Class template multimap

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 (23.1) and of an associative container (23.1.2). A multimap also provides most operations described in (23.1.2) for equal keys. This means that a multimap supports the \( \text{a_eq} \) operations in (23.1.2) but not the \( \text{a_uniq} \) 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>>>
    Draft
```
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 typename Allocator::reference reference;
    typedef typename Allocator::const_reference const_reference;
    typedef implementation_defined iterator;
    // See 23.1
    typedef implementation_defined const_iterator;
    // See 23.1
    typedef implementation_defined size_type; // See 23.1
    typedef implementation_defined difference_type; // See 23.1
    typedef typename Allocator::pointer pointer;
    typedef typename Allocator::const_pointer const_pointer;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    class value_compare :
        public binary_function<value_type,value_type,bool> {
    protected:
        Compare comp;
    public:
        value_compare(Compare c) : comp(c) {}
        bool operator()(const value_type& x, const value_type& y) const {
            return comp(x.first, y.first);
        }
    };

    // construct/copy/destroy:
    explicit multimap(const Compare& comp = Compare(),
                      const Allocator& = Allocator());
    template <class InputIterator>
    multimap(InputIterator first, InputIterator last,
              const Compare& comp = Compare(),
              const Allocator& = Allocator());
    multimap(const multimap<Key,T,Compare,Allocator>& x);
    multimap(multimap<Key,T,Compare,Allocator>&& x);
    multimap();
    multimap<Key,T,Compare,Allocator>&
        operator=(const multimap<Key,T,Compare,Allocator>&& x);
    multimap<Key,T,Compare,Allocator>&
        operator=(const multimap<Key,T,Compare,Allocator>&& x);
    allocator_type get_allocator() const;

    // iterators:
    iterator begin();
    const_iterator begin() const;
};
iterator end();
const_iterator end() const;

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// capacity:
bool empty() const;
size_type size() const;
size_type max_size() const;

// modifiers:
template <class... Args> iterator emplace(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
iterator insert(const value_type& x);
template <class P> iterator insert(P&& x);
iterator insert(const_iterator position, const value_type& x);
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<Key,T,Compare,Allocator>&&);
void clear();

// 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;
size_type count(const key_type& x) const;
iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& 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 Key, class T, class Compare, class Allocator>
   bool operator==(const multimap<Key,T,Compare,Allocator>& x,
                  const multimap<Key,T,Compare,Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
   bool operator<(const multimap<Key,T,Compare,Allocator>& x,
                 const multimap<Key,T,Compare,Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
   bool operator!=(const multimap<Key,T,Compare,Allocator>& x,
                 const multimap<Key,T,Compare,Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
   bool operator>(const multimap<Key,T,Compare,Allocator>& x,
                 const multimap<Key,T,Compare,Allocator>& y);

// 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);

23.3.2.1 multimap constructors

explicit multimap(const Compare& comp = Compare(),
                  const Allocator& = Allocator());

Effects: Constructs an empty multimap using the specified comparison object and allocator.

Complexity: Constant.

template <class InputIterator>
   multimap(InputIterator first, InputIterator last,
             const Compare& comp = Compare(),
             const Allocator& = Allocator());

Requires: If the iterator’s dereference operator returns an lvalue or a const rvalue pair<key_type, mapped_type>, then both key_type and mapped_type shall be CopyConstructible.
Effects: Constructs an empty multimap using the specified comparison object and allocator, and inserts elements from the range \([first, last)\).

Complexity: Linear in \(N\) if the range \([first, last)\) is already sorted using \(\text{comp}\) and otherwise \(N \log N\), where \(N = last - first\).

### 23.3.2.2 multimap modifiers

```cpp
template <class P> iterator insert(P&& x);
template <class P> iterator insert(const_iterator position, P&& x);
```

Requires: \(P\) shall be convertible to value_type.

If \(P\) is instantiated as a reference type, then the argument \(x\) is copied from. Otherwise \(x\) is considered to be an rvalue as it is converted to value_type and inserted into the map. Specifically, in such cases CopyConstructible is not required of key_type or t mismapped_type unless the conversion from \(P\) specifically requires it (e.g. if \(P\) is a tuple<const key_type, mapped_type>, then key_type must be CopyConstructible). The signature taking InputIterator parameters does not require CopyConstructible of either key_type or mapped_type if the dereferenced InputIterator returns a non-const rvalue pair<key_type,mapped_type>. Otherwise CopyConstructible is required for both key_type and mapped_type.

### 23.3.2.3 multimap operations

```cpp
iterator find(const key_type &x);
const_iterator find(const key_type& x) const;

iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;

pair<iterator, iterator> equal_range(const key_type& x);
pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
```

The \(\text{find}\), \(\text{lower_bound}\), \(\text{upper_bound}\), and \(\text{equal_range}\) member functions each have two versions, one const and one non-const. In each case the behavior of the two versions is identical except that the const version returns a const_iterator and the non-const version an iterator (23.1.2).

### 23.3.2.4 multimap specialized algorithms

```cpp
template <class Key, class T, class Compare, class Allocator>
void swap(multimap<Key,T,Compare,Allocator>& x, multimap<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
void swap(multimap<Key,T,Compare,Allocator>&& x, multimap<Key,T,Compare,Allocator>&& y);
template <class Key, class T, class Compare, class Allocator>
void swap(multimap<Key,T,Compare,Allocator>& x, multimap<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
void swap(multimap<Key,T,Compare,Allocator>&& x, multimap<Key,T,Compare,Allocator>&& y);
```

Draft
23.3.3 Class template set

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. Class set supports bidirectional iterators.

A set satisfies all of the requirements of a container and of a reversible container (23.1), and of an associative container (23.1.2). A set also provides most operations described in (23.1.2) for unique keys. This means that a set supports the a_uniq operations in (23.1.2) 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 typename Allocator::reference reference;
            typedef typename Allocator::const_reference const_reference;
            typedef implementation-defined iterator;
            // See 23.1
            typedef implementation-defined const_iterator;
            // See 23.1
            typedef implementation-defined size_type;
            // See 23.1
            typedef implementation-defined difference_type;
            // See 23.1
            typedef typename Allocator::pointer pointer;
            typedef typename Allocator::const_pointer const_pointer;
            typedef std::reverse_iterator<iterator> reverse_iterator;
            typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

            // 23.3.3.1 construct/copy/destroy:
            explicit set(const Compare& comp = Compare(),
                         const Allocator& = Allocator());
            template <class InputIterator>
            set(InputIterator first, InputIterator last,
                 const Compare& comp = Compare(), const Allocator& = Allocator());
            set(const set<Key,Compare,Allocator>& x);
            set(set<Key,Compare,Allocator>&& x);
            ~set();
            set<Key,Compare,Allocator>& operator=(
                const set<Key,Compare,Allocator>& x);
```

Draft
set<Key,Compare,Allocator>& operator=
(set<Key,Compare,Allocator>&& x);
allocator_type get_allocator() const;

// iterators:
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// capacity:
bool empty() const;
size_type size() const;
size_type max_size() const;

// modifiers:
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
template <class... Args> iterator emplace(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);

iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);
void swap(set<Key,Compare,Allocator>&);
void clear();

// 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;

size_type count(const key_type& x) const;
iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;

iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& 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 Key, class Compare, class Allocator>
bool operator==(const set<Key,Compare,Allocator>& x,
                const set<Key,Compare,Allocator>& y);

// specialized algorithms:
void swap(set<Key,Compare,Allocator>& x,
          set<Key,Compare,Allocator>& y);

23.3.3.1 set constructors, copy, and assignment

explicit set(const Compare& comp = Compare(),
              const Allocator& = Allocator());

Effects: Constructs an empty set using the specified comparison objects and allocator.

Complexity: Constant.
23.3 Associative containers

`template <class InputIterator>`
`set(InputIterator first, InputIterator last,`  
`    const Compare& comp = Compare(), const Allocator& = Allocator());`

3 Effects: Constructs an empty set using the specified comparison object and allocator, and inserts elements from the range \([first, last)\).

4 Requires: If the iterator's dereference operator returns an lvalue or a non-const rvalue, then Key shall be CopyConstructible.

5 Complexity: Linear in \(N\) if the range \([first, last)\) is already sorted using \(comp\) and otherwise \(N \log N\), where \(N\) is \(last - first\).

23.3.3.2 `set` specialized algorithms

`template <class Key, class Compare, class Allocator>`
`void swap(set<Key,Compare,Allocator>& x,`  
`    set<Key,Compare,Allocator>& y);`
`template <class Key, class Compare, class Allocator>`
`void swap(set<Key,Compare,Allocator>&& x,`  
`    set<Key,Compare,Allocator>& y);`
`template <class Key, class Compare, class Allocator>`
`void swap(set<Key,Compare,Allocator>& x,`  
`    set<Key,Compare,Allocator>&& y);`

1 Effects:

\(x\).swap(\(y\));

23.3.4 Class template `multiset`

1 A multiset is an associative container that supports equivalent keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of the keys themselves. Class `multiset` supports bidirectional iterators.

2 A multiset satisfies all of the requirements of a container and of a reversible container (23.1), and of an associative container (23.1.2). multiset also provides most operations described in (23.1.2) for duplicate keys. This means that a multiset supports the `a_eq` operations in (23.1.2) but not the `a_uniq` operations. For a `multiset<Key>` both the `key_type` and `value_type` are Key. Descriptions are provided here only for operations on `multiset` that are not described in one of these tables and for operations where there is additional semantic information.

```cpp
namespace std {
    template <class Key, class Compare = less<Key>,
              class Allocator = allocator<Key> >
    class multiset {
    public:
        // types:
        typedef Key key_type;
        typedef Key value_type;
        typedef Compare key_compare;
        typedef Compare value_compare;
        typedef Allocator allocator_type;
    }
```
typedef typename Allocator::reference reference;
typedef typename Allocator::const_reference const_reference;
typedef implementation-defined iterator;  // See 23.1
typedef implementation-defined const_iterator;  // See 23.1
typedef implementation-defined size_type;  // See 23.1
typedef implementation-defined difference_type;  // See 23.1
typedef typename Allocator::pointer pointer;
typedef typename Allocator::const_pointer const_pointer;
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

// construct/copy/destroy:
explicit multiset(const Compare& comp = Compare(),
                 const Allocator& = Allocator());
template <class InputIterator>
multiset(InputIterator first, InputIterator last,
         const Compare& comp = Compare(),
         const Allocator& = Allocator());
multiset(const multiset<Key,Compare,Allocator>& x);
multiset(multiset<Key,Compare,Allocator>&& x);
~multiset();
multiset<Key,Compare,Allocator>&
    operator=(const multiset<Key,Compare,Allocator>&& x);
multiset<Key,Compare,Allocator>&
    operator=(multiset<Key,Compare,Allocator>&& x);
allocator_type get_allocator() const;

// iterators:
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// capacity:
bool empty() const;
size_type size() const;
size_type max_size() const;

// modifiers:
template <class... Args> iterator emplace(Args&&... args);

Draft
23.3 Associative containers

Containers library 720

```cpp
template <class... Args> iterator emplace(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);

iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);
void swap(multiset<Key, Compare, Allocator>&&);
void clear();

// 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;
size_type count(const key_type& x) const;

iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;
pair<iterator, iterator> equal_range(const key_type& x);
pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
```

```cpp
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);
```

Draft
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);

23.3.4.1 multiset constructors

explicit multiset(const Compare& comp = Compare(), const Allocator& = Allocator());

Effects: Constructs an empty set using the specified comparison object and allocator.

Complexity: Constant.

template <class InputIterator>
multiset(InputIterator first, last, const Compare& comp = Compare(), const Allocator& = Allocator());

Requires: If the iterator’s dereference operator returns an lvalue or a const rvalue, then Key shall be CopyConstructible.

Effects: Constructs an empty multiset using the specified comparison object and allocator, and inserts elements from the range [first, last).

Complexity: Linear in N if the range [first, last) is already sorted using comp and otherwise N log N, where N is last - first.

23.3.4.2 multiset specialized algorithms

template <class Key, class Compare, class Allocator>
void swap(multiset<Key,Compare,Allocator>& x, multiset<Key,Compare,Allocator>& y);

Effects:

x.swap(y);
23.3 Associative containers

23.3.5 Class template bitset

Header <bitset> synopsis

```cpp
#include <cstddef> // for size_t
#include <string>
#include <stdexcept> // for invalid_argument,
                   // out_of_range, overflow_error
#include <iosfwd> // for istream, ostream
namespace std {
    template <size_t N> class bitset;

    // 23.3.5.3 bitset operators:
    template <size_t N>
    bitset<N> operator&(const bitset<N>&, const bitset<N>&);
    template <size_t N>
    bitset<N> operator|(const bitset<N>&, const bitset<N>&);
    template <size_t N>
    bitset<N> operator^(const bitset<N>&, const bitset<N>&);
    template <class charT, class traits, size_t N>
    basic_istream<charT, traits>&
        operator>>(basic_istream<charT, traits>& is, bitset<N>& x);
    template <class charT, class traits, size_t N>
    basic_ostream<charT, traits>&
        operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);
}
```

The header <bitset> defines a class template and several related functions for representing and manipulating fixed-size sequences of bits.

```cpp
namespace std {
    template <size_t N> class bitset {
        public:
            // bit reference:
            class reference {
                friend class bitset;
                reference();
                public:
                    "reference();
                    reference& operator=(bool x); // for b[i] = x;
                    reference& operator=(const reference&); // for b[i] = b[j];
                    bool operator~() const; // flips the bit
                    operator bool() const; // for x = b[i];
                    reference& flip(); // for b[i].flip();
            };

            // 23.3.5.1 constructors:
            constexpr bitset();
            constexpr bitset(unsigned long val);
            template<class charT, class traits, class Allocator>
            explicit bitset(
```
The template class `bitset<N>` describes an object that can store a sequence consisting of a fixed number of bits, \( N \).

Each bit represents either the value zero (reset) or one (set). To *toggle* a bit is to change the value zero to one, or the value one to zero. Each bit has a non-negative position \( pos \). When converting between an object of class `bitset<N>` and a value of some integral type, bit position \( pos \) corresponds to the *bit value* \( 1 \ll pos \). The integral value corresponding to two or more bits is the sum of their bit values.
The functions described in this subclause can report three kinds of errors, each associated with a distinct exception:

- an invalid-argument error is associated with exceptions of type invalid_argument (19.1.3);
- an out-of-range error is associated with exceptions of type out_of_range (19.1.5);
- an overflow error is associated with exceptions of type overflow_error (19.1.8).

### 23.3.5.1 bitset constructors

```cpp
constexpr bitset();
```

**Effects:** Constructs an object of class `bitset<N>`, initializing all bits to zero.

```cpp
constexpr bitset(unsigned long val);
```

**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 (section 3.9) of unsigned long. If $M < N$, the remaining bit positions are initialized to zero.

```cpp
template <class charT, class traits, class Allocator>
explicit
bitset(const basic_string<charT, traits, Allocator>& str,
      typename basic_string<charT, traits, Allocator>::size_type pos = 0,
      typename basic_string<charT, traits, Allocator>::size_type n =
      basic_string<charT, traits, Allocator>::npos);
```

**Requires:** $pos \leq str.size()$.

**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 0 or 1.

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 string has value zero if the corresponding character in `str`, beginning at position $pos$, is 0. 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.

### 23.3.5.2 bitset members

```cpp
bitset<N>& operator=(const bitset<N>& rhs);
```

**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);

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);

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);

Effects: Replaces each bit at position I in *this with a value determined as follows:
- If I < pos, the new value is zero;
- If I >= pos, the new value is the previous value of the bit at position I - pos.

Returns: *this.

bitset<N>& operator>>=(size_t pos);

Effects: Replaces each bit at position I in *this with a value determined as follows:
- If pos >= N - I, the new value is zero;
- If pos < N - I, the new value is the previous value of the bit at position I + pos.

Returns: *this.

bitset<N>& set();

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();

Effects: Resets all bits in *this.

Returns: *this.

Draft
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;
   Effects: Constructs an object z of class bitset<N> and initializes it with *this.
   Returns: z.flip().

bitset<N>& flip();
   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.

template <class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator> to_string() 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 0, bit value one becomes the character 1.
   Returns: The created object.

template <class charT, class traits>
  basic_string<charT, traits, allocator<charT>> to_string() const;
   Returns: to_string<charT, traits, allocator<charT>>().

template <class charT>
  basic_string<charT, char_traits<charT>, allocator<charT>> to_string() const;
Returns: to_string<charT, char_traits<charT>, allocator<charT> >().

basic_string<char, char_traits<char>, allocator<char> > to_string() const;

Returns: to_string<char, char_traits<char>, allocator<char> >().

size_t count() const;

Returns: A count of the number of bits set in *this.

constexpr size_t size() const;

Returns: N.

bool operator==(const bitset<N>& rhs) const;

Returns: A nonzero value if the value of each bit in *this equals the value of the corresponding bit in rhs.

bool operator!=(const bitset<N>& rhs) const;

Returns: A nonzero value if !(*this == rhs).

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.

bool any() const;

Returns: true if any bit in *this is one.

bool none() const;

Returns: true if no bit in *this is one.

bitset<N> operator<<(size_t pos) const;

Returns: bitset<N>(*this) <<= pos.

bitset<N> operator>>(size_t pos) const;

Returns: bitset<N>(*this) >>= pos.

constexpr bool operator[](size_t pos) const;

Requires: pos is valid.

Throws: nothing.

Returns: test(pos).

bitset<N>::reference operator[](size_t pos);

Requires: pos is valid.
23.4 Unordered associative containers

Headers `<unordered_map>` and `<unordered_set>`:

Header `<unordered_map>` synopsis

Draft
namespace std {
    // 23.4.1, 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;

    // 23.4.2, 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);

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);
} // namespace std

**Header <unordered_set> synopsis**

namespace std {
    // 23.4.3, class template unordered_set:
    template <class Value, 
              class Hash = hash<Value>, 
              class Pred = std::equal_to<Value>, 
              class Alloc = std::allocator<Value> > 
    class unordered_set;

    // 23.4.4, class template unordered_multiset:
    template <class Value, 
              class Hash = hash<Value>, 
              class Pred = std::equal_to<Value>, 
              class Alloc = std::allocator<Value> > 
    class unordered_multiset;

template <class Value, class Hash, class Pred, class Alloc>
void swap(unordered_set<Value, Hash, Pred, Alloc>& x, 
          unordered_set<Value, Hash, Pred, Alloc>& y);

template <class Value, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x, 
          unordered_multiset<Value, Hash, Pred, Alloc>& y);

Draft
23.4 Unordered associative containers

23.4.1 Class template unordered_map

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.

An unordered_map satisfies all of the requirements of a container and of an unordered associative container. 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.

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
{
    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 Alloc             allocator_type;

typedef typename allocator_type::pointer                  pointer;

typedef typename allocator_type::const_pointer            const_pointer;

typedef typename allocator_type::reference                reference;

typedef typename allocator_type::const_reference          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
    explicit unordered_map(size_type n = implementation-defined, 
                            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 = implementation-defined, 
                   const hasher& hf = hasher(), 
                   const key_equal& eql = key_equal(), 
                   const allocator_type& a = allocator_type());
const key_equal& eql = key_equal(),
const allocator_type& a = allocator_type());
unordered_map(const unordered_map&);
~unordered_map();
unordered_map& operator=(const unordered_map&);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator begin();
const_iterator begin() const;
itator end();
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;

// modifiers
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
std::pair<iterator, bool> insert(const value_type& obj);
const_iterator insert(const_iterator hint, const value_type& obj);
std::pair<iterator, iterator> insert(const_iterator hint, const value_type& obj);
template <class InputIterator> void insert(InputIterator first, InputIterator last);
iteration erase(const_iterator position);
const_iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
const_iterator erase(const_iterator first, const_iterator last);
void clear();
void swap(unordered_map&);

// 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);

Draft
23.4 Unordered associative containers

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type n);
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n) const;
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;

// hash policy
float load_factor() const;
float max_load_factor() const;
void max_load_factor(float z);
void rehash(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);

23.4.1.1 unordered_map constructors

explicit unordered_map(size_type n = implementation-defined,
    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. max_load_factor() returns 1.0.

Complexity: Constant.

template <class InputIterator>
unordered_map(InputIterator f, InputIterator l,
    size_type n = implementation-defined,
    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). max_load_factor() returns 1.0.

Complexity: Average case linear, worst case quadratic.

23.4.1.2 unordered_map element access

mapped_type& operator[](const key_type& k);

Draft
23.4.1.3 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);

Effects: x.swap(y).

23.4.2 Class template unordered_multimap

An unordered_multimap is an unordered associative container that supports equivalent keys (an unordered_multimap may contain multiple copies of each key value) and that associates values of another type mapped_type with the keys.

An unordered_multimap satisfies all of the requirements of a container and of an unordered associative container. 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>.

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.

Draft
typedef implementation-defined const_iterator;
typedef implementation-defined local_iterator;
typedef implementation-defined const_local_iterator;

// construct/destroy/copy
explicit unordered_multimap(size_type n = implementation-defined,
                           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 = implementation-defined,
                   const hasher& hf = hasher(),
                   const key_equal& eql = key_equal(),
                   const allocator_type& a = allocator_type());
unordered_multimap(const unordered_multimap&);
~unordered_multimap();
unordered_multimap& operator=(const unordered_multimap&);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;

// modifiers
template <class... Args> iterator emplace(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
iterator insert(const value_type& obj);
iterator insert(const_iterator hint, const value_type& obj);
const_iterator insert(const_iterator hint, const value_type& obj);
template <class InputIterator> void insert(InputIterator first, InputIterator last);iterator erase(const_iterator position);
const_iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
const_iterator erase(const_iterator first, const_iterator last);
void clear();
void swap(unordered_multimap&);
// 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;
size_type max_bucket_count() const;
size_type bucket(size_type n) const;
local_iterator begin(size_type n) const;
local_iterator end(size_type n);

// hash policy
float load_factor() const;
float max_load_factor() const;
void max_load_factor(float z);
void rehash(size_type n);

};

#include <unordered_multimap>

23.4.2.1 unordered_multimap constructors

explicit unordered_multimap(size_type n = implementation-defined,
      const hasher& hf = hasher(),
      const key_equal& eql = key_equal(),
      const allocator_type& a = allocator_type());

Effects: Constructs an empty unordered_multimap using the specified hash function, key equality function, and allocator, and using at least n buckets. If n is not provided, the number of buckets is implementation defined. max_load_factor() returns 1.0.

Complexity: Constant.
23.4 Unordered associative containers

Containers library

const allocator_type& a = allocator_type();

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). max_load_factor() returns 1.0.

Complexity: Average case linear, worst case quadratic.

23.4.2.2 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);

Effects: x.swap(y).

23.4.3 Class template unordered_set

An unordered_set is an unordered associative container that supports unique keys (an unordered_set contains at most one of each key value) and in which the elements' keys are the elements themselves.

An unordered_set satisfies all of the requirements of a container and of an unordered associative container. It provides the operations described in the preceding requirements table for unique keys; that is, an unordered_set supports the a_uniq operations in that table, not the a_eq operations. For an unordered_set<Value> the key type and the value type are both Value. 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_set that are not described in one of the requirement tables, or for which there is additional semantic information.

```cpp
template <class Value,
          class Hash = hash<Value>,
          class Pred = std::equal_to<Value>,
          class Alloc = std::allocator<Value> >
class unordered_set
{
public:
  // types
  typedef Value key_type;
  typedef Value value_type;
  typedef Hash hasher;
  typedef Pred key_equal;
  typedef Alloc allocator_type;
  typedef typename allocator_type::pointer pointer;
  typedef typename allocator_type::const_pointer const_pointer;
  typedef typename allocator_type::reference reference;
  typedef typename allocator_type::const_reference 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
explicit unordered_set(size_type n = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());
template <class InputIterator>
unordered_set(InputIterator f, InputIterator l,
 size_type n = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());
unordered_set(const unordered_set&);
~unordered_set();
unordered_set& operator=(const unordered_set&);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;

// modifiers
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
std::pair<iterator, bool> insert(const value_type& obj);
iterator insert(const_iterator hint, const value_type& obj);
const_iterator insert(const_iterator hint, const value_type& obj);
template <class InputIterator> void insert(InputIterator first, InputIterator last);
iterator erase(const_iterator position);
const_iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
const_iterator erase(const_iterator first, const_iterator last);
void clear();
void swap(unordered_set&);

// 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;
size_type max_bucket_count() const;
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;

// hash policy
float load_factor() const;
float max_load_factor() const;
void max_load_factor(float z);
void rehash(size_type n);
};

template <class Value, class Hash, class Pred, class Alloc>
void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
          unordered_set<Value, Hash, Pred, Alloc>& y);

23.4.3.1 unordered_set constructors

explicit unordered_set(size_type n = implementation-defined,
const hasher& hf = hasher(),
const key_equal& eql = key_equal(),
const allocator_type& a = allocator_type());

1 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. max_load_factor() returns 1.0.

2 Complexity: Constant.
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 \([f, l)\). max_load_factor() returns 1.0.

Complexity: Average case linear, worst case quadratic.

### 23.4.3.2 unordered_set swap

```cpp
template <class Value, class Hash, class Pred, class Alloc>
void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
           unordered_set<Value, Hash, Pred, Alloc>& y);
```

**Effects:** \( x.\text{swap}(y) \).

### 23.4.4 Class template unordered_multiset

An unordered_multiset is an unordered associative container that supports equivalent keys (an unordered_multiset may contain multiple copies of the same key value) and in which each element’s key is the element itself.

An unordered_multiset satisfies all of the requirements of a container and of an unordered associative container. It provides the operations described in the preceding requirements table for equivalent keys; that is, an unordered_multiset supports the \( \text{a}_{-}\text{eq} \) operations in that table, not the \( \text{a}_{-}\text{uniq} \) operations. For an \( \text{unordered}\_\text{multiset}<\text{Value}> \) the \text{key type} and the value type are both \text{Value}. The iterator and \text{const_iterator} types are both \text{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.

```cpp
template <class Value, class Hash = hash<Value>, class Pred = std::equal_to<Value>,
          class Alloc = std::allocator<Value> >
class unordered_multiset
{
public:
  // types
  typedef Value key_type;
  typedef Value value_type;
  typedef Hash hasher;
  typedef Pred key_equal;
  typedef Alloc allocator_type;
  typedef typename allocator_type::pointer pointer;
  typedef typename allocator_type::const_pointer const_pointer;
  typedef typename allocator_type::reference reference;
  typedef typename allocator_type::const_reference 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
explicit unordered_multiset(size_type n = implementation-defined,
                            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 = implementation-defined,
                   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& operator=(const unordered_multiset&);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;

// modifiers
template <class... Args> iterator emplace(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
iterator insert(const value_type& obj);
iterator insert(const_iterator hint, const value_type& obj);
const_iterator insert(const_iterator hint, const value_type& obj);
template <class InputIterator> void insert(InputIterator first, InputIterator last);

iterator erase(const_iterator position);
const_iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
const_iterator erase(const_iterator first, const_iterator last);
void clear();
void swap(unordered_multiset&);

// 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;
size_type max_bucket_count() const;
size_type bucket_size(size_type n);
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n) const;
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;

// hash policy
float load_factor() const;
float max_load_factor() const;
void max_load_factor(float z);
void rehash(size_type n);};

template <class Value, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
          unordered_multiset<Value, Hash, Pred, Alloc>& y);

23.4.4.1 unordered_multiset constructors [unord.multiset.cnstr]

explicit unordered_multiset(size_type n = implementation-defined,
                            const hasher& hf = hasher(),
                            const key_equal& eql = key_equal(),
                            const allocator_type& a = allocator_type());

1 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.
max_load_factor() returns 1.0.

2 Complexity: Constant.
template <class InputIterator>
unordered_multiset(InputIterator f, InputIterator l,
  size_type n = implementation-defined,
  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 \([f, l)\). \texttt{max_load_factor()} returns 1.0.

*Complexity:* Average case linear, worst case quadratic.

### 23.4.4.2 unordered_multiset swap

```cpp
3 template <class Value, class Hash, class Pred, class Alloc>
4   void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
5            unordered_multiset<Value, Hash, Pred, Alloc>& y);
6   Effects: x.swap(y);
```

*Effects:* x.swap(y);
Chapter 24  Iterators library

1 This clause describes components that C++ programs may use to perform iterations over containers (clause 23), streams (27.6), and stream buffers (27.5).

2 The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 94.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.1 Requirements</td>
<td></td>
</tr>
<tr>
<td>24.3 Iterator primitives</td>
<td>&lt;iterator&gt;</td>
</tr>
<tr>
<td>24.4 Predefined iterators</td>
<td></td>
</tr>
<tr>
<td>24.5 Stream iterators</td>
<td></td>
</tr>
</tbody>
</table>

### 24.1 Iterator requirements

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 class, enumeration, or built-in 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 \( (\ast i).m \) 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 integral 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 95.

<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 the input and output iterators and can be used whenever either kind is specified; Bidirectional iterators also satisfy all the requirements of the forward iterators and can be used whenever a
24.1 Iterator requirements

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.

Besides its category, a forward, bidirectional, or random access iterator can also be mutable or constant depending on whether the result of the expression \( *i \) behaves as a reference or as a reference to a constant. Constant iterators do not satisfy the requirements for output iterators, and the result of the expression \( *i \) (for constant iterator \( i \)) cannot be used in an expression where an lvalue is required.

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 container. 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 container. \( \text{[Example: After the declaration of an uninitialized pointer } x \text{ (as with int* } x);, x \text{ 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 and the assignment of a non-singular value to an iterator that holds a singular value. In this case 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 the same container.

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 one pointed to by \( i \) and up to but not including the one 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.\(^{261}\)

In the following sections, \( a \) and \( b \) denote values of type \( X \) or const \( X \), \( n \) denotes a value of the difference type \( \text{Distance} \), \( u \), \( tmp \), and \( m \) denote identifiers, \( r \) denotes a value of \( X \& \), \( t \) denotes a value of value type \( T \), \( o \) denotes a value of some type that is writable to the output iterator.

24.1.1 Input iterators

A class or a built-in type \( X \) satisfies the requirements of an input iterator for the value type \( T \) if the following expressions are valid, as shown in Table 96.

In Table 96, 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 \(!==\). \( \text{[Example: the call find(a,b,x) is defined only if the value of a has the property } p \text{ defined as follows: b has property } p \text{ and a value } i \text{ has property } p \text{ if } (\!==x) \text{ or if } (\!==x \text{ and } ++i \text{ has property } p). — end example} \]

---

\(^{261}\)This definition applies to pointers, since pointers are iterators. The effect of dereferencing an iterator that has been invalidated is undefined.

Draft
24.1 Iterator requirements

### 24.1.2 Output iterators

A class or a built-in type `X` satisfies the requirements of an output iterator if `X` is a CopyConstructible (34) and Assignable type (23.1) and also the following expressions are valid, as shown in Table 97.

Table 97: Output iterator requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>operational semantics</th>
<th>assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X(a)</code></td>
<td></td>
<td></td>
<td><code>a = t</code> is equivalent to <code>X(a) = t</code>.</td>
</tr>
<tr>
<td><code>X u(a);</code></td>
<td></td>
<td></td>
<td>note: a destructor is assumed.</td>
</tr>
<tr>
<td><code>X u = a;</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>*r = o</code></td>
<td></td>
<td>result is not used</td>
<td></td>
</tr>
<tr>
<td><code>++r</code></td>
<td><code>X&amp;</code></td>
<td></td>
<td><code>kr == &amp;++r</code>.</td>
</tr>
</tbody>
</table>
24.1 Iterator requirements

Iterator requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>operational semantics</th>
<th>assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>r++</td>
<td>convertible to</td>
<td>{ X tmp = r;</td>
<td>{}</td>
</tr>
<tr>
<td>const X&amp;</td>
<td></td>
<td>++r;</td>
<td>return tmp; }</td>
</tr>
<tr>
<td>*r++ = o</td>
<td></td>
<td>result is not used</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 ostrems as the destination for placing data through the ostream_iterator class as well as with insert iterators and insert pointers. — end note]

24.1.3 Forward iterators

A class or a built-in type X satisfies the requirements of a forward iterator if the following expressions are valid, as shown in Table 98.

---

Table 98: Forward iterator requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>operational semantics</th>
<th>assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>X u;</td>
<td></td>
<td>note: u might have a singular value.</td>
<td></td>
</tr>
<tr>
<td>X()</td>
<td></td>
<td>note: X() might be singular.</td>
<td></td>
</tr>
<tr>
<td>X(a)</td>
<td></td>
<td>a == X(a)</td>
<td></td>
</tr>
<tr>
<td>X u(a);</td>
<td></td>
<td>X u; u = a;</td>
<td>post: u == a.</td>
</tr>
<tr>
<td>X u = a;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a == b</td>
<td>convertible to bool</td>
<td>== is an equivalence relation.</td>
<td></td>
</tr>
<tr>
<td>a != b</td>
<td>convertible to bool</td>
<td>!(a == b)</td>
<td></td>
</tr>
<tr>
<td>r = a</td>
<td>X&amp;</td>
<td>post: r == a</td>
<td></td>
</tr>
<tr>
<td>*a</td>
<td>T&amp; if X is mutable, otherwise const</td>
<td>pre: a is dereferenceable.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>a == b implies *a == *b.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>If X is mutable, *a = t is valid.</td>
<td></td>
</tr>
<tr>
<td>a-&gt;m</td>
<td>U&amp; if X is mutable, otherwise const</td>
<td>pre: (*a).m is well-defined.</td>
<td></td>
</tr>
</tbody>
</table>

Draft
— If \( a \) and \( b \) are equal, then either \( a \) and \( b \) are both dereferenceable or else neither is dereferenceable.

— If \( a \) and \( b \) are both dereferenceable, then \( a == b \) if and only if \( *a \) and \( *b \) are the same object.

2 [Note: The condition 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 the iterator (which applies to output iterators) allows the use of multi-pass one-directional algorithms with forward iterators. — end note]

### 24.1.4 Bidirectional iterators

1 A class or a built-in 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 99.

Table 99: Bidirectional iterator requirements (in addition to forward iterator)

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>operational semantics</th>
<th>assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>(++r)</td>
<td>&amp;X</td>
<td></td>
<td>pre: ( r ) is dereferenceable. post: ( r ) is dereferenceable or ( r ) is past-the-end. ( r == s ) and ( r ) is dereferenceable implies (++r) == (++s). &amp;r == &amp;++r.</td>
</tr>
<tr>
<td>(r++)</td>
<td>convertible to &amp;X</td>
<td>{ ( X ) tmp = ( r ); ++( r ); return tmp; }</td>
<td></td>
</tr>
<tr>
<td>(*r++)</td>
<td>T if ( X ) is mutable, otherwise &amp;&amp;X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Draft
24.1.5 Random access iterators

A class or a built-in 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 100.

Table 100: Random access iterator requirements (in addition to bidirectional iterator)

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>operational semantics</th>
<th>assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r += n</code></td>
<td><code>X&amp;</code></td>
<td>`{ Distance m = n;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (m &gt;= 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>while (m--)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>++r;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>while (m++)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>--r;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>return r; }</td>
<td></td>
</tr>
<tr>
<td><code>a + n</code></td>
<td><code>X</code></td>
<td>`{ X tmp = a;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>return tmp += n; }</td>
<td></td>
</tr>
<tr>
<td><code>n + a</code></td>
<td><code>X&amp;</code></td>
<td>return r += n;</td>
<td></td>
</tr>
<tr>
<td><code>r -= n</code></td>
<td><code>X&amp;</code></td>
<td>`{ X tmp = a;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>return tmp -= n; }</td>
<td></td>
</tr>
<tr>
<td><code>a − n</code></td>
<td><code>X</code></td>
<td>`{ X tmp = a;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>return tmp -= n; }</td>
<td></td>
</tr>
<tr>
<td><code>b − a</code></td>
<td><code>Distance</code></td>
<td>`(a &lt; b) ? distance(a,b) :</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>−distance(b,a) }</td>
<td></td>
</tr>
<tr>
<td><code>a[n]</code></td>
<td>convertible to <code>const T &amp;</code></td>
<td>*(a + n)</td>
<td></td>
</tr>
<tr>
<td><code>a &lt; b</code></td>
<td><code>convertible to bool</code></td>
<td><code>b - a &gt; 0</code></td>
<td><code>&lt;</code> is a total ordering relation</td>
</tr>
<tr>
<td><code>a &gt; b</code></td>
<td><code>convertible to bool</code></td>
<td><code>b &lt; a</code></td>
<td><code>&gt;</code> is a total ordering relation opposite to <code>&lt;.</code></td>
</tr>
<tr>
<td><code>a &gt;= b</code></td>
<td><code>convertible to bool</code></td>
<td><code>!(a &lt; b)</code></td>
<td></td>
</tr>
<tr>
<td><code>a &lt;= b</code></td>
<td><code>convertible to bool</code></td>
<td><code>!(a &gt; b)</code></td>
<td></td>
</tr>
</tbody>
</table>
template<class Category, class T, class Distance = ptrdiff_t,  
    class Pointer = T*, class Reference = T&> struct iterator;

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 {};

// 24.3.4, 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 InputIterator>
    InputIterator next(InputIterator x,
        typename std::iterator_traits<InputIterator>::difference_type n = 1);
  template <class BidirectionalIterator>
    BidirectionalIterator prev(BidirectionalIterator x,
        typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);

// 24.4, predefined iterators:
  template <class Iterator> class reverse_iterator;

  template <class Iterator>
    bool operator==(const reverse_iterator<Iterator>& x,  
        const reverse_iterator<Iterator>& y);
  template <class Iterator>
    bool operator<(const reverse_iterator<Iterator>& x,  
        const reverse_iterator<Iterator>& y);
  template <class Iterator>
    bool operator!=(const reverse_iterator<Iterator>& x,  
        const reverse_iterator<Iterator>& y);
  template <class Iterator>
    bool operator>(const reverse_iterator<Iterator>& x,  
        const reverse_iterator<Iterator>& y);
  template <class Iterator>
    bool operator<=(const reverse_iterator<Iterator>& x,  
        const reverse_iterator<Iterator>& y);
  template <class Iterator>
    bool operator>=(const reverse_iterator<Iterator>& x,  
        const reverse_iterator<Iterator>& y);
template <class Iterator>
  typename reverse_iterator<Iterator>::difference_type operator-(
    const reverse_iterator<Iterator>& x,
    const reverse_iterator<Iterator>& y);

template <class Iterator>
  reverse_iterator<Iterator>
    operator+(n,
    typename reverse_iterator<Iterator>::difference_type n,
    const reverse_iterator<Iterator>& x);

template <class Container> class back_insert_iterator;

template <class Container>
  back_insert_iterator<Container> back_inserter(Container& x);

template <class Container> class front_insert_iterator;

template <class Container>
  front_insert_iterator<Container> front_inserter(Container& x);

template <class Container> class insert_iterator;

template <class Container, class Iterator>
  insert_iterator<Container> inserter(Container& x, Iterator i);

template <class Container> class move_iterator;

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);

template <class Iterator1, class Iterator2>
  typename move_iterator<Iterator>::difference_type operator-(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

template <class Iterator>
  move_iterator<Iterator> operator+(n,
    typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x);

template <class Iterator>
  move_iterator<Iterator> make_move_iterator(const Iterator& i);
To simplify the task of defining iterators, the library provides several classes and functions:

To implement algorithms only in terms of iterators, it is often necessary to determine the value and difference types that correspond to a particular iterator type. Accordingly, it is required that if Iterator is the type of an iterator, the types

iterator_traits<Iterator>::difference_type
iterator_traits<Iterator>::value_type
iterator_traits<Iterator>::iterator_category

be defined as the iterator’s difference type, value type and iterator category, respectively. In addition, the types

iterator_traits<Iterator>::reference
iterator_traits<Iterator>::pointer

shall be defined as the iterator’s reference and pointer types, that is, for an iterator object a, the same type as the type of *a and a->, respectively. In the case of an output iterator, the types

iterator_traits<Iterator>::difference_type
iterator_traits<Iterator>::value_type
iterator_traits<Iterator>::reference
iterator_traits<Iterator>::pointer

may be defined as void.

The template `iterator_traits<Iterator>` is defined as

```cpp
namespace std {
    template<class Iterator> struct iterator_traits {
        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;
    };
}
```

It is specialized for pointers as

```cpp
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;
    };
}
```

and for pointers to const as

```cpp
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;
    };
}
```

[Note: If there is an additional pointer type `__far` such that the difference of two `__far` is of type `long`, an implementation may define]

```cpp
namespace std {
    template<class T> struct iterator_traits<T__far*> {
        typedef long difference_type;
        typedef T value_type;
        typedef T__far* pointer;
        typedef T__far& reference;
        typedef random_access_iterator_tag iterator_category;
    };
}
```
5 [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 ]

24.3.2 Basic iterator [iterator.basic]

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;
    };
}
```

24.3.3 Standard iterator tags [std.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 {};
}
```

Draft
24.3 Iterator primitives

[Example: For a program-defined iterator BinaryTreeIterator, it could be included into the bidirectional iterator category by specializing the iterator_traits template:

```cpp
template<class T> struct iterator_traits<BinaryTreeIterator<T>> {
    typedef std::ptrdiff_t difference_type;
    typedef T value_type;
    typedef T* pointer;
    typedef T& reference;
    typedef bidirectional_iterator_tag iterator_category;
};
```

Typically, however, it would be easier to derive BinaryTreeIterator<T> from iterator<bidirectional_iterator_tag, T, ptrdiff_t, T*, T&>.

— end example ]

3 [Example: If evolve() is well defined for bidirectional iterators, but can be implemented more efficiently for random access iterators, then the implementation is as follows:

```cpp
template <class BidirectionalIterator>
inline void evolve(BidirectionalIterator first, BidirectionalIterator last) {
    evolve(first, last,
    typename iterator_traits<BidirectionalIterator>::iterator_category());
}

template <class BidirectionalIterator>
void evolve(BidirectionalIterator first, BidirectionalIterator last,
            bidirectional_iterator_tag) {
    // ... more generic, but less efficient algorithm
}

template <class RandomAccessIterator>
void evolve(RandomAccessIterator first, RandomAccessIterator last,
            random_access_iterator_tag) {
    // ... more efficient, but less generic algorithm
}
```

— end example ]

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:

```cpp
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 ]

24.3.4 Iterator operations

Since only random access iterators provide + and - operators, the library provides two function templates advance and distance. These function templates use + and - for random access iterators (and are, therefore, constant time for...
them); for input, forward and bidirectional iterators they use ++ to provide linear time implementations.

```cpp
template <class InputIterator, class Distance>
void advance(InputIterator& i, Distance n);
```

**Requires:** \( n \) may be negative only for random access and bidirectional iterators.

**Effects:** Increments (or decrements for negative \( n \)) iterator reference \( i \) by \( n \).

```cpp
template<class InputIterator>
    typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

**Requires:** \( \text{last} \) shall be reachable from \( \text{first} \).

**Effects:** Returns the number of increments or decrements needed to get from \( \text{first} \) to \( \text{last} \).

```cpp
template<class InputIterator>
InputIterator next(InputIterator x,
    typename std::iterator_traits<InputIterator>::difference_type n = 1);
```

**Effects:** Equivalent to \( \text{advance}(x, n); \text{return } x; \)

```cpp
template <class BidirectionalIterator>
BidirectionalIterator prev(BidirectionalIterator x,
    typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);
```

**Effects:** Equivalent to \( \text{advance}(x, -n); \text{return } x; \)

### 24.4 Predefined iterators

#### 24.4.1 Reverse iterators

1 Bidirectional and random access iterators have corresponding reverse iterator adaptors that iterate through the data structure in the opposite direction. They have the same signatures as the corresponding iterators. The fundamental relation between a reverse iterator and its corresponding iterator \( i \) is established by the identity: \( \&\*(\text{reverse_iterator}(i)) = \&\*(i - 1) \).

2 This mapping is dictated by the fact that while there is always a pointer past the end of an array, there might not be a valid pointer before the beginning of an array.

#### 24.4.1.1 Class template reverse_iterator

```cpp
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> {
        protected:
            Iterator current;
        }

    // Implementation...
}
```

Draft
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--();
    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;
};

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);
24.4.1.2 reverse_iterator requirements

1 The template parameter Iterator shall meet all the requirements of a Bidirectional Iterator (24.1.4).

2 Additionally, Iterator shall meet the requirements of a Random Access Iterator (24.1.5) if any of the members operator+ (24.4.1.3.8), operator- (24.4.1.3.10), operator+= (24.4.1.3.9), operator-= (24.4.1.3.11), operator[] (24.4.1.3.12), or the global operators operator< (24.4.1.3.14), operator> (24.4.1.3.16), operator<= (24.4.1.3.18), operator>= (24.4.1.3.17), operator< (24.4.1.3.19) or operator+ (24.4.1.3.20). is referenced in a way that requires instantiation (14.7.1).

24.4.1.3 reverse_iterator operations

24.4.1.3.1 reverse_iterator constructor

reverse_iterator();

1 \textit{Effects}: Default initializes \texttt{current}. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a default constructed iterator of type Iterator.

explicit reverse_iterator(Iterator \texttt{x});

1 \textit{Effects}: Initializes \texttt{current} with \texttt{x}.

2 template <class \texttt{U}> reverse_iterator(const reverse_iterator<\texttt{U}> \&u);

3 \textit{Effects}: Initializes \texttt{current} with \texttt{u.current}.
24.4 Predefined iterators

```cpp
template <class U>
reverse_iterator&
operator=(const reverse_iterator<U>& u);
```

1 Effects: Assigns u.base() to current.
2 Returns: *this.

24.4.1.3.3 Conversion [reverse.iter.conv]

Iterator base() const;  // explicit
1 Returns: current.

24.4.1.3.4 operator* [reverse.iter.op.star]

reference operator*() const;
1 Effects:
   this->tmp = current;
   --this->tmp;
   return *this->tmp;
2 [Note: This operation must use an auxiliary member variable, rather than a temporary variable, to avoid returning a reference that persists beyond the lifetime of its associated iterator. (See 24.1.) The name of this member variable is shown for exposition only. — end note ]

24.4.1.3.5 operator-> [reverse.iter.opref]

pointer operator->() const;
1 Effects:
   return &(operator*());

24.4.1.3.6 operator++ [reverse.iter.op++]

reverse_iterator& operator++();
1 Effects: --current;
2 Returns: *this.

reverse_iterator operator++(int);
3 Effects:
```
reverse_iterator tmp = *this;
--current;
return tmp;

### 24.4.1.3.7 operator--

reverse_iterator& operator--();

1. **Effects:** ++current  
2. **Returns:** *this.

reverse_iterator operator--(int);

3. **Effects:**  
   reverse_iterator tmp = *this;  
   ++current;  
   return tmp;

### 24.4.1.3.8 operator+

reverse_iterator
operator+(typename reverse_iterator<Iterator>::difference_type n) const;

1. **Returns:** reverse_iterator(current-n).

### 24.4.1.3.9 operator+=

reverse_iterator&
operator+=(typename reverse_iterator<Iterator>::difference_type n);

1. **Effects:** current -= n;  
2. **Returns:** *this.

### 24.4.1.3.10 operator-

reverse_iterator
operator-(typename reverse_iterator<Iterator>::difference_type n) const;

1. **Returns:** reverse_iterator(current+n).

### 24.4.1.3.11 operator=*

reverse_iterator&
operator=(typename reverse_iterator<Iterator>::difference_type n);
Effects: current += n;
Returns: *this.

24.4.1.3.12 operator[] [reverse.iter.opindex]

unspecified operator[](
    typename reverse_iterator<Iterator>::difference_type n) const;
Returns: current[-n-1].

24.4.1.3.13 operator== [reverse.iter.op==]

template <class Iterator1, class Iterator2>
bool operator==(const reverse_iterator<Iterator1>& x,
               const reverse_iterator<Iterator2>& y);
Returns: x.current == y.current.

24.4.1.3.14 operator< [reverse.iter.op<]

template <class Iterator1, class Iterator2>
bool operator<(const reverse_iterator<Iterator1>& x,
              const reverse_iterator<Iterator2>& y);
Returns: x.current > y.current.

24.4.1.3.15 operator!= [reverse.iter.op!=]

template <class Iterator1, class Iterator2>
bool operator!=(const reverse_iterator<Iterator1>& x,
               const reverse_iterator<Iterator2>& y);
Returns: x.current != y.current.

24.4.1.3.16 operator> [reverse.iter.op>]

template <class Iterator1, class Iterator2>
bool operator>(const reverse_iterator<Iterator1>& x,
               const reverse_iterator<Iterator2>& y);
Returns: x.current < y.current.

Draft
24.4.1.3.17 operator>=

```
template <class Iterator1, class Iterator2>
bool operator>=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

*Returns:* \(x\.current \leq y\.current\).

24.4.1.3.18 operator<=

```
template <class Iterator1, class Iterator2>
bool operator<=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

*Returns:* \(x\.current \geq y\.current\).

24.4.1.3.19 operator-

```
template <class Iterator1, class Iterator2>
    typename reverse_iterator<Iterator>::difference_type operator-(
        const reverse_iterator<Iterator1>& x,
        const reverse_iterator<Iterator2>& y);
```

*Returns:* \(y\.current - x\.current\).

24.4.1.3.20 operator+

```
template <class Iterator>
    reverse_iterator<Iterator> operator+(
        typename reverse_iterator<Iterator>::difference_type n,
        const reverse_iterator<Iterator>& x);
```

*Returns:* \(reverse\_iterator<Iterator>\.current - n\).

24.4.2 Insert iterators

To make it possible to deal with insertion in the same way as writing into an array, a special kind of iterator adaptors, called *insert iterators*, are provided in the library. With regular iterator classes,

```
while (first != last) *result++ = *first++;
```

causes a range \([\text{first}, \text{last})\) to be copied into a range starting with result. The same code with result being an insert iterator will insert corresponding elements into the container. This device allows all of the copying algorithms in the library to work in the *insert mode* instead of the *regular overwrite mode*. 
An insert iterator is constructed from a container and possibly one of its iterators pointing to where insertion takes place if it is neither at the beginning nor at the end of the container. Insert iterators satisfy the requirements of output iterators. operator* returns the insert iterator itself. The assignment operator=(const T& x) is defined on insert iterators to allow writing into them, it inserts x right before where the insert iterator is pointing. In other words, an insert iterator is like a cursor pointing into the container where the insertion takes place. back_insert_iterator inserts elements at the end of a container, front_insert_iterator inserts elements at the beginning of a container, and insert_iterator inserts elements where the iterator points to in a container. back_inserter, front_inserter, and inserter are three functions making the insert iterators out of a container.

24.4.2.1 Class template back_insert_iterator

```cpp
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<Container>&
                operator=(typename Container::const_reference value);
            back_insert_iterator<Container>&
                operator=(typename Container::value_type&& value);

            back_insert_iterator<Container>& operator*();
            back_insert_iterator<Container>& operator++();
            back_insert_iterator<Container> operator++(int);
        };

    template <class Container>
    back_insert_iterator<Container> back_inserter(Container& x);
}
```

24.4.2.2 back_insert_iterator operations

24.4.2.2.1 back_insert_iterator constructor

```cpp
explicit back_insert_iterator(Container& x);
```

Effects: Initializes container with kx.

24.4.2.2.2 back_insert_iterator::operator=

```cpp
back_insert_iterator<Container>&
            operator=(typename Container::const_reference value);
```

Effects: container->push_back(value);
24.4.2.2.3 back_insert_iterator::operator*

back_insert_iterator<Container>&
operator=(typename Container::value_type&& value);

Effects: container->push_back(std::move(value));

Returns: *this.

24.4.2.2.4 back_insert_iterator::operator++

back_insert_iterator<Container>& operator++();
back_insert_iterator<Container> operator++(int);

Returns: *this.

24.4.2.2.5 back_inserter

template <class Container>
back_insert_iterator<Container> back_inserter(Container& x);

Returns: back_insert_iterator<Container>(x).

24.4.2.3 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<Container>&
        operator=(typename Container::const_reference value);
    front_insert_iterator<Container>&
        operator=(typename Container::value_type&& value);

    front_insert_iterator<Container>& operator*();
    front_insert_iterator<Container>& operator++();
24.4 Predefined iterators

```cpp
front_insert_iterator<Container> operator++(int);
};

template <class Container>
  front_insert_iterator<Container> front_inserter(Container& x);
}

24.4.2.4 front_insert_iterator operations

24.4.2.4.1 front_insert_iterator constructor

explicit front_insert_iterator(Container& x);
1       Effects: Initializes container with &x.

24.4.2.4.2 front_insert_iterator::operator=

front_insert_iterator<Container>&
  operator=(typename Container::const_reference value);
1       Effects: container->push_front(value);
2       Returns: *this.

front_insert_iterator<Container>&
  operator=(typename Container::value_type&& value);
3       Effects: container->push_front(std::move(value));
4       Returns: *this.

24.4.2.4.3 front_insert_iterator::operator*

front_insert_iterator<Container>& operator*();
1       Returns: *this.

24.4.2.4.4 front_insert_iterator::operator++

front_insert_iterator<Container>& operator++();
front_insert_iterator<Container> operator++(int);
1       Returns: *this.

24.4.2.4.5 front_inserter

template <class Container>
  front_insert_iterator<Container> front_inserter(Container& x);
1       Returns: front_insert_iterator<Container>(x).
```

Draft
24.4.2.5 Class template insert_iterator

namespace std {
  template <class Container>
  class insert_iterator :
    public iterator<output_iterator_tag,void,void,void,void> {
    protected:
      Container* container;
      typename Container::iterator iter;

    public:
      typedef Container container_type;
      insert_iterator(Container& x, typename Container::iterator i);
      insert_iterator<Container>&
        operator=(typename Container::const_reference value);
      insert_iterator<Container>&
        operator=(typename Container::value_type&& value);

      insert_iterator<Container>& operator*();
      insert_iterator<Container>& operator++();
      insert_iterator<Container>& operator++(int);
    }

    template <class Container, class Iterator>
    insert_iterator<Container> inserter(Container& x, Iterator i);
  }

24.4.2.6 insert_iterator operations

24.4.2.6.1 insert_iterator constructor

insert_iterator(Container& x, typename Container::iterator i);

Effects: Initializes container with &x and iter with i.

24.4.2.6.2 insert_iterator::operator=

insert_iterator<Container>&
  operator=(typename Container::const_reference value);

Effects:
  iter = container->insert(iter, value);
  ++iter;

Returns: *this.

insert_iterator<Container>&
  operator=(typename Container::value_type&& value);

Effects:

Draft
24.4 Predefined iterators

4

Returns: *this.

24.4.2.6.3 insert_iterator::operator*

insert_iterator<Container>& operator*();

1 Returns: *this.

24.4.2.6.4 insert_iterator::operator++

insert_iterator<Container>& operator++();
insert_iterator<Container>& operator++(int);

1 Returns: *this.

24.4.2.6.5 inserter

template <class Container, class Inserter>
insert_iterator<Container> inserter(Container& x, Inserter i);

1 Returns: insert_iterator<Container>(x,typename Container::iterator(i)).

24.4.3 Move iterators

Class template move_iterator is an iterator adaptor with the same behavior as the underlying iterator except that its dereference operator implicitly converts the value returned by the underlying iterator’s dereference operator to an rvalue reference. Some generic algorithms can be called with move iterators to replace copying with moving.

[Example:

set<string> s;
// ... populate the set 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 ]

24.4.3.1 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 typename iterator_traits<Iterator>::pointer pointer;
typedef typename iterator_traits<Iterator>::value_type value_type;
typedef typename iterator_traits<Iterator>::iterator_category iterator_category;
typedef value_type&& 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);
reference 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);
24.4 Predefined iterators

```cpp
typedef move_iterator<Iterator>::difference_type operator-(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

-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(const Iterator& i);
```

24.4.3.2 move_iterator requirements

The template parameter `Iterator` shall meet the requirements for an Input Iterator (24.1.1). Additionally, if any of the bidirectional or random access traversal functions are instantiated, the template parameter shall meet the requirements for a Bidirectional Iterator (24.1.4) or a Random Access Iterator (24.1.5), respectively.

24.4.3.3 move_iterator operations

24.4.3.3.1 move_iterator constructors

```cpp
move_iterator();

-explicit move_iterator(Iterator i);

-template <class U> move_iterator(const move_iterator<U>& u);
```

Effects: Constructs a `move_iterator`, default initializing `current`.

Effects: Constructs a `move_iterator`, initializing `current` with `i`.

Effects: Constructs a `move_iterator`, initializing `current` with `u.base()`.

Requires: `U` shall be convertible to `Iterator`.

24.4.3.3.2 move_iterator::operator=

```cpp
-template <class U> move_iterator& operator=(const move_iterator<U>& u);
```

Effects: Assigns `u.base()` to `current`.

Requires: `U` shall be convertible to `Iterator`.

24.4.3.3.3 move_iterator conversion

```cpp
Iterator base() const;
```

Returns: `current`.

24.4.3.3.4 move_iterator::operator*

```cpp
reference operator*() const;
```

Draft
Returns: *current, implicitly converted to an rvalue reference.

24.4.3.5 move_iterator::operator->

pointer operator->() const;

Returns: current.

24.4.3.6 move_iterator::operator++

move_iterator& operator++();

Effects: ++current.

Returns: *this.

move_iterator& operator++(int);

Effects:

move_iterator tmp = *this;

++current;

return tmp;

24.4.3.7 move_iterator::operator--

move_iterator& operator--();

Effects: --current.

Returns: *this.

move_iterator& operator--(int);

Effects:

move_iterator tmp = *this;

--current;

return tmp;

24.4.3.8 move_iterator::operator+

move_iterator operator+(difference_type n) const;

Returns: move_iterator(current + n).
24.4 Predefined iterators

24.4.3.3.9 move_iterator::operator+=

move_iterator& operator+=(difference_type n);
1  Effects: current += n.
2  Returns: *this.

24.4.3.3.10 move_iterator::operator-

move_iterator operator-(difference_type n) const;
1  Returns: move_iterator(current - n).

24.4.3.3.11 move_iterator::operator-=

move_iterator& operator-=(difference_type n);
1  Effects: current -= n.
2  Returns: *this.

24.4.3.3.12 move_iterator::operator[]

reference operator[](difference_type n) const;
1  Returns: current[n], implicitly converted to an rvalue reference.

24.4.3.3.13 move_iterator comparisons

template <class Iterator1, class Iterator2>
bool operator==(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
1  Returns: x.base() == y.base().

template <class Iterator1, class Iterator2>
bool operator!=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
2  Returns: !(x == y).

template <class Iterator1, class Iterator2>
bool operator<(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
3  Returns: x.base() < y.base().

template <class Iterator1, class Iterator2>
bool operator<=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
4  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).

24.4.3.3.14 move_iterator non-member functions [move.iter.nonmember]

template <class Iterator1, class Iterator2>
typename move_iterator<Iterator>::difference_type operator-(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

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(const Iterator& i);

Returns: move_iterator<Iterator>(i).

24.5 Stream iterators [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_copy(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 ]

24.5.1 Class template istream_iterator [istream.iterator]

istream_iterator 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 end of stream is reached (operator void*() on the stream returns false), 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 is not defined. For any other iterator value a const T& is returned. The result of operator-> on an end of stream is not defined. For any other iterator value a const T* is returned. It is impossible to store things into istream iterators. The
The practical consequence of this fact is that istream iterators can be used only for one-pass algorithms, which actually makes perfect sense, since for multi-pass algorithms it is always more appropriate to use in-memory data structures.

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.

```cpp
namespace std {
    template <class T, class charT = char, class traits = char_traits<charT>, class Distance = ptrdiff_t>
    class istream_iterator:
    public iterator<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;
        istream_iterator();
        istream_iterator(istream_type& s);
        istream_iterator(const istream_iterator<T,charT,traits,Distance>& x);
        ~istream_iterator();

        const T& operator*() const;
        const T* operator->() const;
        istream_iterator<T,charT,traits,Distance>& operator++();
        istream_iterator<T,charT,traits,Distance> 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);
}

24.5.1.1 istream_iterator constructors and destructor

istream_iterator();

Effects: Constructs the end-of-stream iterator.

istream_iterator(istream_type& s);

Effects: Initializes in_stream with s. value may be initialized during construction or the first time it is referenced.

istream_iterator(const istream_iterator<T,charT,traits,Distance>& x);

Draft
3 Effects: Constructs a copy of \( x \).

`istream_iterator();`

4 Effects: The iterator is destroyed.

### 24.5.1.2 `istream_iterator` operations

```c++
const T& operator*() const;
Returns: \( value \).

const T* operator->() const;
Returns: \&(operator*()).

`istream_iterator<T,charT,traits,Distance>& operator++();`
3 Effects: \( \text{in}\_stream >> \text{value} \).
4 Returns: \*this.

`istream_iterator<T,charT,traits,Distance> operator++(int);`
5 Effects:

\[
\text{in}\_stream >> \text{value};
\]

```c++
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);
6 Returns: \( x.\text{in}\_stream == y.\text{in}\_stream \).

```c++
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);
7 Returns: \! (x == y)
```

### 24.5.2 Class template `ostream_iterator`

`ostream_iterator` writes (using `operator<<`) successive elements onto the output stream from which it was constructed. If it was constructed with `char*` as a constructor argument, this string, called a delimiter string, is written to the stream after every `T` is written. It is not possible to get a value out of the output iterator. Its only use is as an output iterator in situations like

```c++
while (first != last) *result++ = *first++;
```

`ostream_iterator` is defined as:

---

Draft
namespace std {
    template <class T, class charT = char, class traits = char_traits<charT> >
    class ostream_iterator {
        public iterator<output_iterator_tag, void, void, void, void> {
            public:
                typedef charT char_type;
                typedef traits traits_type;
                typedef basic_ostream<charT,traits> ostream_type;
                ostream_iterator(ostream_type& s);
                ostream_iterator(ostream_type& s, const charT* delimiter);
                ostream_iterator(const ostream_iterator<T,charT,traits>& x);
                ~ostream_iterator();
                ostream_iterator<T,charT,traits>& operator=(const T& value);
                ostream_iterator<T,charT,traits>& operator*();
                ostream_iterator<T,charT,traits>& operator++();
                ostream_iterator<T,charT,traits>& operator++(int);
        }
    }
}

24.5.2.1 ostream_iterator constructors and destructor

    ostream_iterator(ostream_type& s);
    1 Effects: Initializes out_stream with s and delim with null.

    ostream_iterator(ostream_type& s, const charT* delimiter);
    2 Effects: Initializes out_stream with s and delim with delimiter.

    ostream_iterator(const ostream_iterator<T>& x);
    3 Effects: Constructs a copy of x.

    ~ostream_iterator();
    4 Effects: The iterator is destroyed.

24.5.2.2 ostream_iterator operations

    ostream_iterator<T>& operator=(const T& value);
    1 Effects:

        *out_stream << value;
        if(delim != 0) *out_stream << delim;
        return (*this);
24.5 Stream iterators

namespace std {

    template<class charT, class traits = char_traits<charT>>
    class istreambuf_iterator
    : public iterator<input_iterator_tag, charT, typename traits::off_type, charT*, 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

public:
    istreambuf_iterator() throw();
    istreambuf_iterator(istream_type& s) throw();
    istreambuf_iterator(streambuf_type* s) throw();
    istreambuf_iterator(const proxy& p) throw();
    charT operator*() const;
    istreambuf_iterator<charT, traits>& operator++();
    proxy operator++(int);
    bool equal(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);
}
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.

2 The result of `operator*()` on an end of stream is undefined. For any other iterator value a char_type value is returned. It is impossible to assign a character via an input iterator.

3 Note that in the input iterators, `++` operators are not *equality preserving*, that is, `i == j` does not guarantee at all that `++i == ++j`. Every time `++` is evaluated a new value is used.

4 The practical consequence of this fact is that an `istreambuf_iterator` object can be used only for *one-pass algorithms*. Two end of stream iterators are always equal. An end of stream iterator is not equal to a non-end of stream iterator.

### 24.5.3.1 Class template `istreambuf_iterator::proxy`  

```cpp
namespace std {
    template <class charT, class traits = char_traits<charT> >
    class istreambuf_iterator<charT, traits>::proxy {
        charT keep_;
        basic_streambuf<charT,traits>* sbuf_;
        proxy(charT c,  
              basic_streambuf<charT,traits>* sbuf)  
              : keep_(c), sbuf_(sbuf) {}
        public:
            charT operator*() { return keep_; }
    };
}
```

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.

### 24.5.3.2 `istreambuf_iterator` constructors

`istreambuf_iterator()` throw();

1 *Effects:* Constructs the end-of-stream iterator.

`istreambuf_iterator(basic_istream<charT,traits>& s)` throw();

`istreambuf_iterator(basic_streambuf<charT,traits>* s)` throw();

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)` throw();

3 *Effects:* Constructs a `istreambuf_iterator<>` that uses the `basic_streambuf<>` object pointed to by the `proxy` object's constructor argument `p`.

Draft
24.5.3.3  `istreambuf_iterator::operator*`  

```cpp
charT operator*() const
```

Returns: The character obtained via the streambuf member `sbuf_`->`sgetc()`.

24.5.3.4  `istreambuf_iterator::operator++`  

```cpp
istreambuf_iterator<charT,traits>&
istreambuf_iterator<charT,traits>::operator++();
```

Effects: `sbuf_`->`sbumpc()`.

Returns: `*this`.

proxy `istreambuf_iterator<charT,traits>::operator++(int);`

Returns: `proxy(sbuf_`->`sbumpc(), sbuf_)`.

24.5.3.5  `istreambuf_iterator::equal`  

```cpp
bool equal(istreambuf_iterator<charT,traits>& b) const;
```

Returns: `true` if and only if both iterators are at end-of-stream, or neither is at end-of-stream, regardless of what streambuf object they use.

24.5.3.6  `operator==`  

```cpp
template <class charT, class traits>
bool operator==(const istreambuf_iterator<charT,traits>& a, const istreambuf_iterator<charT,traits>& b);`n```

Returns: `a` . `equal(b)`.

24.5.3.7  `operator!=`  

```cpp
template <class charT, class traits>
bool operator!=(const istreambuf_iterator<charT,traits>& a, const istreambuf_iterator<charT,traits>& b);`n```

Returns: `!a` . `equal(b)`.

24.5.4  Class template `ostreambuf_iterator`  

```cpp
namespace std {
    template <class charT, class traits = char_traits<charT> >
    class ostreambuf_iterator:
        public iterator<output_iterator_tag, void, void, void, void> {
```
24.5 Stream iterators

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) throw();
    ostreambuf_iterator(streambuf_type* s) throw();
    ostreambuf_iterator& operator=(charT c);
    ostreambuf_iterator& operator*();
    ostreambuf_iterator& operator++();
    ostreambuf_iterator& operator++(int);
    bool failed() const throw();

private:
    // streambuf_type* sbuf_;  
    exposition only

};

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.

24.5.4.1 ostreambuf_iterator constructors

ostreambuf_iterator(ostream_type& s) throw();
1    Requires: s.rdbuf() is not null.
2    Effects: :sbuf_(s.rdbuf()) {}. 

ostreambuf_iterator(streambuf_type* s) throw();
3    Requires: s is not null.
4    Effects: :sbuf_(s) {} .

24.5.4.2 ostreambuf_iterator operations

ostreambuf_iterator<charT,traits>&
operator=(charT c);
1    Effects: If failed() yields false, calls sbuf_->sputc(c); otherwise has no effect.
2    Returns: *this.

ostreambuf_iterator<charT,traits>& operator*();
3    Returns: *this.
ostreambuf_iterator<charT,traits>& operator++();
ostreambuf_iterator<charT,traits>& operator++(int);

Returns: *this.

bool failed() const throw();

Returns: true if in any prior use of member operator=, the call to sbuf_->sputc() returned traits::eof(); or false otherwise.
Chapter 25  Algorithms library

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 101.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.1 Non-modifying sequence operations</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>25.2 Mutating sequence operations</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>25.3 Sorting and related operations</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>25.4 C library algorithms</td>
<td>&lt;cstdlib&gt;</td>
</tr>
</tbody>
</table>

Header <algorithm> synopsis

namespace std {

    // 25.1. non-modifying sequence operations:
    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 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);

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 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);

template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
OutputIterator result);

template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(
BidirectionalIterator1 first, BidirectionalIterator1 last,
BidirectionalIterator2 result);

template<class InputIterator, class OutputIterator>
OutputIterator move(InputIterator first, InputIterator last,
OutputIterator result);

template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator move_backward(
BidirectionalIterator1 first, BidirectionalIterator1 last,
BidirectionalIterator2 result);

template<class T> void swap(T& a, T& b);

template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(ForwardIterator1 first1,
ForwardIterator1 last1, ForwardIterator2 first2);

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);

Draft
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);

void fill(ForwardIterator first, ForwardIterator last, const T& value);

template<class OutputIterator, class Size, class T>  
void 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>  
void 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);

void reverse(BidirectionalIterator first, BidirectionalIterator last);

template<class BidirectionalIterator>  
void reverse(BidirectionalIterator first, BidirectionalIterator last);

template<class BidirectionalIterator, class OutputIterator>  
OutputIterator unique_copy(BidirectionalIterator first, BidirectionalIterator last,  
    OutputIterator result);
OutputIterator reverse_copy(BidirectionalIterator first,
                           BidirectionalIterator last,
                           OutputIterator result);

template<class ForwardIterator>
void rotate(ForwardIterator first, ForwardIterator middle,
           ForwardIterator last);

template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy
  (ForwardIterator first, ForwardIterator middle,
   ForwardIterator last, OutputIterator result);

template<class RandomAccessIterator>
void random_shuffle(RandomAccessIterator first,
                    RandomAccessIterator last);

template<class RandomAccessIterator, class RandomNumberGenerator>
void random_shuffle(RandomAccessIterator first,
                    RandomAccessIterator last,
                    RandomNumberGenerator& rand);

template<class RandomAccessIterator, class UniformRandomNumberGenerator>
void random_shuffle(RandomAccessIterator first,
                    RandomAccessIterator last,
                    UniformRandomNumberGenerator& rand);

// 25.2.13, partitions:

// 25.3, sorting and related operations:
// 25.3.1, 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 last,
                 Compare comp);
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);

// 25.3.3, 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);

// 25.3.4, 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);

// 25.3.5, 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);

template<class InputIterator1, class InputIterator2, class OutputIterator,
         class Compare>
  OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2,
                            OutputIterator result, Compare comp);
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);

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);

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);

// 25.3.6, 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);

// 25.3.7, minimum and maximum:
template<class T> const T& min(const T& a, const T& b);

template<class T, class Compare>
const T& min(const T& a, const T& b, Compare comp);

template<class T> const T& max(const T& a, const T& b);

template<class T, class Compare>
const T& max(const T& a, const T& b, Compare comp);

template<class T> pair<const T&, const T&> minmax(const T& a, const T& b);

template<class T, class Compare>
pair<const T&, const T&> minmax(const T& a, const T& b, 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>
Draft
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.

Throughout this clause, the names of template parameters are used to express type requirements. If an algorithm’s template parameter is `InputIterator`, `InputIterator1`, or `InputIterator2`, the actual template argument shall satisfy the requirements of an input iterator (24.1.1). If an algorithm’s template parameter is `OutputIterator`, `OutputIterator1`, or `OutputIterator2`, the actual template argument shall satisfy the requirements of an output iterator (24.1.2). If an algorithm’s template parameter is `ForwardIterator`, `ForwardIterator1`, or `ForwardIterator2`, the actual template argument shall satisfy the requirements of a forward iterator (24.1.3). If an algorithm’s template parameter is `BidirectionalIterator`, `BidirectionalIterator1`, or `BidirectionalIterator2`, the actual template argument shall satisfy the requirements of a bidirectional iterator (24.1.4). If an algorithm’s template parameter is `RandomAccessIterator`, `RandomAccessIterator1`, or `RandomAccessIterator2`, the actual template argument shall satisfy the requirements of a random-access iterator (24.1.5).

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 (24.1). [Note: this requirement does not affect arguments that are declared as `OutputIterator`, `OutputIterator1`,
or OutputIterator2, because output iterators must always be mutable. —end note]

Both in-place and copying versions are provided for certain algorithms.262 When such a version is provided for algorithm it is called algorithm_copy. Algorithms that take predicates end with the suffix _if (which follows the suffix _copy).

The Predicate parameter is used whenever an algorithm expects a function object 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 if (pred(*first)) {...}. The function object pred shall not apply any non-constant function through the dereferenced iterator. This function object may be a pointer to function, or an object of a type with an appropriate function call operator.

The BinaryPredicate parameter is used whenever an algorithm expects a function object that when applied to the result of dereferencing two corresponding iterators or to dereferencing an iterator and type T when T is part of the signature returns a value testable as true. In other words, if an algorithm takes BinaryPredicate binary_pred as its argument and first1 and first2 as its iterator arguments, it should work correctly in the construct if (binary_pred(*first1, *first2)) {...}. BinaryPredicate always takes the first iterator type as its first argument, that is, in those cases when T value is part of the signature, it should work correctly in the context of if (binary_pred(*first1, value)) {...}. binary_pred shall not apply any non-constant function through the dereferenced iterators.

[Note: Unless otherwise specified, algorithms that take function objects as arguments are permitted to copy those function objects freely. Programmers for whom object identity is important should consider using a wrapper class that points to a noncopied implementation object such as reference_wrapper<T> (20.5.5), or some equivalent solution. —end note]

When the description of an algorithm gives an expression such as *first == value for a condition, the expression shall evaluate to either true or false in boolean contexts.

In the description of the algorithms operators + and - are used for some of the iterator categories for which they do not have to be defined. In these cases the semantics of a+n is the same as that of

```c
{ X tmp = a;
    advance(tmp, n);
    return tmp;
}
```

and that of b-a is the same as of

```c
return distance(a, b);
```

25.1 Non-modifying sequence operations [alg.nonmodifying]

25.1.1 For each [alg.foreach]

262] 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 Non-modifying sequence operations

```cpp
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);
```

**Effects:** Applies \( f \) to the result of dereferencing every iterator in the range \([first, last)\), starting from \( first \) and proceeding to \( last - 1 \). [Note: If the type of \( first \) satisfies the requirements of a mutable iterator, \( f \) may apply nonconstant functions through the dereferenced iterator. — end note]

**Returns:** \( f \).

**Complexity:** Applies \( f \) exactly \( last - first \) times.

**Remarks:** If \( f \) returns a result, the result is ignored.

### 25.1.2 Find

```cpp
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value);
```

```cpp
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);
```

**Returns:** The first iterator \( i \) in the range \([first, last)\) for which the following corresponding conditions hold:

\[
*i == value,\ pred(*i) != false.
\]

Returns \( last \) if no such iterator is found.

**Complexity:** At most \( last - first \) applications of the corresponding predicate.

### 25.1.3 Find End

```cpp
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);
```

```cpp
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);
```

**Effects:** Finds a subsequence of equal values in a sequence.

**Returns:** The last iterator \( i \) in the range \([first1, last1 - (last2 - first2))\) such that for any non-negative integer \( n < (last2 - first2) \), the following corresponding conditions hold:

\[
*i + n == *({first2 + n), pred(*i + n), (*first2 + n)) != false.
\]

Returns \( last1 \) if no such iterator is found.

**Complexity:** At most \((last2 - first2) \cdot (last1 - first1 - (last2 - first2)) + 1\) applications of the corresponding predicate.
25.1.4 Find First

\texttt{find\_first\_of(InputIterator }first1, InputIterator \ last1, \ ForwardIterator first2, ForwardIterator last2);\texttt{)}

\texttt{template<class InputIterator, class ForwardIterator, class BinaryPredicate>}
\texttt{InputIterator find\_first\_of(InputIterator }first1, InputIterator \ last1, \ ForwardIterator first2, ForwardIterator last2, BinaryPredicate \ pred);\texttt{)}

1 \textbf{Effects:} Finds an element that matches one of a set of values.

2 \textbf{Returns:} The first iterator \textit{i} in the range \([first1, last1)\) such that for some iterator \textit{j} in the range \([first2, last2)\) the following conditions hold: \textit{i} == \textit{j}, \textit{pred}(\textit{i}, \textit{j}) != false. Returns \textit{last1} if no such iterator is found.

3 \textbf{Complexity:} At most \((last1-first1) \times (last2-first2)\) applications of the corresponding predicate.

25.1.5 Adjacent find

\texttt{adjacent\_find(ForwardIterator first, ForwardIterator last);}\texttt{)}

\texttt{template<class ForwardIterator, class BinaryPredicate>}
\texttt{ForwardIterator adjacent\_find(ForwardIterator }first, ForwardIterator \ last, BinaryPredicate \ pred);\texttt{)}

1 \textbf{Returns:} The first iterator \textit{i} such that both \textit{i} and \textit{i} + 1 are in the range \([first, last)\) for which the following corresponding conditions hold: \textit{i} == *(\textit{i} + 1), \textit{pred}(\textit{i}, *(\textit{i} + 1)) != false. Returns \textit{last} if no such iterator is found.

2 \textbf{Complexity:} For a nonempty range, exactly min((\textit{i} - \textit{first}) + 1, (\textit{last} - \textit{first}) - 1) applications of the corresponding predicate, where \textit{i} is adjacent\_find's return value.

25.1.6 Count

\texttt{template<class InputIterator, class T>}
\texttt{typename iterator_traits<InputIterator>::difference\_type count(InputIterator }first, InputIterator \ last, const T& value);\texttt{)}

\texttt{template<class InputIterator, class Predicate>}
\texttt{typename iterator_traits<InputIterator>::difference\_type count\_if(InputIterator }first, InputIterator \ last, Predicate \ pred);\texttt{)}

1 \textbf{Effects:} Returns the number of iterators \textit{i} in the range \([first, last)\) for which the following corresponding conditions hold: \textit{i} == value, \textit{pred}(\textit{i}) != false.
25.1 Non-modifying sequence operations

Complexity: Exactly last \( - \) first applications of the corresponding predicate.

25.1.7 Mismatch

\[
\text{template<class InputIterator1, class InputIterator2>}
\]

\[
\text{pair<InputIterator1, InputIterator2>}
\]

\[
\text{mismatch(InputIterator1 first1, InputIterator1 last1,}
\]

\[
\text{InputIterator2 first2);
\]

\[
\text{template<class InputIterator1, class InputIterator2,}
\]

\[
\text{class BinaryPredicate>}
\]

\[
\text{pair<InputIterator1, InputIterator2>}
\]

\[
\text{mismatch(InputIterator1 first1, InputIterator1 last1,}
\]

\[
\text{InputIterator2 first2, BinaryPredicate pred);}
\]

\[
\text{Returns: A pair of iterators i and j such that j == first2 + (i - first1) and i is the first iterator in the}
\]

\[
\text{range [first1, last1) for which the following corresponding conditions hold:}
\]

\[
!(i == *(first2 + (i - first1)))
\]

\[
pred(*i, *(first2 + (i - first1))) == false
\]

\[
\text{Returns the pair last1 and first2 + (last1 - first1) if such an iterator i is not found.}
\]

Complexity: At most last \( - \) first1 applications of the corresponding predicate.

25.1.8 Equal

\[
\text{template<class InputIterator1, class InputIterator2>}
\]

\[
\text{bool equal(InputIterator1 first1, InputIterator1 last1,}
\]

\[
\text{InputIterator2 first2);}
\]

\[
\text{template<class InputIterator1, class InputIterator2,}
\]

\[
\text{class BinaryPredicate>}
\]

\[
\text{bool equal(InputIterator1 first1, InputIterator1 last1,}
\]

\[
\text{InputIterator2 first2, BinaryPredicate pred);}
\]

\[
\text{Returns: true if for every iterator i in the range [first1, last1) the following corresponding conditions hold:}
\]

\[
i == *(first2 + (i - first1)), pred(*i, *(first2 + (i - first1))) != false.
\]

\[
\text{Otherwise, returns false.}
\]

Complexity: At most last \( - \) first1 applications of the corresponding predicate.

25.1.9 Search

\[
\text{template<class ForwardIterator1, class ForwardIterator2>}
\]

\[
\text{ForwardIterator1}
\]

\[
\text{search(ForwardIterator1 first1, ForwardIterator1 last1,}
\]

\[
\text{ForwardIterator2 first2, ForwardIterator2 last2);}
\]
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
ForwardIterator1
search(ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2,
    BinaryPredicate pred);

Effects: Finds a subsequence of equal values in a sequence.
Returns: The first iterator \(i\) in the range \([first1, last1 - (last2-first2))\) such that for any non-negative integer \(n\) less than \(last2 - first2\) the following corresponding conditions hold: \(*(i + n) == *(first2 + n)\), \(pred(*(i + n), *(first2 + n)) != false\). Returns \(last1\) if no such iterator is found.
Complexity: At most \((last1 - first1) * (last2 - first2)\) applications of the corresponding predicate.

\[
template<class ForwardIterator, class Size, class T>
ForwardIterator
search_n(ForwardIterator first, ForwardIterator last, Size count,
    const T& value);
\]

Requires: The type \(Size\) is 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 \([first, last-count)\) such that for any non-negative integer \(n\) less than \(count\) the following corresponding conditions hold: \(*(i + n) == value\), \(pred(*(i + n), value) != false\). Returns \(last\) if no such iterator is found.
Complexity: At most \((last - first) * count\) applications of the corresponding predicate if \(count\) is positive, or 0 otherwise.

25.2 Mutating sequence operations [alg.modifying.operations]

25.2.1 Copy [alg.copy]

\[
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
    OutputIterator result);
\]

Effects: Copies elements in the range \([first, last)\) into the range \([result, result + (last - first))\) starting from \(first\) and proceeding to \(last\). For each non-negative integer \(n < (last-first)\), performs \(*(result + n) = *(first + n)\).
Returns: \(result + (last - first)\).
Requires: \(result\) shall not be in the range \([first, last)\).
25.2 Mutating sequence operations

Complexity: Exactly last - first assignments.

```cpp
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.

25.2.2 Move

```cpp
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)).

Requires: result shall not be in the range [first, last).

Returns: result + (last - first).

Complexity: Exactly last - first move assignments.

```cpp
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 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.

---

263) ```cpp
copy_backward should be used instead of copy when last is in the range [result - (last - first), result).
```

264) ```cpp
move_backward should be used instead of move when last is in the range [result - (last - first), result).
```
25.2.3 Swap

\[\text{template<class } T\text{> void } \text{swap}(T & a, T & b);\]

1. **Requires:** Type T shall be MoveConstructible (33) and MoveAssignable (35).
2. **Effects:** Exchanges values stored in two locations.

\[\text{template<class ForwardIterator1, class ForwardIterator2> ForwardIterator2 }\]
\[\text{swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,}\]
\[\text{ForwardIterator2 first2);}\]

3. **Effects:** For each non-negative integer \(n < (last1 - first1)\) performs: \(\text{swap}(\ast(first1 + n), \ast(first2 + n))\).
4. **Requires:** The two ranges \([first1, last1]\) and \([first2, first2 + (last1 - first1)]\) shall not overlap. The type of \(\ast first1\) shall be the same as the type of \(\ast first2\) and that type shall satisfy the Swappable requirements (37).
5. **Returns:** \(first2 + (last1 - first1)\).
6. **Complexity:** Exactly \(last1 - first1\) swaps.

\[\text{template<class ForwardIterator1, class ForwardIterator2> }\]
\[\text{void iter_swap(ForwardIterator1 a, ForwardIterator2 b);}\]

7. **Effects:** \(\text{swap}(\ast a, \ast b)\).
8. **Requires:** The type of \(\ast a\) shall be the same as the type of \(\ast b\) and that type shall satisfy the Swappable requirements (37).

25.2.4 Transform

\[\text{template<class InputIterator, class OutputIterator,}\]
\[\text{class UnaryOperation> OutputIterator}\]
\[\text{transform(InputIterator first, InputIterator last,}\]
\[\text{OutputIterator result, UnaryOperation op);}\]

\[\text{template<class InputIterator1, class InputIterator2,}\]
\[\text{class OutputIterator, class BinaryOperation> OutputIterator}\]
\[\text{transform(InputIterator1 first1, InputIterator1 last1,}\]
\[\text{InputIterator2 first2, OutputIterator result,}\]
\[\text{BinaryOperation binary_op);}\]

1. **Effects:** Assigns through every iterator \(i\) in the range \([result, result + (last1 - first1)]\) a new corresponding value equal to \(op(\ast(first1 + (i - result)), \ast(first2 + (i - result)))\).
25.2 Mutating sequence operations

Requires: \textit{op} and \textit{binary\_op} shall not invalidate iterators or subranges, or modify elements in the ranges \([\textit{first1},\textit{last1}], [\textit{first2},\textit{first2} + (\textit{last1} - \textit{first1})], \) and \([\textit{result},\textit{result} + (\textit{last1} - \textit{first1})] \).\footnote{265}\textsuperscript{265}

Returns: \textit{result} + (\textit{last1} - \textit{first1}).

Complexity: Exactly \textit{last1} - \textit{first1} applications of \textit{op} or \textit{binary\_op}.

Remarks: \textit{result} may be equal to \textit{first} in case of unary transform, or to \textit{first1} or \textit{first2} in case of binary transform.

25.2.5 Replace \footnote{265}\textsuperscript{265}

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

Requires: The expression *\textit{first} = \textit{new\_value} must be valid.

Effects: Substitutes elements referred by the iterator \textit{i} in the range \([\textit{first},\textit{last})\) with \textit{new\_value}, when the following corresponding conditions hold: *\textit{i} == \textit{old\_value}, \textit{pred}(*\textit{i}) != false.

Complexity: Exactly \textit{last} - \textit{first} applications of the corresponding predicate.

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

Requires: The results of the expressions *\textit{first} and \textit{new\_value} shall be writable to the \textit{result} output iterator. The ranges \([\textit{first},\textit{last})\) and \([\textit{result},\textit{result} + (\textit{last} - \textit{first})]\) shall not overlap.

Effects: Assigns to every iterator \textit{i} in the range \([\textit{result},\textit{result} + (\textit{last} - \textit{first})]\) either \textit{new\_value} or *(\textit{first} + (\textit{i} - \textit{result})) depending on whether the following corresponding conditions hold:

\begin{itemize}
  \item *(\textit{first} + (\textit{i} - \textit{result})) == \textit{old\_value}
  \item \textit{pred}(*(\textit{first} + (\textit{i} - \textit{result}))) != false
\end{itemize}

Returns: \textit{result} + (\textit{last} - \textit{first}).

\footnote{265}\textsuperscript{265}The use of fully closed ranges is intentional.
25.2.6 Fill [alg.fill]

\[
\text{template<class ForwardIterator, class T>}
\text{void fill(ForwardIterator first, ForwardIterator last, const T& value);}\]

\[
\text{template<class OutputIterator, class Size, class T>}
\text{void fill_n(OutputIterator first, Size n, const T& value);}\]

1 \text{Requires: The expression value shall be writable to the output iterator. The type Size shall be convertible to an integral type (4.7, 12.3).}

2 \text{Effects: The first algorithm assigns value through all the iterators in the range [first, last). The second algorithm assigns value through all the iterators in the range [first, first + n) if n is positive, otherwise it does nothing.}

3 \text{Complexity: Exactly last - first, n, or 0 assignments, respectively.}

25.2.7 Generate [alg.generate]

\[
\text{template<class ForwardIterator, class Generator>}
\text{void generate(ForwardIterator first, ForwardIterator last, Generator gen);}\]

\[
\text{template<class OutputIterator, class Size, class Generator>}
\text{void generate_n(OutputIterator first, Size n, Generator gen);}\]

1 \text{Effects: The first algorithm invokes the function object gen and assigns the return value of gen through all the iterators in the range [first, last). The second algorithm invokes the function object gen and assigns the return value of gen through all the iterators in the range [first, first + n) if n is positive, otherwise it does nothing.}

2 \text{Requires: gen takes no arguments, Size is convertible to an integral type (4.7, 12.3).}

3 \text{Complexity: Exactly last - first, n, or 0 invocations of gen and assignments, respectively.}

25.2.8 Remove [alg.remove]

\[
\text{template<class ForwardIterator, class T>}
\text{ForwardIterator remove(ForwardIterator first, ForwardIterator last, const T& value);}\]

\[
\text{template<class ForwardIterator, class Predicate>}
\text{ForwardIterator remove_if(ForwardIterator first, ForwardIterator last, Predicate pred);}\]

1 \text{Requires: The type of *first shall satisfy the MoveAssignable requirements (Table 35).}
25.2 Mutating sequence operations

Effects: Eliminates all the elements referred to by iterator \(i\) in the range \([\textit{first}, \textit{last})\) for which the following corresponding conditions hold: \(*i == \textit{value}, \textit{pred}(\*i) \neq \text{false}\).

Returns: The end of the resulting range.

Remarks: Stable.

Complexity: Exactly \(\textit{last} - \textit{first}\) applications of the corresponding predicate.

```cpp
template<class InputIterator, class OutputIterator, class T>
OutputIterator
remove_copy(InputIterator \textit{first}, InputIterator \textit{last},
            OutputIterator \textit{result}, const T& \textit{value});
```

```cpp
template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator
remove_copy_if(InputIterator \textit{first}, InputIterator \textit{last},
               OutputIterator \textit{result}, Predicate \textit{pred});
```

Requires: Type \(T\) is EqualityComparable (31). The ranges \([\textit{first}, \textit{last})\) and \([\textit{result}, \textit{result} + (\textit{last} - \textit{first}))\) shall not overlap.

Effects: Copies all the elements referred to by the iterator \(i\) in the range \([\textit{first}, \textit{last})\) for which the following corresponding conditions do not hold: \(*i == \textit{value}, \textit{pred}(\*i) = \text{false}\).

Returns: The end of the resulting range.

Remarks: Exactly \(\textit{last} - \textit{first}\) applications of the corresponding predicate.

25.2.9 Unique

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 \([\textit{first} + 1, \textit{last})\) for which the following conditions hold: 
\(*(i - 1) == \*i\) or \(\textit{pred}(*(i - 1), \*i) \neq \text{false}\).

Requires: The comparison function shall be an equivalence relation. The type of \(\*\textit{first}\) shall satisfy the MoveAssignable requirements (Table 35).

Returns: The end of the resulting range.

Complexity: For nonempty ranges, exactly \((\textit{last} - \textit{first}) - 1\) applications of the corresponding predicate.

```cpp
template<class InputIterator, class OutputIterator>
OutputIterator
unique(InputIterator \textit{first}, InputIterator \textit{last});
```

```cpp
template<class InputIterator, class BinaryPredicate>
InputIterator
unique(InputIterator \textit{first}, InputIterator \textit{last},
       BinaryPredicate \textit{pred});
```
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 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 (34) and CopyAssignable (table 36). 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.

25.2.10 Reverse \[alg.reverse\]

template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);

Effects: For each non-negative integer \(i <= (last - first)/2\), applies \(\text{iter_swap}\) to all pairs of iterators \(first + i, (last - i) - 1\).

Requires: The type of \(*first\) shall satisfy the Swappable requirements (37).

Complexity: Exactly \((last - first)/2\) swaps.

template<class BidirectionalIterator, class OutputIterator>
OutputIterator
reverse_copy(BidirectionalIterator first, BidirectionalIterator last, OutputIterator result);

Effects: Copies the range \([first, last)\) to the range \([result, result+(last-first))\) such that for any non-negative integer \(i < (last - first)\) the following assignment takes place: \(*(result + (last - first) - i) = *(first + i)\).

Requires: The ranges \([first, last)\) and \([result, result+(last-first))\) shall not overlap.

Returns: \(result + (last - first)\).

Complexity: Exactly \(last - first\) assignments.

25.2.11 Rotate \[alg.rotate\]

Draft
25.2 Mutating sequence operations

```cpp
template<class ForwardIterator>
void rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);
```

**Effects:** For each non-negative integer \( i < (last - first) \), places the element from the position \( first + i \) into position \( first + (i + (last - middle)) \% (last - first) \).

**Remarks:** This is a left rotate.

**Requires:** \([first, middle)\) and \([middle, last)\) are valid ranges. The type of \(*first\) shall satisfy the Swappable requirements (37), the MoveConstructible requirements (Table 33), and the MoveAssignable requirements (Table 35).

**Complexity:** At most \( last - first \) swaps.

```cpp
template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle, ForwardIterator last, OutputIterator result);
```

**Effects:** Copies the range \([first, last)\) to the range \([result, result + (last - first))\) such that for each non-negative integer \( i < (last - first) \) the following assignment takes place: \(*(result + i) = *(first + (i + (middle - first)) \% (last - first))\).

**Returns:** \( result + (last - first) \).

**Requires:** The ranges \([first, last)\) and \([result, result + (last - first))\) shall not overlap.

**Complexity:** Exactly \( last - first \) assignments.

### 25.2.12 Random shuffle

```cpp
template<class RandomAccessIterator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class RandomNumberGenerator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last, RandomNumberGenerator&& rand);
```

```cpp
template<class RandomAccessIterator, class UniformRandomNumberGenerator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last, UniformRandomNumberGenerator& rand);
```

**Effects:** Shuffles the elements in the range \([first, last)\) with uniform distribution.

**Requires:** The type of \(*first\) shall satisfy the Swappable requirements (37).

**Complexity:** Exactly \( (last - first) - 1 \) swaps.
4 Remarks: The underlying source of random numbers for the first form of the function is implementation-defined. An implementation may use the rand function from the standard C library. The second form of the function takes a random number generating function object rand such that if n is an argument for rand, with a positive value, that has type iterator_traits<RandomAccessIterator>::difference_type, then rand(n) returns a randomly chosen value, which lies in the interval [0,n), and which has a type that is convertible to iterator_traits<RandomAccessIterator>::difference_type. The third form of the function takes an object meeting the requirements of uniform random number generator (26.4.1.2).

25.2.13 Partitions

template<class BidirectionalIterator, class Predicate>
BidirectionalIterator
partition(BidirectionalIterator first,
         BidirectionalIterator last, Predicate pred);

1 Effects: Places all the elements in the range [first,last) that satisfy pred before all the elements that do not satisfy it.

2 Returns: An iterator i such that for any iterator j in the range [first,i) pred(*j) != false, and for any iterator k in the range [i,last), pred(*k) == false.

3 Requires: The type of *first shall satisfy the Swappable requirements (37).

4 Complexity: At most (last - first)/2 swaps. Exactly last - first applications of the predicate are done.

template<class BidirectionalIterator, class Predicate>
BidirectionalIterator
stable_partition(BidirectionalIterator first,
                BidirectionalIterator last, Predicate pred);

5 Effects: Places all the elements in the range [first,last) that satisfy pred before all the elements that do not satisfy it.

6 Returns: An iterator i such that for any iterator j in the range [first,i), pred(*j) != false, and for any iterator k in the range [i,last), pred(*k) == false. The relative order of the elements in both groups is preserved.

7 Requires: The type of *first shall satisfy the Swappable requirements (37), the MoveConstructible requirements (Table 33), and the the MoveAssignable requirements (Table 35).

8 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.

25.3 Sorting and related operations

All the operations in 25.3 have two versions: one that takes a function object of type Compare and one that uses an operator<.
2 Compare is used as a function object which returns true if the first argument is less than the second, and false otherwise. Compare comp is used throughout for algorithms assuming an ordering relation. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.

3 For all algorithms that take Compare, there is a version that uses operator< instead. That is, comp(*i, *j) != false defaults to *i < *j != false. For algorithms other than those described in 25.3.3 to work correctly, comp has to induce a strict weak ordering on the values.

4 The term strict refers to the requirement of an irreflexive relation (!comp(x, x) for all x), and the term weak to requirements that are not as strong as those for a total ordering, but stronger than those for a partial ordering. If we define equiv(a, b) as !comp(a, b) && !comp(b, a), then the requirements are that comp and equiv both be transitive relations:
   — comp(a, b) && comp(b, c) implies comp(a, c)
   — equiv(a, b) && equiv(b, c) implies equiv(a, c) [Note: Under these conditions, it can be shown that
       — equiv is an equivalence relation
       — comp induces a well-defined relation on the equivalence classes determined by equiv
       — The induced relation is a strict total ordering. — end note]

5 A sequence is sorted with respect to a comparator comp if for any iterator i pointing to the sequence and any non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, comp(*(i + n), *i) == false.

6 A sequence [start, finish) is partitioned with respect to an expression f(e) if there exists an integer n such that for all 0 <= i < distance(start, finish), f(*(begin + 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 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).

25.3.1 Sorting

25.3.1.1 sort

```cpp
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
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: The type of *first shall satisfy the Swappable requirements (37), the MoveConstructible requirements (Table 33), and the the MoveAssignable requirements (Table 35).
**Complexity:** Approximately $N \log(N)$ (where $N = \text{last} - \text{first}$) comparisons on the average.\textsuperscript{266}

### 25.3.1.2 stable_sort

```cpp
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:** The type of `*first` shall satisfy the Swappable requirements (37), the `MoveConstructible` requirements (Table 33), and the `MoveAssignable` requirements (Table 35).  
3. **Complexity:** It does at most $N \log^2(N)$ (where $N = \text{last} - \text{first}$) comparisons; if enough extra memory is available, it is $N \log(N)$.  
4. **Remarks:** Stable.

### 25.3.1.3 partial_sort

```cpp
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:** The type of `*first` shall satisfy the Swappable requirements (37), the `MoveConstructible` requirements (Table 33), and the `MoveAssignable` requirements (Table 35).  
3. **Complexity:** It takes approximately $(\text{last} - \text{first}) \times \log(\text{middle} - \text{first})$ comparisons.

### 25.3.1.4 partial_sort_copy

```cpp
template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator
```

\textsuperscript{266} If the worst case behavior is important `stable_sort()` (25.3.1.2) or `partial_sort()` (25.3.1.3) should be used.
25.3 Sorting and related operations

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);

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)).

Returns: The smaller of: result_last or result_first + (last - first).

Requires: The type of *result_first shall satisfy the Swappable requirements (37), the MoveConstructible
requirements (Table 33), and the MoveAssignable requirements (Table 35).

Complexity: Approximately (last - first) * log(min(last - first, result_last - result_first))
comparisons.

25.3.1.5 is_sorted

template<class ForwardIterator>
bool is_sorted(ForwardIterator first, ForwardIterator last);

Returns: is_sorted_until(first, last) == last

template<class ForwardIterator, class Compare>
bool is_sorted(ForwardIterator first, ForwardIterator last,
Compare comp);

Returns: is_sorted_until(first, last, comp) == last

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);

Returns: If distance(first, last) < 2, returns last. Otherwise, returns the last iterator i in [first,last
] for which the range [first,i) is sorted.

Complexity: Linear.

25.3.2 Nth element

template<class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
Draft
RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth, 
RandomAccessIterator last, Compare comp);

After nth_element the element in the position pointed to by nth is the element that would be in that position if
the whole range were sorted. Also for any iterator i in the range [first, nth) and any iterator j in the range
[nth, last) it holds that: !(i > j) or comp (*j, *i) == false.

Requires: The type of *first shall satisfy the Swappable requirements (37), the MoveConstructible require-
ments (Table 33), and the the MoveAssignable requirements (Table 35).

Complexity: Linear on average.

25.3.3 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.

25.3.3.1 lower_bound

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);

Requires: The elements e of [first, last) are partitioned with respect to the expression e < value or comp (e, 
value).

Returns: The furthermost iterator i in the range [first, last] such that for any iterator j in the range [first, 
i) the following corresponding conditions hold: *j < value or comp (*j, value)! = false.

Complexity: At most log₂(last − first) + Θ(1) comparisons.

25.3.3.2 upper_bound

template<class ForwardIterator, class T>
ForwardIterator
upper_bound(ForwardIterator first, ForwardIterator last, 
const T& value);
25.3 Sorting and related operations

```
25.3.3.3 equal_range

template<class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last, 
            const T& value, Compare comp);

1 Requires: The elements e of [first, last) are partitioned with respect to the expression !(value < e) or 
            !comp (value, e).
2 Returns: The furthermost iterator i in the range [first, last) such that for any iterator j in the range [first, 
            i) the following corresponding conditions hold: !(value < *j) or comp (value, *j) == false.
3 Complexity: At most log2(last - first) + O(1) comparisons.
```

25.3.3.4 binary_search

```
template<class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last, 
                   const T& value);
```

Draft
bool binary_search(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);

Requires: The elements e of [first, last) are partitioned with respect to the expressions e < value and !(value < e) or comp(e, value) and !comp(value, e). Also, for all elements e of [first, last), e < value implies !(value < e) or comp(e, value) implies !comp(value, e).

Returns: true if there is an iterator i in the range [first, last) that satisfies the corresponding conditions: !(i < value) && !(value < i) or comp(i, value) == false && comp(value, i) == false.

Complexity: At most log(last - first) + 2 comparisons.

25.3.4 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);

Effects: Merges two sorted ranges [first1, last1) and [first2, last2) into the range [result, result + (last1 - first1) + (last2 - first2)).

The resulting range shall not overlap with either of the original ranges. The list will be sorted in non-decreasing order according to the ordering defined by comp; that is, for every iterator i in [first, last) other than first, the condition *i < *(i - 1) or comp(*i, *(i - 1)) will be false.

Returns: result + (last1 - first1) + (last2 - first2).

Complexity: At most (last1 - first1) + (last2 - first2) - 1 comparisons.

Remarks: Stable.

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);
Effects: Merges two sorted consecutive ranges \([\text{first}, \text{middle})\) and \([\text{middle}, \text{last})\), putting the result of the merge into the range \([\text{first}, \text{last})\). The resulting range will be in non-decreasing order; that is, for every iterator \(i\) in \([\text{first}, \text{last})\) other than \(\text{first}\), the condition \(*i < *(i - 1)\) or, respectively, \(\text{comp}(*i, *(i - 1))\) will be false.

Requires: The type of \(*\text{first}\) shall satisfy the Swappable requirements (37), the MoveConstructible requirements (Table 33), and the the MoveAssignable requirements (Table 35).

Complexity: When enough additional memory is available, \((\text{last} - \text{first}) - 1\) comparisons. If no additional memory is available, an algorithm with complexity \(N \log(N)\) (where \(N\) is equal to \(\text{last} - \text{first}\)) may be used.

Remarks: Stable.

25.3.5 Set operations on sorted structures

This section defines all the basic set operations on sorted structures. They also work with multisets (23.3.4) containing multiple copies of equivalent elements. The semantics of the set operations are generalized to multisets in a standard way by defining \(\text{set_union}()\) to contain the maximum number of occurrences of every element, \(\text{set_intersection}()\) to contain the minimum, and so on.

25.3.5.1 includes

\[
\text{template}<\text{class InputIterator1, class InputIterator2}>
\]

\[
\begin{align*}
\text{bool includes(InputIterator1 } & \text{first1, InputIterator1 last1,} \\
& \text{InputIterator2 first2, InputIterator2 last2);} \\
\text{template}<\text{class InputIterator1, class InputIterator2, class Compare}>
\end{align*}
\]

\[
\begin{align*}
\text{bool includes(InputIterator1 } & \text{first1, InputIterator1 last1,} \\
& \text{InputIterator2 first2, InputIterator2 last2,} \\
& \text{Compare comp);}
\end{align*}
\]

Returns: true if every element in the range \([\text{first2}, \text{last2})\) is contained in the range \([\text{first1}, \text{last1})\). Returns false otherwise.

Complexity: At most \(2 * ((\text{last2} - \text{first2}) + (\text{last1} - \text{first1})) - 1\) comparisons.

25.3.5.2 set_union

\[
\text{template}<\text{class InputIterator1, class InputIterator2,} \\
& \text{class OutputIterator}> \\
\text{OutputIterator} \\
\text{set_union(InputIterator1 first1, InputIterator1 last1,} \\
& \text{InputIterator2 first2, InputIterator2 last2,} \\
& \text{OutputIterator result});}
\]

\[
\text{template}<\text{class InputIterator1, class InputIterator2,} \\
& \text{class OutputIterator, class Compare}> \\
\text{OutputIterator} \\
\text{set_union(InputIterator1 first1, InputIterator1 last1,}
\]

Draft
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator
set_intersection(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2,
                OutputIterator result);

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.
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 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.

25.3.5.3 set_intersection

**set.intersection**

\[
\text{template}\langle\text{class InputIterator1, class InputIterator2, class OutputIterator}\rangle
\text{OutputIterator}
\text{set_intersection(InputIterator1 } \text{first1}, \text{InputIterator1 } \text{last1},
\text{InputIterator2 } \text{first2}, \text{InputIterator2 } \text{last2},
\text{OutputIterator } \text{result});
\]

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 \(2 \times ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1\) 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, the first \(\min(m, n)\) elements shall be copied from the first range to the output range, in order.

25.3.5.4 set_difference

**set.difference**

\[
\text{template}\langle\text{class InputIterator1, class InputIterator2, class OutputIterator}\rangle
\text{OutputIterator}
\text{set_difference(InputIterator1 } \text{first1}, \text{InputIterator1 } \text{last1},
\text{InputIterator2 } \text{first2}, \text{InputIterator2 } \text{last2},
\text{OutputIterator } \text{result});
\]
set_difference(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, InputIterator2 last2,
   OutputIterator result);

template<class InputIterator1, class InputIterator2,
   class OutputIterator, class Compare>
OutputIterator
   set_difference(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, InputIterator2 last2,
   OutputIterator result, Compare comp);

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.

25.3.5.5 set_symmetric_difference

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);

Effects: Copies the elements of the range \([first1, last1)\) which are not present in the range \([first2, last2)\), and the elements of the range \([first2, last2)\) which 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 \(2 \times ((last1 - first1) + (last2 - first2)) - 1\) 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, 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 $m < n$.

25.3.6 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 $\text{pop\_heap}()$, or a new element added by $\text{push\_heap}()$, in $\mathcal{O}(\log(N))$ time.

These properties make heaps useful as priority queues.

$\text{make\_heap}()$ converts a range into a heap and $\text{sort\_heap}()$ turns a heap into a sorted sequence.

25.3.6.1 push_heap

\begin{verbatim}
template<class RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);
\end{verbatim}

**Effects:** Places the value in the location $\text{last} - 1$ 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 33) and the the MoveAssignable requirements (Table 35).

**Complexity:** At most $\log(last - first)$ comparisons.

25.3.6.2 pop_heap

\begin{verbatim}
template<class RandomAccessIterator>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);
\end{verbatim}

**Effects:** Swaps the value in the location $first$ with the value in the location $\text{last} - 1$ and makes $[first, last - 1)$ into a heap.

**Requires:** The range $[first, last)$ shall be a valid heap. The type of $*first$ shall satisfy the Swappable requirements (37), the MoveConstructible requirements (Table 33), and the the MoveAssignable requirements (Table 35).

**Complexity:** At most $2 \times \log(last - first)$ comparisons.
25.3 Sorting and related operations

25.3.6.3 make_heap

```cpp
template<class RandomAccessIterator>
void make_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void make_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

1. **Effects:** Constructs a heap out of the range \([first, last)\).
2. **Requires:** The type of \*first shall satisfy the `MoveConstructible` requirements (Table 33) and the `MoveAssignable` requirements (Table 35).
3. **Complexity:** At most \(3 \times (last - first)\) comparisons.

25.3.6.4 sort_heap

```cpp
template<class RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

1. **Effects:** Sorts elements in the heap \([first, last)\).
2. **Requires:** The range \([first, last)\) shall be a valid heap. The type of \*first shall satisfy the `Swappable` requirements (37), the `MoveConstructible` requirements (Table 33), and the `MoveAssignable` requirements (Table 35).
3. **Complexity:** At most \(N \log(N)\) comparisons (where \(N == last - first\)).

25.3.6.5 is_heap

```cpp
template<class RandomAccessIterator>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

1. **Returns:** `is_heap_until(first, last) == last`
2. **Returns:** `is_heap_until(first, last, comp) == last`

```cpp
RandomAccessIterator is_heap_until(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
RandomAccessIterator is_heap_until(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Draft
Returns: If distance(first, last) < 2, returns last. Otherwise, returns the last iterator i in [first, last) for which the range [first, i) is a heap.

Complexity: Linear.

25.3.7 Minimum and maximum

template<class T> const T& min(const T& a, const T& b);
template<class T, class Compare>
const T& min(const T& a, const T& b, Compare comp);

Requires: Type T is LessThanComparable (32).

Returns: The smaller value.

Remarks: Returns the first argument when the arguments are equivalent.

template<class T> const T& max(const T& a, const T& b);
template<class T, class Compare>
const T& max(const T& a, const T& b, Compare comp);

Requires: Type T is LessThanComparable (32).

Returns: The larger value.

Remarks: Returns the first argument when the arguments are equivalent.

template<class T> pair<const T&, const T&> minmax(const T& a, const T& b);
template<class T, class Compare>
pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);

Requires: Type T shall be LessThanComparable (32).

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 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 any iterator j in the range [first, last) the following corresponding conditions hold: !(j < i) or comp(*j, *i) == 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);

13 Returns: The first iterator \(i\) in the range \([first, last)\) such that for any iterator \(j\) in the range \([first, last)\) the following corresponding conditions hold: \(!(*i < *j)\) or \(comp(*i, *j) == false\). Returns \(last\) if \(first == last\).

14 Complexity: Exactly \(\text{max}((last - first) - 1, 0)\) 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);

15 Returns: \(\text{make_pair}(m, M)\), where \(m\) is \(\text{min_element}(first, last)\) or \(\text{min_element}(first, last, comp)\) and \(M\) is \(\text{max_element}(first, last)\) or \(\text{max_element}(first, last, comp)\).

16 Complexity: At most \(\text{max}(2 \cdot (last - first) - 2, 0)\) applications of the corresponding comparisons.

### 25.3.8 Lexicographical comparison

[alg.lex.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);

1 Returns: true if the sequence of elements defined by the range \([first1, last1)\) is lexicographically less than the sequence of elements defined by the range \([first2, last2)\).

Returns false otherwise.

2 Complexity: At most \(2 \cdot \text{min}((last1 - first1), (last2 - first2))\) applications of the corresponding comparison.

3 Remarks: If two sequences have the same number of elements and their corresponding elements 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.
for (; first1 != last1 && first2 != last2 ; ++first1, ++first2) {
    if (*first1 < *first2) return true;
    if (*first2 < *first1) return false;
} return first1 == last1 && first2 != last2;

25.3.9 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: The type of *first shall satisfy the Swappable requirements (37).

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: The type of *first shall satisfy the Swappable requirements (37).

Complexity: At most \((last - first)/2\) swaps.

25.4 C library algorithms

Header <cstdlib> (partial, Table 102).

The contents are the same as the Standard C library header <stdlib.h> with the following exceptions:

The function signature:
Table 102: Header `<cstdlib>` synopsis

<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>

bsearch(const void *, const void *, size_t, size_t,
    int (*)(const void *, const void *));

is replaced by the two declarations:

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.

4 The function signature:

qsort(void *, size_t, size_t,
    int (*)(const void *, const void *));

is replaced by the two declarations:

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 **POD trivial** type.

[Note: Because the function argument compar() may throw an exception, bsearch() and qsort() are allowed to propagate the exception (17.4.4.8). — end note]

See also: ISO C subclause 7.10.5.
Chapter 26  Numerics library

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 \((n\text{-}at-a\text{-}time)\) arrays, generalized numeric algorithms, and facilities included from the ISO C library, as summarized in Table 103.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>26.1 Requirements</td>
<td></td>
</tr>
<tr>
<td>26.3 Complex Numbers</td>
<td><code>&lt;complex&gt;</code></td>
</tr>
<tr>
<td>26.4 Random number generation</td>
<td><code>&lt;random&gt;</code></td>
</tr>
<tr>
<td>26.5 Numeric arrays</td>
<td><code>&lt;valarray&gt;</code></td>
</tr>
<tr>
<td>26.6 Generalized numeric operations</td>
<td><code>&lt;numeric&gt;</code></td>
</tr>
<tr>
<td>26.7 C library</td>
<td><code>&lt;cmath&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;ctgmath&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;tgmath.h&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
</tbody>
</table>

26.1 Numeric type requirements

The `complex` and `valarray` components are parameterized by the type of information they contain and manipulate. A C++ program shall instantiate these components only with a type `T` that satisfies the following requirements:

- `T` is not an abstract class (it has no pure virtual member functions);
- `T` is not a reference type;
- `T` is not cv-qualified;
- If `T` is a class, it has a public default constructor;
- If `T` is a class, it has a public copy constructor with the signature `T::T(const T&)`;
- If `T` is a class, it has a public destructor;
- If `T` is a class, it has a public assignment operator whose signature is either `T& T::operator=(const T&)` or `T& T::operator=(T)`.

\(^{267}\) In other words, value types. These include arithmetic types, pointers, the library class `complex`, and instantiations of `valarray` for value types.
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. 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 (23.2.5) instead of valarray for that class; — end note]

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$<\text{complex}>$, but operator() will not be successfully instantiated for valarray$<\text{complex}>$ operands, since complex does not have any ordering operators. — end example ]

### 26.2 The floating-point environment

#### 26.2.1 Header `<cfenv>` synopsis

```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);
} // namespace std
```

Draft
The header also defines the macros:

- FE_ALL_EXCEPT
- FE_DIVBYZERO
- FE_INEXACT
- FE_INVALID
- FE_OVERFLOW
- FE_UNDERFLOW
- FE_DOWNWARD
- FE_TONEAREST
- FE_TOWARDZERO
- FE_UPWARD
- FE_DFL_ENV

The header defines all functions, types, and macros the same as C99 subclause 7.6.

### 26.2.2 Header <fenv.h>

The header behaves as if it includes the header <cfenv>, and provides sufficient using declarations to declare in the global namespace all function and type names declared or defined in the header <cfenv>.

### 26.3 Complex numbers

The header <complex> defines a class template, and numerous functions for representing and manipulating complex numbers.

The effect of instantiating the template complex for any type other than float, double, or long double is unspecified. The specializations complex<float>, complex<double>, and complex<long double> are literal types (3.9).

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.

### 26.3.1 Header <complex> synopsis

```cpp
namespace std {
    template<class T> class complex;
    template<> class complex<float>;
    template<> class complex<double>;
    template<> class complex<long double>;

    // 26.3.6 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>&);

    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>&);
```

Draft
template<class T> complex<T> operator*
  (const complex<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 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> bool operator==
  (const complex<T>&, const complex<T>&);

template<class T> bool operator==(const complex<T>&, const T&);

template<class T> bool operator==(const T&, const complex<T>&);

template<class T> bool operator!=(const complex<T>&, const complex<T>&);

template<class T> bool operator!=(const complex<T>&, const T&);

template<class T> bool operator!=(const T&, const complex<T>&);

// 26.3.7 values:

template<class T> T real(const complex<T>&);

template<class T> 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> fabs(const complex<T>&);

template<class T> complex<T> polar(const T&, const T& = 0);

// 26.3.8 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>&);
26.3 Complex numbers

```cpp
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>&, int);
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>&);
}

26.3.2 Class template complex

namespace std {
    template<class T>
    class complex {
    public:
        typedef T value_type;

        complex(const T& re = T(), const T& im = T());
        complex(const complex&);
        template<class X> complex(const complex<X>&);

        T real() const;
        T imag() const;

        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>&);
        template<class X> complex<T>& operator-=(const complex<X>&);
        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.

### 26.3.3 complex specializations

```cpp
template<> class complex<float> {  
public:
    typedef float value_type;
    
    constexpr complex(float re = 0.0f, float im = 0.0f);
    explicit constexpr complex(const complex<double>&);
    explicit constexpr complex(const complex<long double>&);

    constexpr float real() const;
    constexpr float imag() const;

    complex<float>& operator= (float);
    complex<float>& operator+=(float);
    complex<float>& operator-=(float);
    complex<float>& operator*=(float);
    complex<float>& operator/=(float);
    complex<float>& operator=(const complex<float>&);
    template<class X> complex<float>& operator= (const complex<X>&);
    template<class X> complex<float>& operator+=(const complex<X>&);
    template<class X> complex<float>& operator-=(const complex<X>&);
    template<class X> complex<float>& operator*=(const complex<X>&);
    template<class X> complex<float>& operator/=(const complex<X>&);
};
```

```cpp
template<> class complex<double> {  
public:
    typedef double value_type;
    
    constexpr complex(double re = 0.0, double im = 0.0);
    constexpr complex(const complex<float>&);
    explicit constexpr complex(const complex<long double>&);

    constexpr double real() const;
    constexpr double imag() const;

    complex<double>& operator= (double);
    complex<double>& operator+=(double);
    complex<double>& operator-=(double);
    complex<double>& operator*=(double);
    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 X> complex<double>& operator/=(const complex<X>&);

};

template<> class complex<long double> {
public:
    typedef long double value_type;

    complex<long double> real() const;
    complex<long double> imag() const;

    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);

    template<class X> complex<long double>& operator=(const complex<X>&);
    template<class X> complex<long double>& operator+=(const complex<X>&);
    template<class X> complex<long double>& operator-=(const complex<X>&);
    template<class X> complex<long double>& operator*=(const complex<X>&);
    template<class X> complex<long double>& operator/=(const complex<X>&);
};

26.3.4 complex member functions

    template<class T> complex(const T& re = T(), const T& im = T());

1   Effects: Constructs an object of class complex.
2   Postcondition: real() == re && imag() == im.

26.3.5 complex member operators

    template <class T> complex<T>& operator+=(const T& rhs);

1   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.
2   Returns: *this.
26.3 Complex numbers

Effects: Subtracts the scalar value \( \text{rhs} \) from the real part of the complex value \(*\text{this}\) and stores the result in the real part of \(*\text{this}\), leaving the imaginary part unchanged.

Returns: \(*\text{this}\).

template <class T> complex<T>& operator-= (const T& rhs);

Effects: Multiplies the scalar value \( \text{rhs} \) by the complex value \(*\text{this}\) and stores the result in \(*\text{this}\).

Returns: \(*\text{this}\).

template <class T> complex<T>& operator/= (const T& rhs);

Effects: Divides the scalar value \( \text{rhs} \) into the complex value \(*\text{this}\) and stores the result in \(*\text{this}\).

Returns: \(*\text{this}\).

template <class T> complex<T>& operator+= (const complex<T>& rhs);

Effects: Adds the complex value \( \text{rhs} \) to the complex value \(*\text{this}\) and stores the sum in \(*\text{this}\).

Returns: \(*\text{this}\).

template <class T> complex<T>& operator-= (const complex<T>& rhs);

Effects: Subtracts the complex value \( \text{rhs} \) from the complex value \(*\text{this}\) and stores the difference in \(*\text{this}\).

Returns: \(*\text{this}\).

template <class T> complex<T>& operator*= (const complex<T>& rhs);

Effects: Multiplies the complex value \( \text{rhs} \) by the complex value \(*\text{this}\) and stores the product in \(*\text{this}\).

Returns: \(*\text{this}\).

template <class T> complex<T>& operator/= (const complex<T>& rhs);

Effects: Divides the complex value \( \text{rhs} \) into the complex value \(*\text{this}\) and stores the quotient in \(*\text{this}\).

Returns: \(*\text{this}\).

26.3.6 complex non-member operations

[complex.ops]

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);

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>
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>
bool operator==(const complex<T>& lhs, const complex<T>& rhs);
template<class T> bool operator==(const complex<T>& lhs, const T& rhs);
template<class T> bool operator==(const T& lhs, const complex<T>& rhs);

Returns: lhs.real() == rhs.real() && lhs.imag() == rhs.imag().

Remarks: The imaginary part is assumed to be T(), or 0.0, for the T arguments.

template<class T>
bool operator!=(const complex<T>& lhs, const complex<T>& rhs);
template<class T> bool operator!=(const complex<T>& lhs, const T& rhs);
template<class T> bool operator!=(const T& lhs, const complex<T>& rhs);

Returns: rhs.real() != lhs.real() || rhs.imag() != lhs.imag().

template<class T, class charT, class traits>
basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>& is, complex<T>& z);

Effects: Extracts a complex number \( z \) of the form: \( u, (u), (u,v) \), where \( u \) is the real part and \( v \) is the imaginary part (27.6.1.2).

Requires: The input values be convertible to T.

If bad input is encountered, calls is.setstate(ios::failbit) (which may throw ios::failure (27.4.4.3).

Returns: is.
26.3 Complex numbers

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();
  }

26.3.7 complex value operations

template<class T> T real(const complex<T>& x);

Returns: \( x . real() \).

template<class T> T imag(const complex<T>& x);

Returns: \( x . imag() \).

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> fabs(const complex<T>& x);

Effects: Behaves the same as C99 function \( \text{cabs} \), defined in subclause 7.3.8.1.

template<class T> complex<T> polar(const T& rho, const T& theta = 0);
8 Returns: the complex value corresponding to a complex number whose magnitude is \( \rho \) and whose phase angle is \( \theta \).

### 26.3.8 complex transcendental

[complex.transcendentals]

```cpp
template<class T> complex<T> acos(const complex<T>& x);
Effects: Behaves the same as C99 function \( \text{cacos} \), defined in subclause 7.3.5.1.

template<class T> complex<T> asin(const complex<T>& x);
Effects: Behaves the same as C99 function \( \text{casin} \), defined in subclause 7.3.5.2.

template<class T> complex<T> atan(const complex<T>& x);
Effects: Behaves the same as C99 function \( \text{catan} \), defined in subclause 7.3.5.3.

template<class T> complex<T> acosh(const complex<T>& x);
Effects: Behaves the same as C99 function \( \text{cacosh} \), defined in subclause 7.3.6.1.

template<class T> complex<T> asinh(const complex<T>& x);
Effects: Behaves the same as C99 function \( \text{casinh} \), defined in subclause 7.3.6.2.

template<class T> complex<T> atanh(const complex<T>& x);
Effects: Behaves the same as C99 function \( \text{catanh} \), defined in subclause 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);
Remarks: the branch cuts are along the negative real axis.

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);
Remarks: the branch cuts are along the negative real axis.

Returns: the complex common (base 10) logarithm of \( x \), defined as \( \log(x)/\log(10) \).
```
26.3 Complex numbers

template<class T> complex<T> pow(const complex<T>& x, int y);
template<class T>
    complex<T> pow(const complex<T>& x, const complex<T>& y);
template<class T> complex<T> pow (const complex<T>& x, const T& y);
template<class T> complex<T> pow (const T& x, const complex<T>& y);

14 Remarks: the branch cuts are along the negative real axis.

15 Returns: the complex power of base \(x\) raised to the \(y\)-th power, defined as \(\exp(y \times \log(x))\). The value returned for \(\text{pow}(0,0)\) is implementation-defined.

template<class T> complex<T> sin (const complex<T>& x);

16 Returns: the complex sine of \(x\).

template<class T> complex<T> sinh (const complex<T>& x);

17 Returns: the complex hyperbolic sine of \(x\).

template<class T> complex<T> sqrt (const complex<T>& x);

18 Remarks: the branch cuts are along the negative real axis.

19 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.

template<class T> complex<T> tan (const complex<T>& x);

20 Returns: the complex tangent of \(x\).

template<class T> complex<T> tanh (const complex<T>& x);

21 Returns: the complex hyperbolic tangent of \(x\).

26.3.9 Additional Overloads

The following function templates shall have additional overloads:

\[
\begin{align*}
\text{arg} & \quad \text{norm} \\
\text{conj} & \quad \text{polar} \\
\text{imag} & \quad \text{real}
\end{align*}
\]

The additional overloads shall be sufficient to ensure:

1. If the argument has type `long double`, then it is effectively cast to `complex<long double>`.

2. Otherwise, if the argument has type `double` or an integer type, then it is effectively cast to `complex<double>`.

3. Otherwise, if the argument has type `float`, then it is effectively cast to `complex<float>`.

Function template `pow` shall have additional overloads sufficient to ensure, for a call with at least one argument of type `complex<T>`:

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>`.

26.3.10 **Header `<ccomplex>`**

The header behaves as if it simply includes the header `<complex>`.

26.3.11 **Header `<complex.h>`**

The header behaves as if it includes the header `<ccomplex>`, and provides sufficient using declarations to declare in the global namespace all function and type names declared or defined in the header `<complex>`.

26.4 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`.

If integer-valued, an entity may optionally be further characterized as `signed` or `unsigned`, according to `T`.

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 $2^w$.

26.4.1 Requirements

26.4.1.1 General requirements

Throughout this subclause 26.4, the effect of instantiating a template
26.4 Random number generation

Numerics library

26.4 Random number generation

- a) that has a template type parameter named UniformRandomNumberGenerator is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of uniform random number generator (26.4.1.2).

- b) 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 (26.4.1.3).

- c) 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.

- d) 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.

- e) that has a template type parameter named UIntType is undefined unless the corresponding template argument is cv-unqualified and is one of unsigned short, unsigned int, unsigned long, or unsigned long long.

2 All members declared static const in any of the following classes or class templates shall be defined in such a way that they are usable as integral constant expressions.

26.4.1.2 Uniform random number generator requirements

A class X satisfies the requirements of a uniform random number generator if the expressions shown in table 104 are valid and have the indicated semantics. In that table,

- a) T is the type named by X’s associated result_type, and

- b) u is a value of X.

Table 104: 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>X::result_type</td>
<td>T</td>
<td>T is an unsigned integer type (3.9.1).</td>
<td>compile-time</td>
</tr>
<tr>
<td>u()</td>
<td>T</td>
<td>Returns a value in the closed interval [X::min, X::max].</td>
<td>amortized constant</td>
</tr>
<tr>
<td>X::min</td>
<td>T</td>
<td>Denotes the least value potentially returned by operator().</td>
<td>compile-time</td>
</tr>
<tr>
<td>X::max</td>
<td>T</td>
<td>Denotes the greatest value potentially returned by operator().</td>
<td>compile-time</td>
</tr>
</tbody>
</table>

26.4.1.3 Random number engine requirements

A class X that satisfies the requirements of a uniform random number generator (26.4.1.2) also satisfies the requirements of a random number engine if the expressions shown in table 105 are valid and have the indicated semantics, and if X also satisfies all other requirements of this section 26.4.1.3. In that table and throughout this section 26.4.1.3,
833  Numerics library  26.4  Random number generation

(a) \( T \) is the type named by \( X \)'s associated \( \text{result\_type} \);
(b) \( u \) is a value of \( X \), \( v \) is an lvalue of \( X \), \( x \) and \( y \) are (possibly \( \text{const} \)) values of \( X \);
(c) \( a \) is a value of arithmetic type (3.9.1);
(d) \( q \) is an lvalue of type \( \text{seed\_seq} \) (26.4.7.1);
(e) \( z \) is a value of type \( \text{unsigned long long} \);
(f) \( os \) is an lvalue of the type of some class template specialization \( \text{basic\_ostream}\langle \text{charT, traits}\rangle \); and
(g) \( is \) is an lvalue of the type of some class template specialization \( \text{basic\_istream}\langle \text{charT, traits}\rangle \);

where \( \text{charT} \) and \( \text{traits} \) are constrained according to 21 and 27.

A random number engine object \( x \) has at any given time a state \( x_i \) for some integer \( i \geq 0 \). Upon construction, a random number engine \( x \) has an initial state \( x_0 \). An engine’s state may be established by invoking a constructor, \( \text{seed} \) member function, \( \text{operator=} \), or a suitable \( \text{operator} \gg \).

The specification of each random number engine defines the size of its state in multiples of the size of its \( \text{result\_type} \), given as an integral constant expression. The specification of each random number engine also defines

(a) the \( \text{transition algorithm} \) \( TA \) by which the engine’s state \( x_i \) is advanced to its \( \text{successor state} \) \( x_i+1 \), and
(b) the \( \text{generation algorithm} \) \( GA \) by which an engine’s state is mapped to a value of type \( \text{result\_type} \).

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X() )</td>
<td>—</td>
<td>Creates an engine with the same initial state as all other default-constructed engines of type ( X ).</td>
<td>( \mathcal{O}(\text{sizeof state}) )</td>
</tr>
<tr>
<td>( X(x) )</td>
<td>—</td>
<td>Creates an engine that compares equal to ( x ).</td>
<td>( \mathcal{O}(\text{sizeof state}) )</td>
</tr>
<tr>
<td>( X(s) )</td>
<td>—</td>
<td>Creates an engine with initial state determined by ( \text{static_cast}&lt;X::\text{result_type}&gt;(s) ).</td>
<td>( \mathcal{O}(\text{sizeof state}) )</td>
</tr>
<tr>
<td>( X(q) )</td>
<td>—</td>
<td>With ( n = q.\text{size}() ), creates an engine ( u ) with initial state determined as follows: If ( n ) is 0, ( u == X() ); otherwise, the initial state depends on a sequence produced by one call to ( q.\text{randomize} ).</td>
<td>( \mathcal{O}(\max(n, \text{size of state})) )</td>
</tr>
<tr>
<td>( u.\text{seed}() )</td>
<td>void</td>
<td>post: ( u == X() )</td>
<td>same as ( X() )</td>
</tr>
<tr>
<td>( u.\text{seed}(s) )</td>
<td>void</td>
<td>post: ( u == X(s) )</td>
<td>same as ( X(s) )</td>
</tr>
<tr>
<td>( u.\text{seed}(q) )</td>
<td>void</td>
<td>post: ( u == X(q) ).</td>
<td>same as ( X(q) )</td>
</tr>
</tbody>
</table>

\( ^{268} \)This constructor (as well as the corresponding \( \text{seed()} \) function below) may be particularly useful to applications requiring a large number of independent random sequences.

Draft
<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>u()</td>
<td>T</td>
<td>Sets the state to $u_{i+1} = TA(u_i)$ and returns $GA(u_i)$.</td>
<td>amortized constant</td>
</tr>
<tr>
<td>u.discard(z)</td>
<td>void</td>
<td>post: The state of $u$ is identical to that produced by $z$ consecutive calls to $u()$.</td>
<td>no worse than the complexity of $z$ consecutive calls to $u()$</td>
</tr>
<tr>
<td>$x == y$</td>
<td>bool</td>
<td>With $S_x$ and $S_y$ as the infinite sequences of values that would be generated by repeated future calls to $x()$ and $y()$, respectively, returns true if $S_x = S_y$; returns false otherwise.</td>
<td>$O(\text{sizeof state})$</td>
</tr>
<tr>
<td>$x != y$</td>
<td>bool</td>
<td>$!(x == y)$</td>
<td>$O(\text{sizeof state})$</td>
</tr>
<tr>
<td>os &lt;&lt; x</td>
<td>reference to the type of os</td>
<td>With os_fmtflags set to ios_base::dec</td>
<td>ios_base::fixed</td>
</tr>
<tr>
<td>is &gt;&gt; v</td>
<td>reference to the type of is</td>
<td>Sets v’s state as determined by reading its textual representation from is. If bad input is encountered, ensures that v’s state is unchanged by the operation and calls is_setstate(ios::failbit) (which may throw ios::failure (27.4.4.3)). pre: The textual representation 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>$O(\text{sizeof state})$</td>
</tr>
</tbody>
</table>

---

269 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 to $u()$. 

Draft
4 X shall satisfy the requirements of uniform random number generator (26.4.1.2) as well as of CopyConstructible (34) and of Assignable (23.1). Copy construction and assignment shall each be of complexity $O(\text{size_of\_state})$.

5 If a textual representation written via `os << x` was subsequently read via `is >> v`, then $x == v$ provided that there have been no intervening invocations of $x$ or of $v$.

26.4.1.4 Random number engine adaptor requirements

A random number engine adaptor is a random number engine that takes values produced by some other random number engine or engines, and applies an algorithm to those values in order to deliver a sequence of values with different randomness properties. Engines adapted in this way are termed base engines in this context. The terms unary, binary, and so on, may be used to characterize an adaptor depending on the number $n$ of base engines that adaptor utilizes.

A class $X$ satisfies the requirements of a random number engine adaptor if the expressions shown in table 106 are valid and have the indicated semantics, and if $X$ and its associated types also satisfies all other requirements of this section 26.4.1.4. In that table and throughout this section,

1. $B_i$ is the type of the $i\text{th}$ of $X$’s base engines, $1 \leq i \leq n$; and
2. $b_i$ is a value of $B_i$.

If $X$ is unary, $i$ is omitted and understood to be 1.

Table 106: Random number engine adaptor requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X::base_i_type$</td>
<td>$B_i$</td>
<td>—</td>
<td>compile time</td>
</tr>
<tr>
<td>$X::base_i()$</td>
<td>$\text{const} B_i$&amp;</td>
<td>Returns a reference to $b_i$.</td>
<td>constant</td>
</tr>
</tbody>
</table>

3 $X$ shall satisfy the requirements of random number engine (26.4.1.3), subject to the following:

1. The base engines of $X$ are arranged in an arbitrary but fixed order, and that order is consistently used whenever functions are applied to those base engines in turn.
2. The complexity of each function is at most the sum of the complexities of the corresponding functions applied to each base engine.
3. The state of $X$ includes the state of each of its base engines. The size of $X$’s state is no less than the sum of the base engine sizes. Copying $X$’s state (e.g., during copy construction or copy assignment), includes copying, in turn, each base engine of $X$.
4. The textual representation of $X$ includes, in turn, the textual representation of each of its base engines.
5. When $X::X$ is invoked with no arguments, each of $X$’s base engines is constructed, in turn, as if by its respective default constructor. When $X::X$ is invoked with an $X::\text{result\_type}$ value $s$, each of $X$’s base engines is constructed, in turn, with the next available value from the list $s+0, s+1, \ldots$. When $X::X$ is invoked with an argument of type $\text{seed\_seq}$, each of $X$’s base engines is constructed, in turn, with that object as argument.

4 $X$ shall have one additional constructor with $n$ or more parameters such that the type of parameter $i$, $1 \leq i \leq n$, is $\text{const } B_i$& and such that all remaining parameters, if any, have default values. The constructor shall construct $X$, initializing
each of its base engines, in turn, with a copy of the value of the corresponding argument.

### 26.4.1.5 Random number distribution requirements

A class \( X \) satisfies the requirements of a random number distribution if the expressions shown in table 107 are valid and have the indicated semantics, and if \( X \) and its associated types also satisfies all other requirements of this section 26.4.1.5. In that table and throughout this section,

- a) \( T \) is the type named by \( X \)'s associated \texttt{result\_type};
- b) \( P \) is the type named by \( X \)'s associated \texttt{param\_type};
- c) \( u \) is a value of \( X \) and \( x \) is a (possibly const) value of \( X \);
- d) \( \texttt{glb} \) and \( \texttt{lub} \) are values of \( T \) respectively corresponding to the greatest lower bound and the least upper bound on the values potentially returned by \( u \)'s \texttt{operator()} , as determined by the current values of \( u \)'s parameters;
- e) \( p \) is a value of \( P \);
- f) \( e \) is an lvalue of an arbitrary type that satisfies the requirements of a uniform random number generator (26.4.1.2);
- g) \( \texttt{os} \) is an lvalue of the type of some class template specialization \texttt{basic\_ostream<\texttt{charT}, \texttt{traits}>}; and
- h) \( \texttt{is} \) is an lvalue of the type of some class template specialization \texttt{basic\_istream<\texttt{charT}, \texttt{traits}>};

where \texttt{charT} and \texttt{traits} are constrained according to 21 and 27.

The specification of each random number distribution identifies an associated mathematical probability density function \( p(z) \) or an associated discrete probability function \( P(z_i) \). Such functions are 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.

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X::\texttt{result_type} )</td>
<td>( T )</td>
<td>( T ) is an arithmetic type.</td>
<td>compile-time</td>
</tr>
<tr>
<td>( X::\texttt{param_type} )</td>
<td>( P )</td>
<td>—</td>
<td>compile-time</td>
</tr>
<tr>
<td>( X(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>( u.\texttt{reset()} )</td>
<td>\texttt{void}</td>
<td>Subsequent uses of ( u ) do not depend on values produced by ( e ) prior to invoking \texttt{reset}.</td>
<td>constant</td>
</tr>
<tr>
<td>( x.\texttt{param()} )</td>
<td>( P )</td>
<td>Returns a value ( p ) such that ( X(p).\texttt{param()} == p ).</td>
<td>no worse than the complexity of ( X(p) )</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>u.param(p)</td>
<td>void</td>
<td>post: u.param() == p.</td>
<td>no worse than the complexity of X(p)</td>
</tr>
<tr>
<td>u(e)</td>
<td>T</td>
<td>With p = u.param(), the sequence of numbers returned by successive invocations with the same object e is randomly distributed according to the associated p(z</td>
<td>{p}) or P(z</td>
</tr>
<tr>
<td>u(e,p)</td>
<td>T</td>
<td>The sequence of numbers returned by successive invocations with the same objects e and p is randomly distributed according to the associated p(z</td>
<td>{p}) or P(z</td>
</tr>
<tr>
<td>x.min()</td>
<td>T</td>
<td>Returns glb.</td>
<td>constant</td>
</tr>
<tr>
<td>x.max()</td>
<td>T</td>
<td>Returns lub.</td>
<td>constant</td>
</tr>
<tr>
<td>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; u</td>
<td>reference to the type of is</td>
<td>Restores from is the parameters and additional internal data of u. If bad input is encountered, ensures that u is unchanged by the operation and calls is.setstate(ios::failbit) (which may throw ios::failure (27.4.4.3)). 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>

3 X shall satisfy the requirements of CopyConstructible (34) and Assignable (23.1).

4 The sequence of numbers produced by repeated invocations of x(e) shall be independent of any invocation of os << x.
26.4 Random number generation

or of any `const` member function of `X` between any of the invocations `x(e)`.

5 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(e)` shall produce the same sequence of numbers as would repeated invocations of `x(e)`.

6 It is unspecified whether `X::param_type` is declared as a (nested) class or via a typedef. In this subclause 26.4, declarations of `X::param_type` are in the form of typedefs only for convenience of exposition.

7 `P` shall satisfy the requirements of CopyConstructible, Assignable, and EqualityComparable (31).

8 For each of the constructors of `X` 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 `X` that return values corresponding to parameters of the distribution, `P` shall have a corresponding member function with the identical name, type, and semantics.

9 `P` shall have a declaration of the form

```cpp
typedef X distribution_type;
```

26.4.2 Header `<random>` synopsis

```cpp
namespace std {
    // 26.4.3.1 Class template linear_congruential_engine
    template <class UIntType, UIntType a, UIntType c, UIntType m>
    class linear_congruential_engine;

    // 26.4.3.2 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, size_t s,
             UIntType b, size_t t,
             UIntType c, size_t l>
    class mersenne_twister_engine;

    // 26.4.3.3 Class template subtract_with_carry_engine
    template <class UIntType, size_t w, size_t s, size_t r>
    class subtract_with_carry_engine;

    // 26.4.4.1 Class template discard_block_engine
    template <class Engine, size_t p, size_t r>
    class discard_block_engine;

    // 26.4.4.2 Class template independent_bits_engine
    template <class Engine, size_t w, class UIntType>
    class independent_bits_engine;

    // 26.4.4.3 Class template shuffle_order_engine
    template <class Engine, size_t k>
    class shuffle_order_engine;
}
```

Draft
// 26.4.4.4 Class template xor_combine_engine
template <class Engine1, size_t s1, class Engine2, size_t s2=0u>
   class xor_combine_engine;

// 26.4.5 Engines and engine adaptors with predefined parameters
typedef see below minstd_rand0;
typedef see below minstd_rand;
typedef see below mt19937;
typedef see below ranlux24_base;
typedef see below ranlux48_base;
typedef see below ranlux24;
typedef see below ranlux48;
typedef see below knuth_b;

// 26.4.6 Class random_device
class random_device;

// 26.4.7.1 Class seed_seq
class seed_seq;

// 26.4.7.2 Function template generate_canonical
template<class RealType, class UniformRandomNumberGenerator, size_t bits>
   RealType generate_canonical(UniformRandomNumberGenerator& g);

// 26.4.8.1.1 Class template uniform_int_distribution
template <class IntType = int>
   class uniform_int_distribution;

// 26.4.8.1.2 Class template uniform_real_distribution
template <class RealType = double>
   class uniform_real_distribution;

// 26.4.8.2.1 Class bernoulli_distribution
class bernoulli_distribution;

// 26.4.8.2.2 Class template binomial_distribution
template <class IntType = int>
   class binomial_distribution;

// 26.4.8.2.3 Class template geometric_distribution
template <class IntType = int>
   class geometric_distribution;

// 26.4.8.2.4 Class template negative_binomial_distribution
template <class IntType = int>
   class negative_binomial_distribution;

// 26.4.8.3.1 Class template poisson_distribution
template <class IntType = int>
   class poisson_distribution;
26.4 Random number generation

// 26.4.8.3.2 Class template exponential_distribution
template <class RealType = double>
    class exponential_distribution;

// 26.4.8.3.3 Class template gamma_distribution
template <class RealType = double>
    class gamma_distribution;

// 26.4.8.3.4 Class template weibull_distribution
template <class RealType = double>
    class weibull_distribution;

// 26.4.8.3.5 Class template extreme_value_distribution
template <class RealType = double>
    class extreme_value_distribution;

// 26.4.8.4.1 Class template normal_distribution
template <class RealType = double>
    class normal_distribution;

// 26.4.8.4.2 Class template lognormal_distribution
template <class RealType = double>
    class lognormal_distribution;

// 26.4.8.4.3 Class template chi_squared_distribution
template <class RealType = double>
    class chi_squared_distribution;

// 26.4.8.4.4 Class template cauchy_distribution
template <class RealType = double>
    class cauchy_distribution;

// 26.4.8.4.5 Class template fisher_f_distribution
template <class RealType = double>
    class fisher_f_distribution;

// 26.4.8.4.6 Class template student_t_distribution
template <class RealType = double>
    class student_t_distribution;

// 26.4.8.5.1 Class template discrete_distribution
template <class IntType = int>
    class discrete_distribution;

// 26.4.8.5.2 Class template piecewise_constant_distribution
template <class RealType = double>
    class piecewise_constant_distribution;

// 26.4.8.5.3 Class template general_pdf_distribution

Draft
template <class RealType = double>
    class general_pdf_distribution;
} // namespace std

26.4.3 Random number engine class templates

Except where specified otherwise, the complexity of all functions specified in the following sections is constant.

Except as required by table 105, no function described in this section 26.4.3 throws an exception.

The class templates specified in this section 26.4.3 satisfy the requirements of random number engine (26.4.1.3). Descriptions are provided here only for operations on the engines that are not described in those requirements or for operations where there is additional semantic information. Declarations for copy constructors, for copy assignment operators, and for equality and inequality operators are not shown in the synopses.

26.4.3.1 Class template linear_congruential_engine

A linear_congruential_engine random number engine produces unsigned integer random numbers. The state $x_i$ of a linear_congruential_engine object $x$ is of size 1 and consists of a single integer. The transition algorithm is a modular linear function of the form $T_A(x_i) = (a \cdot x_i + c) \mod m$; the generation algorithm is $G_A(x_i) = x_i + 1$.

```
#template <class UIntType, UIntType a, UIntType c, UIntType m>
class linear_congruential_engine
{
public:
    // types
    typedef UIntType result_type;

    // engine characteristics
    static const result_type multiplier = a;
    static const result_type increment = c;
    static const result_type modulus = m;
    static const result_type min = c == 0u ? 1u : 0u;
    static const result_type max = m - 1u;
    static const result_type default_seed = 1u;

    // constructors and seeding functions
    explicit linear_congruential_engine(result_type s = default_seed);
    explicit linear_congruential_engine(seed_seq& q);
    void seed(result_type s = default_seed);
    void seed(seed_seq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);
};
```

The template parameter UIntType shall denote an unsigned integral type large enough to store values as large as $m - 1$. If the template parameter is 0, the modulus $m$ used throughout this section 26.4.3.1 is $\text{numeric\_limits<result\_type>::max()}$ plus 1. [Note: The result need not be representable as a value of type result_type. —end note]

Otherwise, the following relations shall hold: $a < m$ and $c < m$. 

Draft
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
explicit linear_congruential_engine(seed_seq& q);
```

**Effects:** Constructs a `linear_congruential_engine` object. With $k = \lceil \log_2 m \rceil$ and $a$ an array (or equivalent) of length $k+3$, invokes `q.randomize(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 $S$.

### 26.4.3.2 Class template `mersenne_twister_engine`

A `mersenne_twister_engine` random number engine\(^{270}\) 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 $u, s, b, t, c$, 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 $X_{i+m-n} \text{xor} (Y \text{rshift} 1) \text{xor} \alpha$.

c) Let $z_4 = z_3 \text{xor} ((z_2 \text{lshift}_w s) \text{bitand} b)$.

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 $z_1 = X_i \text{xor} (X_i \text{rshift} u)$.

b) Let $z_2 = z_1 \text{xor} ((z_1 \text{lshift}_u s) \text{bitand} b)$.

c) Let $z_3 = z_2 \text{xor} ((z_2 \text{lshift}_w t) \text{bitand} c)$.

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, size_t s,  
          UIntType b, size_t t,  
          UIntType c, size_t l>
class mersenne_twister_engine
{
    public:
    // types
    typedef UIntType result_type;
}
```

\(^{270}\) 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.
// engine characteristics
static const size_t word_size = w;
static const size_t state_size = n;
static const size_t shift_size = m;
static const size_t mask_bits = r;
static const UIntType xor_mask = a;
static const size_t tempering_u = u;
static const size_t tempering_s = s;
static const UIntType tempering_b = b;
static const size_t tempering_t = t;
static const UIntType tempering_c = c;
static const size_t tempering_l = l;
static const result_type min = 0;
static const result_type max = 2w - 1;
static const result_type default_seed = 5489u;

// constructors and seeding functions
explicit mersenne_twister_engine(result_type value = default_seed);
explicit mersenne_twister_engine(seed_seq& q);
void seed(result_type value = default_seed);
void seed(seed_seq& q);

// generating functions
result_type operator()();
void discard(unsigned long long z);

4 The following relations shall hold: 1 ≤ m ≤ n; 0 ≤ r, u, s, t, l ≤ w ≤ numeric_limits<result_type>::digits;
0 ≤ a, b, c ≤ 2w - 1.

5 The textual representation of xi consists of the values of Xi-n, ..., Xi-1, in that order.

6 Effects: Constructs a mersenne_twister_engine object. Sets X-n to value mod 2w. Then, iteratively for
    i = 1 - n, ..., -1, sets Xi to
            \[1812433253 \cdot (X_{i-1} \text{xor} (X_{i-1} \text{rshift}(w - 2))) + i \text{ mod } n\] mod 2w.

7 Complexity: \(\Theta(n)\).

6 Effects: Constructs a mersenne_twister_engine object. With k = [w/32] and a an array (or equivalent)
of length n \cdot k, invokes q.randomize(a + 0, a + n \cdot k) and then, iteratively for i = -n, ..., -1, sets Xi to
    \(\left(\sum_{j=0}^{k-1} a_{k(i+n)+j} \cdot 2^{32}\right) \text{ mod } 2^w\). Finally, if the most significant w - r bits of X-n are zero, and if each of the other
resulting Xi is 0, changes X-n to 2w-1.
26.4 Random number generation

26.4.3.3 Class template subtract_with_carry_engine

A subtract_with_carry_engine random number engine produces unsigned integer random numbers.

The state $x_i$ of a subtract_with_carry_engine object $x$ is of size $O(r)$, 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.

The state transition is performed as follows:

1. Let $Y = X_{i-s} - X_{i-r} - c$.
2. Set $X_i$ to $y = Y \mod m$. Set $c$ to 1 if $Y < 0$, otherwise set $c$ to 0.

[Note: This algorithm corresponds to a modular linear function of the form $T\mathcal{A}(x_i) = (a \cdot x_i) \mod b$, where $b$ is of the form $m^r - m^s + 1$ and $a = b - (b - 1)/m$. — end note]

The generation algorithm is given by $G\mathcal{A}(x_i) = y$, 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 const size_t word_size = w;
    static const size_t short_lag = s;
    static const size_t long_lag = r;
    static const result_type min = 0;
    static const result_type max = m - 1;
    static const result_type default_seed = 19780503u;

    // constructors and seeding functions
    explicit subtract_with_carry_engine(result_type value = default_seed);
    explicit subtract_with_carry_engine(seed_seq& q);
    void seed(result_type value = default_seed);
    void seed(seed_seq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);
};
```

The following relations shall hold: $0 < s < r$, and $0 < w \leq \text{numeric_limits}<result_type>::\text{digits}$.

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);
```
Effects: Constructs a subtract_with_carry_engine object. Sets the values of \(X_r, \ldots, X_1\), in that order, as required below. If \(X_1\) is then 0, sets \(c\) to 1; otherwise sets \(c\) to 0.

Required behavior: First construct \(e\), a linear_congruential_engine object, as if by the following definition:

\[
\text{linear_congruential_engine<result_type}
\]

\[
\begin{align*}
40014u, 0u, 2147483563u > e(value == 0u ? default_seed : value);
\end{align*}
\]

To set an \(X_k\), use new values \(z_0, \ldots, z_{n-1}\) obtained from \(n\) successive invocations of \(e\) taken modulo \(2^{32}\). Set \(X_k\) to \((\sum_{j=0}^{n-1} z_j \cdot 2^{32j}) \mod m\). If \(X_1\) is then 0, sets \(c\) to 1; otherwise sets \(c\) to 0.

Complexity: Exactly \(n \cdot r\) invocations of \(e\).

explicit subtract_with_carry_engine(seed_seq& 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 \(q\).randomize\((a + 0, a + r \cdot k)\) and then, iteratively for \(i = -r, \ldots, -1\), sets \(X_i\) to \((\sum_{j=0}^{k-1} a_k(i+r) + j \cdot 2^{32j}) \mod m\). If \(X_{-1}\) is then 0, sets \(c\) to 1; otherwise sets \(c\) to 0.

26.4.4 Random number engine adaptor class templates [rand.adapt]

Except where specified otherwise, the complexity of all functions specified in the following sections is constant.

Except as required by table 105, no function described in this section 26.4.4 throws an exception.

The class templates specified in this section 26.4.4 satisfy the requirements of random number engine adaptor (26.4.1.4). Descriptions are provided here only for operations on the engine adaptors that are not described in those requirements or for operations where there is additional semantic information. Declarations for copy constructors, for copy assignment operators, and for equality and inequality operators are not shown in the synopses.

26.4.4.1 Class template discard_block_engine [rand.adapt.disc]

A discard_block_engine random number engine adaptor produces random numbers selected from those produced by some base engine \(e\). The state \(x\) of a discard_block_engine engine adaptor object \(x\) consists of the state \(e_i\) of its base engine \(e\) and an additional integer \(n\). The size of the state is the size of \(e\)’s state plus 1.

The transition algorithm discards all but \(r > 0\) values from each block of \(p \geq r\) values delivered by \(e\). The state transition is performed as follows: If \(n \geq r\), advance the state of \(e\) from \(e_i\) to \(e_{i+p-r}\) and set \(n\) to 0. In any case, then increment \(n\) and advance \(e\)’s then-current state \(e_j\) to \(e_{j+1}\).

The generation algorithm yields the value returned by the last invocation of \(e()\) while advancing \(e\)’s state as described above.

\[
\text{template <class Engine, size_t p, size_t r>}
\]

\[
\text{class discard_block_engine}
\]

\[
\{
\text{public:}
\]

\[
// types
\]

\[
\text{typedef Engine base_type;}
\]

\[
\text{typedef typename base_type::result_type result_type;}
\]
26.4 Random number generation

// engine characteristics
static const size_t block_size = p;
static const size_t used_block = r;
static const result_type min = base_type::min;
static const result_type max = base_type::max;

// constructors and seeding functions
discard_block_engine();
explicit discard_block_engine(const base_type& urng);
explicit discard_block_engine(result_type s);
void seed();
void seed(result_type s);
void seed(seed_seq& q);

// generating functions
result_type operator()();
void discard(unsigned long long z);

// property functions
const base_type& base() const;

private:
base_type e; // exposition only
int n; // exposition only

The following relations shall hold: 1 ≤ 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 26.4.1.4, each constructor that is not a copy constructor sets n to 0.

26.4.4.2 Class template independent_bits_engine [rand.adapt.ibits]

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 \( w_0 = \left\lfloor \frac{w}{n} \right\rfloor \), \( n_0 = n - w \mod n \), \( y_0 = 2^{w_0} \left\lfloor \frac{R}{2^{w_0}} \right\rfloor \), and \( y_1 = 2^{w_0+1} \left\lfloor \frac{R}{2^{w_0+1}} \right\rfloor \).

c) Let \( n = \left\lfloor \frac{w}{m} \right\rfloor \) if and only if the relation \( R - y_0 \leq \left\lfloor \frac{y_0}{n} \right\rfloor \) holds as a result. Otherwise let \( n = 1 + \left\lfloor \frac{w}{m} \right\rfloor \).

[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:

$$S = 0;$$

for ($k = 0; k \neq n_0; k += 1$) {
    do $u = \text{e}() - \text{e.min}$; while ($u \geq y_0$);
    $S = 2^w \cdot S + u \mod 2^w$;
}

for ($k = n_0; k \neq n; k += 1$) {
    do $u = \text{e}() - \text{e.min}$; while ($u \geq y_1$);
    $S = 2^w+1 \cdot S + u \mod 2^w+1$;
}

template <class Engine, size_t w, class UIntType>
class independent_bits_engine
{
public:
    // types
    typedef Engine base_type;
    typedef UIntType result_type;

    // engine characteristics
    static const size_t word_size = w;
    static const result_type min = 0;
    static const result_type max = $2^w - 1$;

    // constructors and seeding functions
    independent_bits_engine();
    explicit independent_bits_engine(const base_type& urng);
    explicit independent_bits_engine(result_type s);
    explicit independent_bits_engine(seed_seq& q);
    void seed();
    void seed(result_type s);
    void seed(seed_seq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);

    // property functions
    const base_type& base() const;

private:
    base_type e; // exposition only
};

The following relations shall hold: $0 < w \leq \text{numeric_limits<result_type>::digits}$.
The textual representation consists of the textual representation of $\text{e}$. 

Draft
26.4 Random number generation

26.4.4.3 Class template shuffle_order_engine

A shuffle_order_engine random number engine adaptor produces the same random numbers that are produced by some base engine $e$, but delivers them in a different sequence. The state $x_i$ of a shuffle_order_engine engine adaptor object $x$ consists of the state $e_i$ of its base engine $e$, an additional value $Y$ of the type delivered by $e$, and an additional sequence $V$ of $k$ values also of the type delivered by $e$. The size of the state is the size of $e$’s state plus $k + 1$.

The transition algorithm permutes the values produced by $e$. The state transition is performed as follows:

a) Calculate an integer $j$ as:
$$j = \left\lfloor \frac{k(Y - b_{\min})}{b_{\max} - b_{\min} + 1} \right\rfloor$$

b) Set $Y$ to $V_j$ and then set $V_j$ to $b()$.

The generation algorithm yields the last value of $Y$ produced while advancing $e$’s state as described above.

The following relation shall hold: $1 \leq k$.
The textual representation consists of the textual representation of \( e \), followed by the \( k \) values of \( V \), followed by the value of \( Y \).

In addition to its behavior pursuant to section 26.4.1.4, each constructor that is not a copy constructor initializes \( V[0], \ldots, V[k-1] \) and \( Y \), in that order, with values returned by successive invocations of \( e() \).

### 26.4.4.4 Class template xor_combine_engine

An `xor_combine_engine` random number engine adaptor produces random numbers from two integer-valued base engines \( e1 \) and \( e2 \) by merging their left-shifted random values via bitwise exclusive-or. The state \( x_i \) of an `xor_combine_engine` engine adaptor object \( x \) consists of the states \( e1 \) and \( e2 \) of its base engines. The size of the state is the size of the state of \( e1 \) plus the size of the state of \( e2 \).

The transition algorithm advances, in turn, the state of each base engine.

The generation algorithm is \( GA(x_i) = (v_1 lshift_w s1) \xor (v_2 lshift_w s2) \), where \( w \) denotes the value of `numeric_limits<result_type>::digits` and \( v_1 \) and \( v_2 \), respectively, denote the values of \( (e1()-e1.min) \) and \( (e2()-e2.min) \).

```cpp
template <class Engine1, size_t s1, class Engine2, size_t s2=0u>
class xor_combine_engine
{
public:
  // types
  typedef Engine1 base1_type;
  typedef Engine2 base2_type;
  typedef see below result_type;

  // engine characteristics
  static const size_t shift1 = s1;
  static const size_t shift2 = s2;
  static const result_type min = 0;
  static const result_type max = see below;

  // constructors and seed functions
  xor_combine_engine();
  xor_combine_engine(const base1_type & urng1, const base2_type & urng2);
  xor_combine_engine(result_type s);
  explicit xor_combine_engine(seed_seq& q);
  void seed();
  void seed(result_type s);
  void seed(seed_seq& q);

  // generating functions
  result_type operator()();
  void discard(unsigned long long z);

  // property functions
  const base1_type& base1() const;
  const base2_type& base2() const;
};
```

Draft
26.4 Random number generation

private:
    base1_type e1; // exposition only
    base2_type e2; // exposition only
};

The following relations shall hold: \( s_1 \geq s_2 \geq 0 \).

[Note: An xor_combine_engine engine adaptor that fails to observe the following recommendations may have significantly worse uniformity properties than either of the base engines it is based on:

a) While two shift values (template parameters \( s_1 \) and \( s_2 \)) are provided for simplicity of interface, it is advisable that \( s_2 \) be zero. (If \( s_2 \) is non-zero then the low bits will always be 0.)

b) It is also advisable for \( e_2 \)'s max to be \( 2^n - 1 - \text{min} \) for some non-negative integer \( n \), and for the shift value \( s_1 \) to be no greater than that \( n \).

— end note]

Both \( \text{Engine1}::\text{result\_type} \) and \( \text{Engine2}::\text{result\_type} \) shall denote (possibly different) unsigned integral types. The member \( \text{result\_type} \) shall denote either the type \( \text{Engine1}::\text{result\_type} \) or the type \( \text{Engine2}::\text{result\_type} \), whichever provides the most storage according to clause 3.9.1.

With

\[
\begin{align*}
    a) & \quad m_1 = \text{min}(\text{Engine1}::\text{max} - \text{Engine1}::\text{min}, 2^{w-s_1} - 1), \\
    b) & \quad m_2 = \text{min}(\text{Engine2}::\text{max} - \text{Engine2}::\text{min}, 2^{w-s_2} - 1), \text{ and} \\
    c) & \quad s = s_1 - s_2,
\end{align*}
\]

the value of the member \( \text{max} \) is \( M(m_1, m_2, s) \text{lshift}_w s_2 \), where \( M(a, b, d) \) is defined as follows:

If \( a = 0 \) or \( b < 2^d \), define \( M(a, b, d) \) as \( a \cdot 2^d + b \).

Otherwise, let \( t \) and \( u \) denote the greater and the lesser, respectively, of \( a \cdot 2^d \) and \( b \). With \( p = \lfloor \log_2 u \rfloor \), if \( k = \lfloor t/2^p \rfloor \) is odd, define \( M(a, b, d) \) as \((k+1) \cdot 2^p - 1\).

Otherwise, if \( a \cdot 2^d \geq b \), define \( M(a, b, d) \) as \((k+1) \cdot 2^p + M((t \text{ mod } 2^p)/2^d, u \text{ mod } 2^p, d)\).

Otherwise, define \( M(a, b, d) \) as \((k+1) \cdot 2^p + M((u \text{ mod } 2^p)/2^d, t \text{ mod } 2^p, d)\).

The textual representation consists of the textual representation of \( e_1 \) followed by the textual representation of \( e_2 \).

26.4.5 Engines and engine adaptors with predefined parameters

typedef linear_congruential_engine<uint_fast32_t, 16807, 0, 2147483647> minstd_rand0;

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.

```cpp
typedef mersenne_twister_engine<uint_fast32_t, 32, 624, 397, 31, 0x9908b0df, 11, 7, 0x9d2c5680, 15, 0xefc60000, 18> mt19937;
```

3 **Required behavior:** The 10000th consecutive invocation of a default-constructed object of type `mt19937` shall produce the value 4123659995.

```cpp
typedef subtract_with_carry_engine<uint_fast32_t, 24, 10, 24> ranlux24_base;
```

4 **Required behavior:** The 10000th consecutive invocation of a default-constructed object of type `ranlux24_base` shall produce the value 7937952.

```cpp
typedef subtract_with_carry_engine<uint_fast64_t, 48, 5, 12> ranlux48_base;
```

5 **Required behavior:** The 10000th consecutive invocation of a default-constructed object of type `ranlux48_base` shall produce the value 618391285725.

```cpp
typedef discard_block_engine<ranlux24_base, 223, 23> ranlux24;
```

6 **Required behavior:** The 10000th consecutive invocation of a default-constructed object of type `ranlux24` shall produce the value 9901578.

```cpp
typedef discard_block_engine<ranlux48_base, 389, 11> ranlux48;
```

7 **Required behavior:** The 10000th consecutive invocation of a default-constructed object of type `ranlux48` shall produce the value 249142670248501.

```cpp
typedef shuffle_order_engine<minstd_rand0, 256> knuth_b;
```

8 **Required behavior:** The 10000th consecutive invocation of a default-constructed object of type `knuth_b` shall produce the value 1112339016.

### 26.4.6 Class `random_device` [rand.device]

1 A `random_device` uniform random number generator produces non-deterministic random numbers. It satisfies the requirements of uniform random number generator (26.4.1.2).

2 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 const result_type min = see below;
static const result_type max = see below;

// constructors
explicit random_device(const string& token = implementation-defined);

// generating functions
result_type operator()();

// property functions
double entropy() const;

private:
    random_device(const random_device&) = delete;
    void operator=(const random_device&) = delete;
};

The values of the min and max members are identical to the values returned by numeric_limits<result_type>::min() and numeric_limits<result_type>::max(), respectively.

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.\(^\text{271}\)

Throws: A value of an implementation-defined type derived from exception if the random_device could not be initialized.

double entropy() const;

Returns: If the implementation employs a random number engine, returns 0.0. Otherwise, returns an entropy estimate\(^\text{272}\) for the random numbers returned by operator(), in the range \(\min\) to \(\log_2(\max + 1)\).

Throws: Nothing.

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.

\(^{271}\) The parameter is intended to allow an implementation to differentiate between different sources of randomness.

\(^{272}\) If a device has \(n\) states whose respective probabilities are \(P_0, \ldots, P_{n-1}\), the device entropy \(S\) is defined as \(S = -\sum_{i=0}^{n-1} P_i \cdot \log P_i\).
26.4.7 Utilities

26.4.7.1 Class seed_seq

An object of type seed_seq consumes a sequence of integer-valued data and produces a fixed number of unsigned integer values, $0 \leq i < 2^{32}$, based on the consumed data. [Note: Such an object provides a mechanism to avoid replication of streams of random variates. This can be useful in applications requiring large numbers of random number engines. — end note]

In addition to the requirements set forth below, instances of seed_seq shall meet the requirements of CopyConstructible (34) and of Assignable (23.1).

```cpp
class seed_seq
{
public:
    // types
    typedef uint_least32_t result_type;

    // constructors
    seed_seq();
    template<class InputIterator>
    seed_seq(InputIterator begin, InputIterator end);

    // generating functions
    template<class RandomAccessIterator>
    void randomize(RandomAccessIterator begin, RandomAccessIterator end) const;

    // property functions
    size_t size() const;
    template<class OutputIterator>
    void get_seeds(OutputIterator dest) const;

private:
    vector<result_type> v;       // exposition only
};
```

**Effects:** Constructs a seed_seq object as if by default-constructor its member v.

**Throws:** Nothing.

**Requires:** InputIterator shall satisfy the requirements of an input iterator (24.1.1) such that iterator_traits<InputIterator>::value_type shall denote an integral type.

**Effects:** Constructs a seed_seq object by rearranging the bits of the supplied sequence [begin, end) into 32-bit units, as if by first concatenating all the $n$ bits that make up the supplied sequence to initialize a single (possibly very large) unsigned binary number, $b$, and then carrying out the following algorithm:

```cpp
for (v.clear(); n > 0; n -= 32)
    v.push_back(b mod $2^{32}$);  b /= $2^{32}$;
```

Draft
template<class RandomAccessIterator>
void randomize(RandomAccessIterator begin, RandomAccessIterator end) const;

Requires: RandomAccessIterator shall meet the requirements of a random access iterator (24.1.5) such that iterator_traits<RandomAccessIterator>::value_type shall denote an unsigned integral type capable of accommodating 32-bit quantities.

Effects: With \( s = v\.\text{size()} \) and \( n = \text{end} - \text{begin} \), fills the supplied range \([\text{begin}, \text{end})\) according to the following algorithm in which each operation is to be carried out modulo \( 2^{32} \), each indexing operator applied to \( \text{begin} \) is to be taken modulo \( n \), each indexing operator applied to \( v \) is to be taken modulo \( s \), and \( T(x) \) is defined as \( x \oplus (x\ll 30) \):

a) Set \( \text{begin}[0] \) to \( 5489 + s \). Then, iteratively for \( k = 1, \ldots, n - 1 \), set \( \text{begin}[k] \) to

\[
1812433253 \cdot T(\text{begin}[k-1]) + k .
\]

b) With \( m \) as the larger of \( s \) and \( n \), transform each element of the range (possibly more than once): iteratively for \( k = 0, \ldots, m - 1 \), set \( \text{begin}[k] \) to

\[
(\text{begin}[k] \oplus (1664525 \cdot T(\text{begin}[k-1]))) + v[k] + (k \bmod s) .
\]

c) Transform each element of the range one last time, beginning where the previous step ended: iteratively for \( k = m \bmod n, \ldots, n - 1, 0, \ldots, (m - 1) \bmod n \), set \( \text{begin}[k] \) to

\[
(\text{begin}[k] \oplus (1566083941 \cdot T(\text{begin}[k-1]))) - k .
\]

Throws: Nothing.

size_t size() const;

Returns: The number of 32-bit units the object can deliver, as if by returning the result of \( v\.\text{size()} \).

template<class OutputIterator> void get_seeds(OutputIterator dest) const;

Requires: OutputIterator shall satisfy the requirements of an output iterator (24.1.2) such that iterator_traits<OutputIterator>::value_type shall be assignable from result_type.

Effects: Copies the sequence of prepared 32-bit units to the given destination, as if by executing the following statement:

\[
\text{copy}(v\.\text{begin}(), v\.\text{end}(), \text{dest});
\]

26.4.7.2 Function template generate_canonical [rand.util.canonical]

Each function instantiated from the template described in this section 26.4.7.2 maps the result of one or more invocations of a supplied uniform random number generator \( g \) to one member of the specified \( \text{RealType} \) such that, if the values \( g_i \) produced by \( g \) are uniformly distributed, the instantiation’s results \( t_j, 0 \leq t_j < 1 \), are distributed as uniformly as possible as specified below.

[ Note: Obtaining a value in this way can be a useful step in the process of transforming a value generated by a uniform random number generator into a value that can be delivered by a random number distribution. — end note ]

Draft
template<class RealType, class UniformRandomNumberGenerator, size_t bits>
RealType generate_canonical(UniformRandomNumberGenerator& g);

Complexity: Exactly $k = \max(1, \lceil b/\log_2 R \rceil)$ invocations of $g$, where $b$\(^{273}\) is the lesser of \texttt{numeric_limits<}
\texttt{RealType>::digits and bits, and $R$ is the value of $g$.\texttt{max} - g$.\texttt{min} + 1.

Required behavior: Invokes $g()$ $k$ times to obtain values $g_0, \ldots, g_{k-1}$, respectively. Calculates a quantity

$$S = \sum_{i=0}^{k-1} (g_i - g$.\texttt{min}) \cdot R^i$$

using arithmetic of type \texttt{RealType}.

Returns: $S/R^k$.

Throws: What and when $g$ throws.

26.4.8 Random number distribution class templates

The classes and class templates specified in this section 26.4.8 satisfy all the requirements of random number distribution (26.4.1.5). Descriptions are provided here only for operations on the distributions that are not described in those requirements or for operations where there is additional semantic information. Declarations for copy constructors, for copy assignment 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.

26.4.8.1 Uniform distributions

26.4.8.1.1 Class template uniform_int_distribution

A \texttt{uniform_int_distribution} random number distribution produces random integers $i$, $a \leq i \leq b$, distributed according to the constant discrete probability function

$$P(i|a,b) = 1/(b-a+1).$$

\footnote{\texttt{b} is introduced to avoid any attempt to produce more bits of randomness than can be held in \texttt{RealType}.}
26.4 Random number generation

```cpp
void reset();

// generating functions
template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, 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;
```

```cpp
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.

```cpp
result_type a() const;
4 Returns: The value of the \(a\) parameter with which the object was constructed.
```

```cpp
result_type b() const;
5 Returns: The value of the \(b\) parameter with which the object was constructed.
```

26.4.8.1.2 Class template `uniform_real_distribution` [rand.dist.uni.real]

1 A `uniform_real_distribution` random number distribution produces random numbers \(x\), \(a \leq x < b\), distributed according to the constant probability density function

\[
p(x|a,b) = \frac{1}{(b-a)}.
\]

```cpp
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();

// generating functions
template <class UniformRandomNumberGenerator>
result_type operator()(UniformRandomNumberGenerator& urng);
template <class UniformRandomNumberGenerator>
result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

// property functions
result_type a() const;
result_type b() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;

explicit uniform_real_distribution(RealType a = 0.0, RealType b = 1.0);

Requires: \( a \leq b \) and \( b - a \leq \text{numeric_limits<RealType>::max()} \).
Effects: Constructs a uniform_real_distribution object; \( a \) and \( b \) correspond to the respective parameters of the distribution.

result_type a() const;
Returns: The value of the \( a \) parameter with which the object was constructed.

result_type b() const;
Returns: The value of the \( b \) parameter with which the object was constructed.

26.4.8.2 Bernoulli distributions

26.4.8.2.1 Class bernoulli_distribution

A bernoulli_distribution random number distribution produces bool values \( b \) distributed according to the discrete probability function
\[
P(b|p) = \begin{cases} 
  p & \text{if } b = \text{true} \\
  1 - p & \text{if } b = \text{false}
\end{cases}
\]

class bernoulli_distribution
{
public:
// types
  typedef bool result_type;
  typedef unspecified param_type;

// constructors and reset functions
  explicit bernoulli_distribution(double p = 0.5);
# 26.4 Random number generation

```cpp
explicit bernoulli_distribution(const param_type& parm);
void reset();

// generating functions
template <class UniformRandomNumberGenerator>
result_type operator()(UniformRandomNumberGenerator& urng);
template <class UniformRandomNumberGenerator>
result_type operator()(UniformRandomNumberGenerator& urng, 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.
```

## 26.4.8.2.2 Class template `binomial_distribution`

A binomial_distribution random number distribution produces integer values \( i \geq 0 \) distributed according to the discrete probability function

\[
P(i|t, p) = \binom{t}{i} \cdot p^i \cdot (1 - p)^{t-i}.
\]

```cpp
template <class IntType = int>
class binomial_distribution
{
public:
    // types
    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 UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);

    Draft
```
template <class UniformRandomNumberGenerator>
result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

// property functions
IntType t() const;
double p() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;
};

explicit binomial_distribution(IntType t = 1, double p = 0.5);

Requires: \(0 \leq p \leq 1\) and \(0 \leq t\).
Effects: Constructs a binomial_distribution object; \(t\) and \(p\) correspond to the respective parameters of the distribution.

IntType t() const;
Returns: The value of the \(t\) parameter with which the object was constructed.

double p() const;
Returns: The value of the \(p\) parameter with which the object was constructed.

26.4.8.2.3 Class template geometric_distribution

A geometric_distribution random number distribution produces integer values \(i \geq 0\) distributed according to the discrete probability function

\[ P(i|p) = p \cdot (1-p)^i. \]

template <class IntType = int>
class geometric_distribution
{
public:
// types
typedef IntType result_type;
typedef unspecified param_type;

// constructors and reset functions
explicit geometric_distribution(double p = 0.5);
explicit geometric_distribution(const param_type& parm);
void reset();

// generating functions
template <class UniformRandomNumberGenerator>
result_type operator()(UniformRandomNumberGenerator& urng);
template <class UniformRandomNumberGenerator>
26.4 Random number generation

result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

// property functions
double p() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;

explicit geometric_distribution(double p = 0.5);

2 Requires: 0 < p < 1.
3 Effects: Constructs a geometric_distribution object; p corresponds to the parameter of the distribution.

double p() const;
4 Returns: The value of the p parameter with which the object was constructed.

26.4.8.2.4 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

\[
P(i|k, p) = \binom{k+i-1}{i} \cdot p^k \cdot (1-p)^i.
\]

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 UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    IntType k() const;
explicit negative_binomial_distribution(IntType k = 1, double p = 0.5);

2    Requires: 0 < p ≤ 1 and 0 < k.
3    Effects: Constructs a negative_binomial_distribution object; k and p correspond to the respective parameters of the distribution.

IntType k() const;
4    Returns: The value of the k parameter with which the object was constructed.

double p() const;
5    Returns: The value of the p parameter with which the object was constructed.

26.4.8.3 Poisson distributions

26.4.8.3.1 Class template poisson_distribution

1 A poisson_distribution random number distribution produces integer values i ≥ 0 distributed according to the discrete probability function

\[ P(i | \mu) = \frac{e^{-\mu} \mu^i}{i!} \, . \]

The distribution parameter \( \mu \) is also known as this distribution’s mean. 

template <class IntType = int>
class poisson_distribution
{
public:
    // types
    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 UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, 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);

Requires: 0 < mean.

Effects: Constructs a poisson_distribution object; mean corresponds to the parameter of the distribution.

double mean() const;

Returns: The value of the mean parameter with which the object was constructed.

26.4.8.3.2 Class template exponential_distribution

An exponential_distribution random number distribution produces random numbers x > 0 distributed according to the probability density function

\[ p(x | \lambda) = \lambda e^{-\lambda x}. \]

template <class RealType = double>
class exponential_distribution
{
public:
    // types
typedef RealType result_type;
typedef unspecified param_type;

    // constructors and reset functions
    explicit exponential_distribution(RealType lambda = 1.0);
    explicit exponential_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    RealType lambda() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
explicit exponential_distribution(RealType lambda = 1.0);

// Requires: 0 < lambda.
// Effects: Constructs a exponential_distribution object; lambda corresponds to the parameter of the distribution.

RealType lambda() const;
// Returns: The value of the lambda parameter with which the object was constructed.

26.4.8.3.3 Class template gamma_distribution[rand.dist.pois.gamma]

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 UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    RealType alpha() const;
    RealType beta() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};

explicit gamma_distribution(RealType alpha = 1.0, RealType beta = 1.0);

Draft
26.4 Random number generation

Requirements: $0 < \alpha$ and $0 < \beta$.

Effects: Constructs a \texttt{gamma\_distribution} object; \(\alpha\) and \(\beta\) correspond to the parameters of the distribution.

Returns: The value of the \(\alpha\) parameter with which the object was constructed.

Returns: The value of the \(\beta\) parameter with which the object was constructed.

26.4.8.3.4 Class template \texttt{weibull\_distribution} \[\texttt{rand.dist.pois.weibull}\]

A \texttt{weibull\_distribution} random number distribution produces random numbers \(x \geq 0\) distributed according to the probability density function

\[
p(x|a,b) = \frac{a}{b} \left(\frac{x}{b}\right)^{a-1} \exp\left(-\left(\frac{x}{b}\right)^a\right).
\]

template <class RealType = double>
class weibull_distribution
{
    public:
    // types
    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 UniformRandomNumberGenerator>
    result_type operator() (UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
    result_type operator() (UniformRandomNumberGenerator& urng, 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);
26.4 Random number generation

26.4.8.3.5 Class template extreme_value_distribution

An extreme_value_distribution random number distribution produces random numbers \( x \) distributed according to the probability density function

\[
p(x|a, b) = \frac{1}{b} \cdot \exp \left( \frac{a - x}{b} - \exp \left( \frac{a - x}{b} \right) \right).
\]

```
template <class RealType = double>
class extreme_value_distribution
{
  public:
    // types
    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 UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, 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;
};
```

The distribution corresponding to this probability density function is also known (with a possible change of variable) as the Gumbel Type I, the log-Weibull, or the Fisher-Tippett Type I distribution.
26.4 Random number generation

explicit extreme_value_distribution(RealType a = 0.0, RealType b = 1.0);

Requires: 0 < b.

Effects: Constructs an extreme_value_distribution object; a and b correspond to the respective parameters of the distribution.

RealType a() const;

Returns: The value of the a parameter with which the object was constructed.

RealType b() const;

Returns: The value of the b parameter with which the object was constructed.

26.4.8.4 Normal distributions

26.4.8.4.1 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 UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, 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;

    Draft
explicit normal_distribution(RealType mean = 0.0, RealType stddev = 1.0);

requires: 0 < stddev.

effects: Constructs a normal_distribution object; mean and stddev correspond to the respective parameters of the distribution.

RealType mean() const;

returns: The value of the mean parameter with which the object was constructed.

RealType stddev() const;

returns: The value of the stddev parameter with which the object was constructed.

26.4.8.4.2 Class template lognormal_distribution

A lognormal_distribution random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$p(x|m,s) = \frac{1}{sx\sqrt{2\pi}} \cdot \exp \left( -\frac{(\ln x - m)^2}{2s^2} \right).$$

template <class RealType = double>
class lognormal_distribution
{
  public:
    // types
    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 UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, 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;

    Draft
26.4 Random number generation

```cpp
};

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.

26.4.8.4.3 Class template chi_squared_distribution

A chi_squared_distribution random number distribution produces random numbers \( x > 0 \) distributed according to the probability density function

\[
p(x|n) = \frac{x^{(n/2) - 1} \cdot e^{-x/2}}{\Gamma(n/2) \cdot 2^{n/2}},
\]

where \( n \) is a positive integer.

```
explicit chi_squared_distribution(int n = 1);

2 Requires: 0 < n.

3 Effects: Constructs a chi_squared_distribution object; n corresponds to the parameter of the distribution.

int n() const;

4 Returns: The value of the n parameter with which the object was constructed.

26.4.8.4.4 Class template cauchy_distribution [rand.dist.norm.cauchy]

A cauchy_distribution random number distribution produces random numbers \( x \) distributed according to the probability density function

\[
p(x|a,b) = \frac{1}{\pi b \left( 1 + \left( \frac{x-a}{b} \right)^2 \right) ^{\frac{1}{2}}}
\]

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 UniformRandomNumberGenerator>
 result_type operator()(UniformRandomNumberGenerator& urng);
 template <class UniformRandomNumberGenerator>
 result_type operator()(UniformRandomNumberGenerator& urng, 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);

Draft
26.4 Random number generation

26.4.8.4.5 Class template fisher_f_distribution

A fisher_f_distribution random number distribution produces random numbers \( x \geq 0 \) distributed according to the probability density function

\[
p(x|m,n) = \frac{\Gamma((m+n)/2)}{\Gamma(m/2) \Gamma(n/2)} \cdot \left(\frac{m}{n}\right)^{m/2} \cdot x^{(m/2)-1} \cdot \left(1 + \frac{mx}{n}\right)^{-(m+n)/2},
\]

where \( m \) and \( n \) are positive integers.

```cpp
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(int m = 1, int n = 1);
        explicit fisher_f_distribution(const param_type& parm);
        void reset();

        // generating functions
        template <class UniformRandomNumberGenerator>
        result_type operator() (UniformRandomNumberGenerator& urng);
        template <class UniformRandomNumberGenerator>
        result_type operator() (UniformRandomNumberGenerator& urng, const param_type& parm);

        // property functions
        int m() const;
        int n() const;
        param_type param() const;
        void param(const param_type& parm);
        result_type min() const;
        result_type max() const;
};
```
explicit fisher_f_distribution(int m = 1, int n = 1);

Requires: 0 < m and 0 < n.

Effects: Constructs a fisher_f_distribution object; m and n correspond to the respective parameters of the distribution.

int m() const;

Returns: The value of the m parameter with which the object was constructed.

int n() const;

Returns: The value of the n parameter with which the object was constructed.

26.4.8.4.6 Class template student_t_distribution [rand.dist.norm.t]

A student_t_distribution random number distribution produces random numbers x distributed according to the probability density function

\[ p(x|\nu) = \frac{1}{\sqrt{\nu \pi}} \frac{\Gamma((\nu+1)/2)}{\Gamma(\nu/2)} \left( 1 + \frac{x^2}{\nu} \right)^{-(\nu+1)/2} \]

where \(\nu\) is a positive integer.

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(int n = 1);
  explicit student_t_distribution(const param_type& parm);
  void reset();

  // generating functions
  template <class UniformRandomNumberGenerator>
  result_type operator()(UniformRandomNumberGenerator& urng);
  template <class UniformRandomNumberGenerator>
  result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

  // property functions
  int n() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
};
explicit student_t_distribution(int n = 1);

Requires: 0 < n.

Effects: Constructs a student_t_distribution object; n and n correspond to the respective parameters of the distribution.

int n() const;

Returns: The value of the n parameter with which the object was constructed.

26.4.8.5 Sampling distributions

26.4.8.5.1 Class template discrete_distribution

A discrete_distribution random number distribution produces random integers \( i, 0 \leq i < n \), distributed according to the discrete probability function

\[
P(i | p_0, \ldots, p_{n-1}) = p_i.
\]

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);
    explicit discrete_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
        result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
        result_type operator()(UniformRandomNumberGenerator& urng, 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;
};
Effects: Constructs a discrete_distribution object with \( n = 1 \) and \( p_0 = 1 \). [Note: Such an object will always deliver the value 0. — end note]

```cpp
template <class InputIterator>
discrete_distribution(InputIterator firstW, InputIterator lastW);
```

Requires:

a) InputIterator shall satisfy the requirements of an input iterator (24.1.1).

b) If `firstW == lastW`, let the sequence \( w \) have length \( n = 1 \) and consist of the single value \( w_0 = 1 \). Otherwise, \([firstW,lastW)\) shall form a sequence \( w \) of length \( n > 0 \) and \(*firstW\) shall yield a value \( w_0 \) convertible to double. [Note: The values \( w_k \) are commonly known as the weights. — end note]

c) The following relations shall hold: \( w_k \geq 0 \) for \( k = 0, \ldots, n - 1 \), and \( 0 < S = w_0 + \cdots + w_{n-1} \).

Effects: Constructs a discrete_distribution object with probabilities

\[
p_k = \frac{w_k}{S} \text{ for } k = 0, \ldots, n - 1.
\]

```cpp
discrete_distribution(InputIterator firstW, InputIterator lastW);
```

Returns: A `vector<double>` whose size member returns \( n \) and whose `operator[]` member returns \( p_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n - 1 \).

### 26.4.8.5.2 Class template `piecewise_constant_distribution`

A `piecewise_constant_distribution` random number distribution produces random numbers \( x, b_0 \leq x < b_n \), uniformly distributed over each subinterval \([b_i, b_{i+1})\) according to the probability density function

\[
p(x|b_0,\ldots,b_n,\rho_0,\ldots,\rho_{n-1}) = \rho_i, \text{ for } b_i \leq x < b_{i+1}.
\]

The \( n + 1 \) distribution parameters \( b_i \) are also known as this distribution’s *interval boundaries*.

```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);
        explicit piecewise_constant_distribution(const param_type& parm);
        void reset();
};
```
// generating functions
template <class UniformRandomNumberGenerator>
result_type operator()(UniformRandomNumberGenerator& urng);
template <class UniformRandomNumberGenerator>
result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

// property functions
vector<RealType> intervals() const;
vector<double> densities() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;

piecewise_constant_distribution();

Effects: Constructs a piecewise_constant_distribution object with \( n = 1, \rho_0 = 1, b_0 = 0, \) and \( b_1 = 1. \)

template <class InputIteratorB, class InputIteratorW>
piecewise_constant_distribution(InputIteratorB firstB, InputIteratorB lastB, InputIteratorW firstW);

Requires:

a) InputIteratorB shall satisfy the requirements of an input iterator (24.1.1), as shall InputIteratorW.
b) If firstB == lastB,
   (a) let the sequence \( w \) have length \( n = 1 \) and consist of the single value \( w_0 = 1, \) and
   (b) let the sequence \( b \) have length \( n + 1 \) with \( b_0 = 0 \) and \( b_1 = 1. \)

Otherwise,

(c) \( \text{[firstB, lastB)} \) shall form a sequence \( b \) of length \( n + 1 \) whose leading element \( b_0 \) shall be convertible
   to result_type, and
   (d) the length of the sequence \( w \) starting from firstW shall be at least \( n, *\text{firstW} \) shall return a value \( w_0 \)
   that is convertible to double, and any \( w_k \) for \( k \geq n \) shall be ignored by the distribution.

[Note: The values \( w_k \) are commonly known as the weights. — end note]

c) The following relations shall hold for \( k = 0, \ldots, n - 1: b_k < b_{k+1} \) and \( 0 \leq w_k. \) Also, \( 0 < S = w_0 + \cdots + w_{n-1}. \)

Effects: Constructs a piecewise_constant_distribution object with probability densities

\[
\rho_k = \frac{w_k}{S \cdot (b_{k+1} - b_k)} \quad \text{for} \ k = 0, \ldots, n - 1.
\]

vector<result_type> intervals() const;

Returns: A vector<result_type> whose size member returns \( n + 1 \) and whose operator[] member returns \( b_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n. \)
vector<double> 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 \( k = 0, \ldots, n - 1 \).

26.4.8.5.3 Class template general_pdf_distribution

A general_pdf_distribution random number distribution produces random numbers \( x, x_{\min} \leq x < x_{\max} \), distributed according to the probability density function

\[
p(x| x_{\min}, x_{\max}, \rho) = \rho(x), \text{ for } x_{\min} \leq x < x_{\min}.
\]

template <class RealType = double>
class general_pdf_distribution
{
    public:
        // types
        typedef RealType result_type;
        typedef unspecified param_type;

        // constructor and reset functions
        general_pdf_distribution();
        template <class Func>
        general_pdf_distribution(result_type xmin, result_type xmax, Func pdf);
        explicit general_pdf_distribution(const param_type& parm);
        void reset();

        // generating functions
        template <class UniformRandomNumberGenerator>
        result_type operator() (UniformRandomNumberGenerator& urng);
        template <class UniformRandomNumberGenerator>
        result_type operator() (UniformRandomNumberGenerator& urng, const param_type& parm);

        // property functions
        result_type xmin() const;
        result_type xmax() const;
        param_type param() const;
        void param(const param_type& parm);
        result_type min() const;
        result_type max() const;
    };

general_pdf_distribution();

Effects: Constructs a general_pdf_distribution object with \( x_{\min} = 0 \) and \( x_{\max} = 1 \) such that \( p(x) = 1 \) for all \( x_{\min} \leq x < x_{\max} \).

template <class Func>
general_pdf_distribution(result_type xmin, result_type xmax, Func pdf);
26.5 Numeric arrays

Requires:

a) \( \text{pdf} \) shall be callable with one argument of type \texttt{result\_type}, and shall return values of a type convertible to \texttt{double};

b) \( x_{\text{min}} < x_{\text{max}} \), and for all \( x_{\text{min}} \leq x < x_{\text{max}} \), \( \text{pdf}(x) \) shall return a value that is non-negative, non-NaN, and non-infinity; and

c) the following relations shall hold:

\[
0 < z = \int_{x_{\text{min}}}^{x_{\text{max}}} f(x) \, dx < \infty ,
\]

where \( f \) is the mathematical function corresponding to the supplied \( \text{pdf} \). [Note: This implies that the user-supplied \( \text{pdf} \) need not be normalized. — end note]

Effects: Constructs a \texttt{general\_pdf\_distribution} object; \( x_{\text{min}} \) and \( x_{\text{max}} \) correspond to the respective parameters of the distribution and the corresponding probability density function is given by \( \rho(x) = f(x)/z \).

\texttt{result\_type xmin() const;}

Returns: The value of the \( x_{\text{min}} \) parameter with which the object was constructed.

\texttt{result\_type xmax() const;}

Returns: The value of the \( x_{\text{max}} \) parameter with which the object was constructed.

26.5 Numeric arrays

26.5.1 Header \texttt{<valarray> synopsis}

namespace std {

  template<class T> class valarray;  // An array of type \( T \)
  class slice;                      // a BLAS-like slice out of an array
  template<class T> class slice_array;
  class gslice;                     // a generalized slice out of an array
  template<class T> class gslice_array;
  template<class T> class mask_array; // a masked array
  template<class T> class indirect_array; // an indirected array

  template<class T> void swap(valarray<T>&, valarray<T>&);
  template<class T> void swap(valarray<T>&&, valarray<T>&);
  template<class T> void swap(valarray<T>&, 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>&);

Draft
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>&);

template<class T>
  valarray<bool> operator<=(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator<=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator<=(const T&, const valarray<T>&);

template<class T>
  valarray<bool> operator>=(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator>=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator>=(const T&, const valarray<T>&);

template<class T> valarray<T> abs (const valarray<T>&);
template<class T> valarray<T> acos (const valarray<T>&);
template<class T> valarray<T> asin (const valarray<T>&);
template<class T> valarray<T> atan (const valarray<T>&);

template<class T> valarray<T> atan2
  (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> atan2(const valarray<T>&, const T&);
template<class T> valarray<T> atan2(const T&, const valarray<T>&);

template<class T> valarray<T> cos (const valarray<T>&);
template<class T> valarray<T> cosh (const valarray<T>&);
template<class T> valarray<T> exp (const valarray<T>&);
template<class T> valarray<T> log (const valarray<T>&);
template<class T> valarray<T> log10(const valarray<T>&);

template<class T> valarray<T> pow(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> pow(const valarray<T>&, const T&);
template<class T> valarray<T> pow(const T&, const valarray<T>&);

template<class T> valarray<T> sin (const valarray<T>&);
template<class T> valarray<T> sinh (const valarray<T>&);
template<class T> valarray<T> sqrt (const valarray<T>&);
The header `<valarray>` defines five class templates (`valarray`, `slice_array`, `gslice_array`, `mask_array`, and `indirect_array`), two classes (`slice` and `gslice`), and a series of related function templates for representing and manipulating arrays of values.

The `valarray` array classes are defined to be free of certain forms of aliasing, thus allowing operations on these classes to be optimized.

Any function returning a `valarray<T>` is permitted to return an object of another type, provided all the const member functions of `valarray<T>` are also applicable to this type. This return type shall not add more than two levels of template nesting over the most deeply nested argument type.

Implementations introducing such replacement types shall provide additional functions and operators as follows:

- for every function taking a `const valarray<T>&`, identical functions taking the replacement types shall be added;
- for every function taking two `const valarray<T>&` arguments, identical functions taking every combination of `const valarray<T>&` and replacement types shall be added.

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.

These library functions are permitted to throw a `bad_alloc` exception if there are not sufficient resources available to carry out the operation. Note that the exception is not mandated.

### 26.5.2 Class template valarray

```cpp
namespace std {
    template<class T> class valarray {
        public:
            typedef T value_type;

            // 26.5.2.1 construct/destroy:
            valarray();
            explicit valarray(size_t);
            valarray(const T&, size_t);
            valarray(const T*, size_t);
            valarray(const valarray&);
            valarray(valarray&&);
            valarray(const slice_array<T>&);
            valarray(const gslice_array<T>&);
            valarray(const mask_array<T>&);
            valarray(const indirect_array<T>&);

275) Clause 18.2.1 recommends a minimum number of recursively nested template instantiations. This requirement thus indirectly suggests a minimum allowable complexity for valarray expressions.
```
`valarray();`

// 26.5.2.2 assignment:
valarray<T>& operator=(const valarray<T>&);
valarray<T>& operator=(valarray<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>&);

// 26.5.2.3 element access:
const T& operator[](size_t) const;
T& operator[](size_t);

// 26.5.2.4 subset operations:
valarray<T> operator[](slice) const;
slice_array<T> operator[](slice);
valarray<T> operator[](const gslice&) const;
gslice_array<T> operator[](const gslice&);
valarray<T> operator[](const valarray<bool>&) const;
mask_array<T> operator[](const valarray<bool>&);
valarray<T> operator[](const valarray<size_t>&) const;
indirect_array<T> operator[](const valarray<size_t>&);

// 26.5.2.5 unary operators:
valarray<T> operator+() const;
valarray<T> operator-() const;
valarray<T> operator~() const;
valarray<bool> operator!() const;

// 26.5.2.6 computed assignment:
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 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>&);

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.

An implementation is permitted to qualify any of the functions declared in `<valarray>` as `inline`.

### 26.5.2.1 valarray constructors [valarray.cons]

```cpp
class valarray {
public:
    // 26.5.2.7 member functions:
    void swap(valarray&&);
    size_t size() const;
    T sum() const;
    T min() const;
    T max() const;
    valarray<T> shift (int) const;
    valarray<T> cshift(int) const;
    valarray<T> apply(T func(T)) const;
    valarray<T> apply(T func(const T&)) const;
    void resize(size_t sz, T c = T());
};
```

1. **valarray();**
   - **Effects:** Constructs an object of class `valarray<T>`, which has zero length until it is passed into a library function as a modifiable lvalue or through a non-constant `this` pointer.

2. **explicit valarray(size_t);**
   - The array created by this constructor has a length equal to the value of the argument. The elements of the array are constructed using the default constructor for the instantiating type `T`.

3. **valarray(const T&, size_t);**
   - 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.

---

276) 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 `valarray` template is neither a matrix class nor a field class. However, it is a very useful building block for designing such classes.

277) For convenience, such objects are referred to as “arrays” throughout the remainder of 26.5.

278) This default constructor is essential, since arrays of `valarray` are likely to prove useful. There shall also be a way to change the size of an array after initialization; this is supplied by the semantics of the `resize` member function.
26.5 Numeric arrays

valarray(const T*, size_t);

The array created by this constructor has a length equal to the second argument n. The values of the elements of the array are initialized with the first n values pointed to by the first argument. If the value of the second argument is greater than the number of values pointed to by the first argument, the behavior is undefined.

valarray(const valarray<T>&);

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.

valarray(valarray<T>&&);

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. After construction, v is in a valid but unspecified state.

Complexity: Constant.

Throws: Nothing.

valarray(const slice_array<T>&);
valarray(const gslice_array<T>&);
valarray(const mask_array<T>&);
valarray(const indirect_array<T>&);

These conversion constructors convert one of the four reference templates to a valarray.

valarray<T>& operator=(const valarray<T>&);

Each element of the *this array is assigned the value of the corresponding element of the argument array. The resulting behavior is undefined if the length of the argument array is not equal to the length of the *this array.

valarray<T>& operator=(valarray<T>&&);

Effects: *this obtains the value of v. After the assignment, v is in a valid but unspecified state.

Complexity: Constant.

Throws: Nothing.

valarray<T>& operator=(const T&);

The scalar assignment operator causes each element of the *this array to be assigned the value of the argument.

---

279) This constructor is the preferred method for converting a C array to a valarray object.

280) 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.

Draft
These operators allow the results of a generalized subscripting operation to be assigned directly to a valarray.

If the value of an element in the left-hand side of a valarray assignment operator depends on the value of another element in that left-hand side, the resulting behavior is undefined.

### 26.5.2.3 Valarray element access

```cpp
class valarray {
public:
  T& operator[](size_t) const; // When applied to a constant array, the subscript operator returns a reference to the corresponding element of the array. When applied to a non-constant array, the subscript operator returns a reference to the corresponding element of the array.

  const T& operator[](size_t) const; // Thus, the expression (a[i] = q, a[i]) == q evaluates as true for any non-constant valarray<T> a, any T q, and for any size_t i such that the value of i is less than the length of a.

  T& operator[](size_t); // The expression &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 non-constant array a.

  const T& operator[](size_t) const; // Likewise, the expression &a[i] != &b[j] evaluates as true for any two non-constant 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.281)

  T& operator[](size_t); // The reference returned by the subscript operator for a non-constant array is guaranteed to be valid until the member function resize(size_t, T) (26.5.2.7) is called for that array or until the lifetime of that array ends, whichever happens first.

  size_t& operator[](size_t); // 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.

private:

  // valarray subset operations
  operator[](slice) const; // valarray<T> operator[](slice) const;

  slice_array<T> operator[](slice); // slice_array<T> operator[](slice);

  operator[](const gslice&) const; // valarray<T> operator[](const gslice&) const;

  gslice_array<T> operator[](const gslice&); // gslice_array<T> operator[](const gslice&);

  operator[](const valarray<bool>&) const; // valarray<T> operator[](const valarray<bool>&) const;

  mask_array<T> operator[](const valarray<bool>&); // mask_array<T> operator[](const valarray<bool>&);

  operator[](const valarray<size_t>&) const; // valarray<T> operator[](const valarray<size_t>&) const;

  indirect_array<T> operator[](const valarray<size_t>&); // indirect_array<T> operator[](const valarray<size_t>&);

private:

  // Compilers may take advantage of inlining, constant propagation, loop fusion, tracking of pointers obtained from operator new, and other techniques to generate efficient valarrays.

};
```

281) Compilers may take advantage of inlining, constant propagation, loop fusion, tracking of pointers obtained from `operator new`, and other techniques to generate efficient valarrays.
Each of these operations returns a subset of the array. The `const`-qualified versions return this subset as a new `valarray`. The non-`const` versions return a class template object which has reference semantics to the original array.

### 26.5.2.5 `valarray` unary operators

```cpp
valarray<T> operator+() const;
valarray<T> operator-() const;
valarray<T> operator~() const;
valarray<bool> operator!() const;
```

Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `T` (`bool` for `operator!`) or which may be unambiguously converted to type `T` (`bool` for `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.

### 26.5.2.6 `valarray` computed assignment

```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 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.
valarray\langle T\rangle \& \text{ operator}\^\langle = \text{ (const } T\&); 
\text{valarray}\langle T\rangle \& \text{ operator}\&\langle = \text{ (const } T\&); 
\text{valarray}\langle T\rangle \& \text{ operator}|= \text{ (const } T\&); 
\text{valarray}\langle T\rangle \& \text{ operator}\langle\langle = \text{ (const } T\&); 
\text{valarray}\langle T\rangle \& \text{ operator}\rangle\rangle = \text{ (const } T\&); 

Each of these operators may only be instantiated for a type } T \text{ 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 not invalidate references or pointers to the elements of the array.

26.5.2.7 \text{valarray member functions} \quad [\text{valarray.memberr}]

\text{void swap(valarray}\&\& v); 

\text{Effects: } *\text{this} \text{ obtains the value of } v. v \text{ obtains the value of } *\text{this}.

\text{Complexity: Constant.}

\text{Throws: Nothing.}

\text{size_t size()} \text{ const;}

This function returns the number of elements in the array.

\text{T sum()} \text{ const;}

This function may only be instantiated for a type } T \text{ to which } \text{operator}+= \text{ 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, \text{sum()} \text{ returns the value of element 0. Otherwise, the returned value is calculated by applying } \text{operator}+= \text{ to a copy of an element of the array and all other elements of the array in an unspecified order.}

\text{T min()} \text{ const;}

This function returns the minimum value contained in \text{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 \text{operator}<.

\text{T max()} \text{ const;}

This function returns the maximum value contained in \text{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 \text{operator}<.

\text{valarray}\langle T\rangle \text{ shift(int } n) \text{ const;}

Draft
This function returns an object of class `valarray<T>` of length `size()`, each of whose elements \( I \) is \( (*\text{this})[I + n] \) 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 constructed using the default constructor; the third element of the result will be assigned the value of the first element of the argument; etc. — end example

```cpp
valarray<T> cshift(int n) const;
```

This function returns an object of class `valarray<T>`, of length `size()`, each of whose elements \( I \) is \( (*\text{this})[(I + n) \% size()] \). Thus, if element zero is taken as the leftmost element, a positive value of \( n \) shifts the elements circularly left \( n \) places.

```cpp
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.

```cpp
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.

### 26.5.3 `valarray` non-member operations

#### 26.5.3.1 `valarray` binary operators

```cpp
template<class T> valarray<T> operator* (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator/ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator% (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator+ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator- (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator^ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator& (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator| (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator<< (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator>> (const valarray<T>&, const valarray<T>&);
```
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 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.

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 T&);
template<class T> valarray<T> operator/ (const 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 T&);
template<class T> valarray<T> operator+ (const 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 T&);
template<class T> valarray<T> operator^ (const 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 T&);
template<class T> valarray<T> operator| (const 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 T&);
template<class T> valarray<T> operator>>(const T&, const valarray<T>&);

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 converted to type \( T \).

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.

26.5.3.2 valarray logical operators

template<class T> valarray<bool> operator==
 (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator!=
 (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator<
 (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator>
 (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator<=
 (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator=>
 (const valarray<T>&, const valarray<T>&);
(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>&);

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 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.

template<class T> valarray<bool> operator==(const valarray<T>&, const T&);

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 converted to type bool.

Each of these operators returns a bool 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 and the non-array argument.

26.5.3.3 valarray transcendentalists

Draft
(const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> atan2(const valarray<T>&, const valarray<T>&);

(inline) template<class T> valarray<T> atan2(const valarray<T>&, const T&);

(inline) template<class T> valarray<T> atan2(const T&, const valarray<T>&);

(inline) template<class T> valarray<T> cos(const valarray<T>&);

(inline) template<class T> valarray<T> cosh(const valarray<T>&);

(inline) template<class T> valarray<T> exp(const valarray<T>&);

(inline) template<class T> valarray<T> log(const valarray<T>&);

(inline) template<class T> valarray<T> log10(const valarray<T>&);

(inline) template<class T> valarray<T> pow(const valarray<T>&, const valarray<T>&);

(inline) template<class T> valarray<T> pow(const valarray<T>&, const T&);

(inline) template<class T> valarray<T> pow(const T&, const valarray<T>&);

(inline) template<class T> valarray<T> sin(const valarray<T>&);

(inline) template<class T> valarray<T> sinh(const valarray<T>&);

(inline) template<class T> valarray<T> sqrt(const valarray<T>&);

(inline) template<class T> valarray<T> tan(const valarray<T>&);

(inline) 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 converted to type T.

26.5.3.4 valarray specialized algorithms

[valarray.special]

template <class T> void swap(valarray<T>& x, valarray<T>& y);

template <class T> void swap(valarray<T>&& x, valarray<T>& y);

template <class T> void swap(valarray<T>& x, valarray<T>&& y);

1 Effects: x.swap(y).

26.5.4 Class slice

[class.slice]

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
26.5 Numeric arrays

26.5.4.1 slice constructors

slice();
slice(size_t start, size_t length, size_t stride);
slice(const slice&);

The default constructor is equivalent to slice(0, 0, 0). A default constructor is provided only to permit the definition of arrays of slices. The constructor with arguments for a slice takes a start, length, and stride parameter.

Example: slice(3, 8, 2) constructs a slice which selects elements 3, 5, 7, ... 17 from an array. — end example

26.5.4.2 slice access functions

size_t start() const;
size_t size() const;
size_t stride() const;

These functions return the start, length, or stride specified by a slice object.

26.5.5 Class template slice_array

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();
        slice_array& operator=(const slice_array&);
        void operator=(const T&) const;

    private:

282) BLAS stands for Basic Linear Algebra Subprograms. C++ programs may instantiate this class. See, for example, Dongarra, Du Croz, Duff, and Hammarling: A set of Level 3 Basic Linear Algebra Subprograms; Technical Report MCS-P1-0888, Argonne National Laboratory (USA), Mathematics and Computer Science Division, August, 1988.
The `slice_array` template is a helper template used by the `slice` subscript operator

```cpp
slice_array<T> valarray<T>::operator[](slice);
```

It has reference semantics to a subset of an array specified by a `slice` object.

**Example:** The expression `a[slice(1, 5, 3)] = b;` has the effect of assigning the elements of `b` to a slice of the elements in `a`. For the slice shown, the elements selected from `a` are 1, 4, ..., 13. — *end example*]

### 26.5.5.1 slice_array constructors

This constructor is declared to be private. This constructor need not be defined.

### 26.5.5.2 slice_array assignment

These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which the `slice_array` object refers.

### 26.5.5.3 slice_array computed assignment

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 `slice_array` object refers.

### 26.5.5.4 slice_array fill function

This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `slice_array` object refers.
26.5 Numeric arrays

26.5.6 The gslice class

namespace std {
    class gslice {
        public:
            gslice();
            gslice(size_t s, const valarray<size_t>& l, const valarray<size_t>& d);

            size_t start() const;
            valarray<size_t> size() const;
            valarray<size_t> stride() const;
    };
}

This class represents a generalized slice out of an array. A gslice is defined by a starting offset ($s$), a set of lengths ($l_j$), and a set of strides ($d_j$). The number of lengths shall equal the number of strides.

A gslice represents a mapping from a set of indices ($i_j$), equal in number to the number of strides, to a single index $k$. It is useful for building multidimensional array classes using the valarray template, which is one-dimensional. The set of one-dimensional index values specified by a gslice are

\[ k = s + \sum_j i_j d_j \]

where the multidimensional indices $i_j$ range in value from 0 to $l_{ij} - 1$.

[Example: The gslice specification]

\[
\begin{align*}
\text{start} &= 3 \\
\text{length} &= \{2, 4, 3\} \\
\text{stride} &= \{19, 4, 1\}
\end{align*}
\]

yields the sequence of one-dimensional indices

\[ k = 3 + (0, 1) \times 19 + (0, 1, 2, 3) \times 4 + (0, 1, 2) \times 1 \]

which are ordered as shown in the following table:

\[
\begin{align*}
(i_0, \ i_1, \ i_2, \ k) &= \\
(0, \ 0, \ 0, \ 3), \\
(0, \ 0, \ 1, \ 4), \\
(0, \ 0, \ 2, \ 5), \\
(0, \ 1, \ 0, \ 7), \\
(0, \ 1, \ 1, \ 8), \\
(0, \ 1, \ 2, \ 9), \\
(0, \ 2, \ 0, \ 11), \\
(0, \ 2, \ 1, \ 12), \\
(0, \ 2, \ 2, \ 13), \\
(0, \ 3, \ 0, \ 15), \\
(0, \ 3, \ 1, \ 16),
\end{align*}
\]
(0, 3, 2, 17),
(1, 0, 0, 22),
(1, 0, 1, 23),
...
(1, 3, 2, 36)

That is, the highest-ordered index turns fastest. — end example ]

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

(0, 0, 0, 3),
(0, 0, 1, 4),
(0, 0, 2, 5),
(0, 1, 0, 4),
(0, 1, 1, 5),
(0, 1, 2, 6),
...

— 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.

26.5.6.1 gslice constructors

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.

26.5.6.2 gslice access functions

size_t start() const;
valarray<size_t> size() const;
valarray<size_t> stride() const;

These access functions return the representation of the start, lengths, or strides specified for the gslice.

26.5.7 Class template gslice_array

Draft
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();
            gslice_array& operator=(const gslice_array&);
            void operator=(const T&) const;

            private:
                gslice_array() = delete;  // as implied by declaring copy constructor above
        };
    }

This template is a helper template used by the slice subscript operator

    gslice_array<T> valarray<T>::operator[](const gslice&);

It has reference semantics to a subset of an array specified by a gslice object.

Thus, the expression a[gslice(1, length, stride)] = b has the effect of assigning the elements of b to a
generalized slice of the elements in a.

### 26.5.7.1 gslice_array constructors

[gslice.array.cons]

    gslice_array();

This constructor is declared to be private. This constructor need not be defined.

### 26.5.7.2 gslice_array assignment

[gslice.array.assign]

    void operator=(const valarray<T>&) const;
    gslice_array& operator=(const gslice_array&);

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.
26.5.7.3 gslice_array

```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 valarray<T> object to which the gslice_array object refers.

26.5.7.4 gslice_array fill function

```cpp
void operator=(const T&) const;
```

This function has reference semantics, assigning the value of its argument to the elements of the valarray<T> object to which the gslice_array object refers.

26.5.8 Class template mask_array

```cpp
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;

        mask_array(const mask_array&);
    ~mask_array();
        mask_array& operator=(const mask_array&);
        void operator=(const T&) const;

    private:
        mask_array() __delete;  // as implied by declaring copy constructor above
    };
```

Draft
This template is a helper template used by the mask subscript operator:

\[
\text{mask\_array}\langle T\rangle \text{ valarray}\langle T\rangle::\text{operator}\[\text{\} (\text{const valarray<bool}\&).}
\]

It has reference semantics to a subset of an array specified by a boolean mask. Thus, the expression \(a[mask] = b\); has the effect of assigning the elements of \(b\) to the masked elements in \(a\) (those for which the corresponding element in mask is true.)

26.5.8.1 mask_array constructors

\[\text{mask\_array()}.\]

This constructor is declared to be private. This constructor need not be defined.

26.5.8.2 mask_array assignment

\[
\text{void operator=} (\text{const valarray}\langle T\rangle\&)\ \text{const};
\]

\[
\text{mask\_array\& operator=} (\text{const mask\_array}\&);\]

These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the \(\text{valarray}\langle T\rangle\) object to which it refers.

26.5.8.3 mask_array computed assignment

\[
\text{void operator\*= (const valarray}\langle T\rangle\&)\ \text{const};
\]

\[
\text{void operator/= (const valarray}\langle T\rangle\&)\ \text{const};
\]

\[
\text{void operator\%= (const valarray}\langle T\rangle\&)\ \text{const};
\]

\[
\text{void operator\+= (const valarray}\langle T\rangle\&)\ \text{const};
\]

\[
\text{void operator\-= (const valarray}\langle T\rangle\&)\ \text{const};
\]

\[
\text{void operator^=} (\text{const valarray}\langle T\rangle\&)\ \text{const};
\]

\[
\text{void operator\&=} (\text{const valarray}\langle T\rangle\&)\ \text{const};
\]

\[
\text{void operator|=} (\text{const valarray}\langle T\rangle\&)\ \text{const};
\]

\[
\text{void operator<<=} (\text{const valarray}\langle T\rangle\&)\ \text{const};
\]

\[
\text{void operator>>=} (\text{const valarray}\langle T\rangle\&)\ \text{const};
\]

These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the \(\text{valarray}\langle T\rangle\) object to which the mask object refers.

26.5.8.4 mask_array fill function

\[
\text{void operator=} (\text{const T}\&)\ \text{const};
\]

This function has reference semantics, assigning the value of its argument to the elements of the \(\text{valarray}\langle T\rangle\) object to which the mask\_array object refers.
26.5.9 Class template indirect_array

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();
        indirect_array& operator=(const indirect_array&);
        void operator=(const T&) const;
    private:
        indirect_array() = delete;  // as implied by declaring copy constructor above
    }
};

This template is a helper template used by the indirect subscript operator
indirect_array<T> valarray<T>::operator[](const valarray<size_t>&).

It has reference semantics to a subset of an array specified by an indirect_array. Thus the expression a[ indirect] = b; has the effect of assigning the elements of b to the elements in a whose indices appear in indirect.

26.5.9.1 indirect_array constructors

indirect_array();

This constructor is declared to be private. This constructor need not be defined.

26.5.9.2 indirect_array assignment

void operator=(const valarray<T>&) const;
indirect_array& operator=(const indirect_array&);

These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the valarray<T> object to which it refers.
If the indirect_array specifies an element in the valarray<T> object to which it refers more than once, the behavior is undefined.

Example:
```cpp
type 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

26.5.9.3 indirect_array computed assignment

```cpp
void operator==(const valarray<T>&) const;
void operator!=(const valarray<T>&) const;
void operator<==(const valarray<T>&) const;
void operator<>(const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>(=const valarray<T>&) const;
void operator^==(const valarray<T>&) const;
void operator|^=(const valarray<T>&) const;
void operator&==(const valarray<T>&) const;
void operator&^=(const valarray<T>&) const;
```

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.

If the indirect_array specifies an element in the valarray<T> object to which it refers more than once, the behavior is undefined.

26.5.9.4 indirect_array fill function

```cpp
void operator=(const T&) const;
```

This function has reference semantics, assigning the value of its argument to the elements of the valarray<T> object to which the indirect_array object refers.

26.6 Generalized numeric operations

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);
}
```
Draft
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);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator adjacent_difference(InputIterator first,
InputIterator last,
OutputIterator result,
BinaryOperation binary_op);

The requirements on the types of algorithms’ arguments that are described in the introduction to clause 25 also apply to the following algorithms.

26.6.1 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.\(^{283}\)

\(^{283}\) 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.

Draft
26.6 Generalized numeric operations

26.6.2 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);
```

1 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 + (last - first)) in order.

2 Requires: T shall meet the requirements of CopyConstructible (34) and Assignable (23.1) types. In the ranges [first1, last1] and [first2, first2 + (last - first)] binary_op1 and binary_op2 shall neither modify elements nor invalidate iterators or subranges.

26.6.3 Partial sum

```
template <class InputIterator, class OutputIterator>
OutputIterator
partial_sum(InputIterator first, InputIterator last,
            OutputIterator result);
```

```
template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator
partial_sum(InputIterator first, InputIterator last,
            OutputIterator result, BinaryOperation binary_op);
```

1 Effects: Assigns to every element referred to by iterator i in the range [result, result + (last - first)) a value correspondingly equal to

\[ (\ldots (*first + *(first + 1)) + \ldots) + *(first + (i - result)) \]

or

\[ \text{binary_op(binary_op(\ldots,}
\text{binary_op(*first, *(first + 1)),\ldots), *(first + (i - result)))} \]

284) The use of fully closed ranges is intentional

285) The use of fully closed ranges is intentional
Returns: result + (last - first).

Complexity: Exactly (last - first) - 1 applications of binary_op.

Requires: In the ranges [first, last] and [result, result + (last - first)] binary_op shall neither modify elements nor invalidate iterators or subranges.\(^{286}\)

Remarks: result may be equal to first.

### 26.6.4 Adjacent difference

```cpp
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: Assigns to every element referred to by iterator i in the range [result + 1, result + (last - first)) a value correspondingly equal to

\[ *(first + (i - result)) - *(first + (i - result) - 1) \]

or

\[ \text{binary}_\text{op}(*\text{first + (i - result))}, *\text{first + (i - result) - 1}). \]

result gets the value of *first.

Requires: In the ranges [first, last] and [result, result + (last - first)], binary_op shall neither modify elements nor invalidate iterators or subranges.\(^{287}\)

Remarks: result may be equal to first.

Returns: result + (last - first).

Complexity: Exactly (last - first) - 1 applications of binary_op.

### 26.7 C Library

The header `<tgmath>` simply includes the headers `<complex.h>` and `<math.h>`.

[Note: The overloads provided in C99 by magic macros are already provided in `<complex.h>` and `<math.h>` by "sufficient" additional overloads. — end note]

The header `<tgmath.h>` effectively includes the headers `<complex.h>` and `<math.h>`.

\(^286\) The use of fully closed ranges is intentional.

\(^287\) The use of fully closed ranges is intentional.
Tables 108 and 109 describe headers `<cmath>` and `<cstdlib>`, respectively.

Table 108: Header `<cmath>` synopsis

<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_NAN</td>
</tr>
<tr>
<td>FP_ILOGBO</td>
<td>FP_NORMAL</td>
</tr>
<tr>
<td>Types:</td>
<td>double_t</td>
</tr>
<tr>
<td>Functions:</td>
<td>float_t</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>exp</td>
</tr>
<tr>
<td>asinh</td>
<td>exp2</td>
</tr>
<tr>
<td>atan</td>
<td>expm1</td>
</tr>
<tr>
<td>atanh</td>
<td>fabs</td>
</tr>
<tr>
<td>atan2</td>
<td>fdim</td>
</tr>
<tr>
<td>ceil</td>
<td>floor</td>
</tr>
<tr>
<td>copysign</td>
<td>fmax</td>
</tr>
<tr>
<td>cos</td>
<td>fmin</td>
</tr>
<tr>
<td>Templates:</td>
<td></td>
</tr>
<tr>
<td>fpclassify</td>
<td>isgreatequal</td>
</tr>
<tr>
<td>isfinite</td>
<td>isinf</td>
</tr>
<tr>
<td>isgreater</td>
<td>isless</td>
</tr>
<tr>
<td>islessequal</td>
<td>islessgreater</td>
</tr>
</tbody>
</table>

The contents of these headers are the same as the Standard C library headers `<math.h>` and `<stdlib.h>` respectively, with the following changes:

1. The `rand` function has the semantics specified in the C standard, except that the implementation may specify that particular library functions may call `rand`.
2. 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.
3. The added signatures are:
   ```
   long abs(long); //labs()
   long long abs(long long); //llabs()
   ldiv_t div(long, long); //ldiv()
   l1div_t div(long long, long long); //l1div()
   ```
4. 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.

*Draft*
### Table 109: 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</td>
</tr>
<tr>
<td>div</td>
<td>llabs</td>
</tr>
<tr>
<td>labs</td>
<td>lldiv</td>
</tr>
</tbody>
</table>

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);
```
26.7 C Library Numerics library 904

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 pow(float, int);
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()
double pow(double, int);

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 long llrint(long double);
long long 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 llrint(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 pow(long double, int);
long double remainder(long double, long double);
long double remquo(long double, long double, int *);
long double rint(long double);
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);

Moreover, there shall be additional overloads sufficient to ensure:

1. If any argument corresponding to a double parameter has type long double, then all arguments corresponding to double parameters are effectively cast to long double.

2. Otherwise, if any argument corresponding to a double parameter has type double or an integer type, then all arguments corresponding to double parameters are effectively cast to double.

3. Otherwise, all arguments corresponding to double parameters are effectively cast to float.

The templates defined in <cmath> replace the C99 macros with the same names. The templates have the following declarations:

```cpp
namespace std {
    template <class T> bool signbit(T x);
}
```
template <class T> int fpclassify(T x);

template <class T> bool isfinite(T x);
template <class T> bool isinf(T x);
template <class T> bool isnan(T x);
template <class T> bool isnormal(T x);

template <class T> bool isgreater(T x, T y);
template <class T> bool isgreaterequal(T x, T y);
template <class T> bool islessequal(T x, T y);
template <class T> bool islessgreater(T x, T y);
template <class T> bool isunordered(T x, T y);
} // namespace std

The templates behave the same as the C99 macros with corresponding names defined in C99 subclause 7.12.3, Classification macros, and C99 subclause 7.12.14, Comparison macros.

SEE ALSO: ISO C subclauses 7.5, 7.10.2, 7.10.6.
Chapter 27  Input/output library  [input.output]

1. This clause describes components that C++ programs may use to perform input/output operations.

2. The following subclauses describe requirements for stream parameters, and components for forward declarations of iostreams, predefined iostreams objects, base iostreams classes, stream buffering, stream formatting and manipulators, string streams, and file streams, as summarized in Table 110.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.1 Iostreams requirements</td>
<td></td>
</tr>
<tr>
<td>27.1.1 Imbue Limitations</td>
<td>[iostreams.requirements]</td>
</tr>
<tr>
<td>27.2 Forward declarations</td>
<td>&lt;iosfwd&gt;</td>
</tr>
<tr>
<td>27.3 Standard iostream objects</td>
<td>&lt;iostream&gt;</td>
</tr>
<tr>
<td>27.4 Iostreams base classes</td>
<td>&lt;ios&gt;</td>
</tr>
<tr>
<td>27.5 Stream buffers</td>
<td>&lt;streambuf&gt;</td>
</tr>
<tr>
<td>27.6 Formatting and manipulators</td>
<td>&lt;istream&gt;</td>
</tr>
<tr>
<td>27.7 String streams</td>
<td>&lt;sstream&gt;</td>
</tr>
<tr>
<td>27.8 File streams</td>
<td>&lt;fstream&gt;</td>
</tr>
</tbody>
</table>

27.1 Iostreams requirements [iostreams.requirements]

27.1.1 Imbue Limitations [iostream.limits.imbue]

1. No function described in clause 27 except for ios_base::imbue 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.

27.1.2 Positioning Type Limitations [iostreams.limits.pos]

1. 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.

27.2 Forward declarations

Header <iosfwd> synopsis

namespace std {
  template<class charT> class char_traits;
  template<> class char_traits<char>;
  template<> class char_traits<wchar_t>;

  template<class T> class allocator;

  template <class charT, class traits = char_traits<charT> >
    class basic_ios;

  template <class charT, class traits = char_traits<charT> >
    class basic_streambuf;

  template <class charT, class traits = char_traits<charT> >
    class basic_istream;

  template <class charT, class traits = char_traits<charT> >
    class basic_ostream;

  template <class charT, class traits = char_traits<charT> >
    class basic_iostream;

  template <class charT, class traits = char_traits<charT>,
            class Allocator = allocator<charT> >
    class basic_stringbuf;

  template <class charT, class traits = char_traits<charT>,
            class Allocator = allocator<charT> >
    class basic_istringstream;

  template <class charT, class traits = char_traits<charT>,
            class Allocator = allocator<charT> >
    class basic_ostringstream;

  template <class charT, class traits = char_traits<charT>,
            class Allocator = allocator<charT> >
    class basic_stringstream;

  template <class charT, class traits = char_traits<charT> >
    class basic_filebuf;

  template <class charT, class traits = char_traits<charT> >
    class basic_ifstream;

Draft
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_ostream<char> ostream;
  typedef basic_iostream<char> iostream;

typedef basic_stringbuf<char> stringbuf;
  typedef basic_istream<wchar_t> wistream;
  typedef basic_ostream<wchar_t> wostream;
  typedef basic_iostream<wchar_t> wiostream;

typedef basic_stringbuf<wchar_t> wstringbuf;
  typedef basic_istream<wchar_t> wistream;
  typedef basic_ostream<wchar_t> wostream;
  typedef basic_iostream<wchar_t> wiostream;

typedef basic_stringbuf<wchar_t> wstringbuf;
  typedef basic_istream<wchar_t> wistream;
  typedef basic_ostream<wchar_t> wostream;
  typedef basic_iostream<wchar_t> wiostream;

  template <class state> class fpos;
  typedef fpos<char_traits<char>::state_type> streampos;
  typedef fpos<char_traits<wchar_t>::state_type> wstreampos;

}
27.3 Standard iostream objects

[iostream.objects]  Input/output library

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.289)

[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>`.

The class template specialization `basic_streambuf<charT,traits>` serves as a base class for template classes `basic_stringbuf` and `basic_filebuf`.

The class template specialization `basic_istream<charT,traits>` serves as a base class for template classes `basic_istringstream` and `basic_ifstream`.

The class template specialization `basic_ostream<charT,traits>` serves as a base class for template classes `basic_ostringstream` and `basic_ofstream`.

The class template specialization `basic_iostream<charT,traits>` serves as a base class for template classes `basicstringstream` and `basic_fstream`.

Other typedefs define instances of class templates specialized for `char` or `wchar_t` types.

Specializations of the class template `fpos` are used for specifying file position information.

The types `streampos` and `wstreampos` are used for positioning streams specialized on `char` and `wchar_t` respectively.

This synopsis suggests a circularity between `streampos` and `char_traits<char>`. An implementation can avoid this circularity by substituting equivalent types. One way to do this might be

```cpp
template<class stateT> class fpos { ... }; // depends on nothing
typedef ... _STATE; // implementation private declaration of stateT
typedef fpos<_STATE> streampos;

template<> struct char_traits<char> {
    typedef streampos pos_type;
    // ...
};
```

— end note ]

27.3 Standard iostream objects

Header `<iostream>` synopsis

```cpp
#include <ios>
#include <iostream>
#include <istream>
#include <ostream>

namespace std {
    extern istream cin;
}
```

289) 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.
The header `<iostream>` declares objects that associate objects with the standard C streams provided for by the functions declared in `<cstdio>` (27.8.2), and includes all the headers necessary to use these objects.

The objects are constructed and the associations are established at some time prior to or during the first time an object of class `ios_base::Init` is constructed, and in any case before the body of `main` begins execution. If a translation unit includes `<iostream>` or explicitly constructs an `ios_base::Init` object, these stream objects shall be constructed before dynamic initialization of non-local objects defined later in that translation unit, and these stream objects shall be destroyed after the destruction of dynamically initialized non-local objects defined later in that translation unit.

Mixing operations on corresponding wide- and narrow-character streams follows the same semantics as mixing such operations on FILEs, as specified in Amendment 1 of the ISO C standard.

### 27.3.1 Narrow stream objects

The object `cin` controls input from a stream buffer associated with the object `stdin`, declared in `<cstdio>`.

After the object `cin` is initialized, `cin.tie()` returns `&cout`. Its state is otherwise the same as required for `basic_ios<char>::init` (27.4.4.1).

The object `cout` controls output to a stream buffer associated with the object `stdout`, declared in `<cstdio>` (27.8.2).

The object `cerr` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (27.8.2).

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` (27.4.4.1).

The object `clog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (27.8.2).

---

290) If it is possible for them to do so, implementations are encouraged to initialize the objects earlier than required.

291) Constructors and destructors for static objects can access these objects to read input from `stdin` or write output to `stdout` or `stderr`.
27.3.2 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 (27.4.4.1).

wostream wcout;

3. The object wcout controls output to a stream buffer associated with the object stdout, declared in <cstdio> (27.8.2).

wostream wcerr;

4. The object wcerr controls output to a stream buffer associated with the object stderr, declared in <cstdio> (27.8.2).
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 (27.4.4.1).

wostream wclog;

6. The object wclog controls output to a stream buffer associated with the object stderr, declared in <cstdio> (27.8.2).

27.4 Iostreams base classes

Header <ios> synopsis

#include <iosfwd>

namespace std {
    typedef OFF_T streamoff;
    typedef SZ_T streamsize;
    template <class stateT> class fpos;

    class ios_base;
    template <class charT, class traits = char_traits<charT> >
    class basic_ios;

    // 27.4.5, 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);

    Draft
typedef OFF_T streamoff;

The type streamoff is an implementation-defined type that satisfies the requirements of 27.4.3.2.

typedef SZ_T streamsize;

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.²⁹²

²⁹²streamsize 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).
class failure;

typedef T1 fmtflags;
static const fmtflags boolalpha;
static const fmtflags dec;
static const fmtflags fixed;
static const fmtflags hex;
static const fmtflags internal;
static const fmtflags left;
static const fmtflags oct;
static const fmtflags right;
static const fmtflags scientific;
static const fmtflags showbase;
static const fmtflags showpoint;
static const fmtflags showpos;
static const fmtflags skipws;
static const fmtflags unitbuf;
static const fmtflags uppercase;
static const fmtflags adjustfield;
static const fmtflags basefield;
static const fmtflags floatfield;

typedef T2 iostate;
static const iostate badbit;
static const iostate eofbit;
static const iostate failbit;
static const iostate goodbit;

typedef T3 openmode;
static const openmode app;
static const openmode ate;
static const openmode binary;
static const openmode in;
static const openmode out;
static const openmode trunc;

typedef T4 seekdir;
static const seekdir beg;
static const seekdir cur;
static const seekdir end;

class Init;

// 27.4.2.2 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);

// 27.4.2.3 locales:
locale imbue(const locale& loc);
locale getloc() const;

// 27.4.2.5 storage:
static int xalloc();
long& iword(int index);
void*& pword(int index);

// destructor
virtual ~ios_base();

// 27.4.2.6 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);

private:
ios_base(const ios_base&) = delete;
iose_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:
   — a class failure derived from exception;
   — a class Init;
   — three bitmask types, fmtflags, iostate, and openmode;
   — an enumerated type, seekdir.

2. It maintains several kinds of data:
   — state information that reflects the integrity of the stream buffer;
— control information that influences how to interpret (format) input sequences and how to generate (format) output sequences;
— additional information that is stored by the program for its private use.

[Note: For the sake of exposition, the maintained data is presented here as:

— static int index, specifies the next available unique index for the integer or pointer arrays maintained for the private use of the program, initialized to an unspecified value;
— long* iarray, points to the first element of an arbitrary-length long array maintained for the private use of the program;
— void** parray, points to the first element of an arbitrary-length pointer array maintained for the private use of the program. — end note]

27.4.2.1 Types [ios.types]

27.4.2.1.1 Class ios_base::failure [ios::failure]

namespace std {
    class ios_base::failure : public exception {
    public:
        explicit failure(const string& msg);
        explicit failure(const char* msg);
        virtual const char* what() const throw();
    }
}

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.

explicit failure(const string& msg);
Effects: Constructs an object of class failure.

Postcondition: strcmp(what(), msg.c_str()) == 0

explicit failure(const char* msg);

Effects: Constructs an object of class failure.

Postcondition: strcmp(what(), msg) == 0

const char* what() const;

Returns: The message msg with which the exception was created.

27.4.2.1.2 Type ios_base::fmtflags [ios::fmtflags]

typedef T1 fmtflags;

Draft
Table 111: fmtflags effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Effect(s) if set</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolalpha</td>
<td>insert and extract bool type in alphabetic format</td>
</tr>
<tr>
<td>dec</td>
<td>converts integer input or generates integer output in decimal base</td>
</tr>
<tr>
<td>fixed</td>
<td>generate floating-point output in fixed-point notation</td>
</tr>
<tr>
<td>hex</td>
<td>converts integer input or generates integer output in hexadecimal base</td>
</tr>
<tr>
<td>internal</td>
<td>adds fill characters at a designated internal point in certain generated output, or identical to right if no such point is designated</td>
</tr>
<tr>
<td>left</td>
<td>adds fill characters on the right (final positions) of certain generated output</td>
</tr>
<tr>
<td>oct</td>
<td>converts integer input or generates integer output in octal base</td>
</tr>
<tr>
<td>right</td>
<td>adds fill characters on the left (initial positions) of certain generated output</td>
</tr>
<tr>
<td>scientific</td>
<td>generates floating-point output in scientific notation</td>
</tr>
<tr>
<td>showbase</td>
<td>generates a prefix indicating the numeric base of generated integer output</td>
</tr>
<tr>
<td>showpoint</td>
<td>generates a decimal-point character unconditionally in generated floating-point output</td>
</tr>
<tr>
<td>showpos</td>
<td>generates a + sign in non-negative generated numeric output</td>
</tr>
<tr>
<td>skips</td>
<td>skips leading whitespace before certain input operations</td>
</tr>
<tr>
<td>unitbuf</td>
<td>flushes output after each output operation</td>
</tr>
<tr>
<td>uppercase</td>
<td>replaces certain lowercase letters with their uppercase equivalents in generated output</td>
</tr>
</tbody>
</table>

1 The type `fmtflags` is a bitmask type (17.3.2.1.2). Setting its elements has the effects indicated in Table 111.
2 Type `fmtflags` also defines the constants indicated in Table 112.

Table 112: 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>

27.4.2.1.3 **Type** `ios_base::iostate` [ios::iostate]

```cpp
typedef T2 iostate;
```

1 The type `iostate` is a bitmask type (17.3.2.1.2) that contains the elements indicated in Table 113.
2 Type `iostate` also defines the constant:
   — `goodbit`, the value zero.

27.4.2.1.4 **Type** `ios_base::openmode` [ios::openmode]

```cpp
typedef T3 openmode;
```

Draft
Table 113: 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>

The type openmode is a bitmask type (17.3.2.1.2). It contains the elements indicated in Table 114.

Table 114: 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>

27.4.2.1.5 Type ios_base::seekdir

typedef T4 seekdir;

The type seekdir is an enumerated type (17.3.2.1.1) that contains the elements indicated in Table 115.

Table 115: seekdir effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>beg</td>
<td>request a seek (for subsequent input or output) relative to the beginning of the stream</td>
</tr>
<tr>
<td>cur</td>
<td>request a seek relative to the current position within the sequence</td>
</tr>
<tr>
<td>end</td>
<td>request a seek relative to the current end of the sequence</td>
</tr>
</tbody>
</table>

27.4.2.6 Class ios_base::Init

namespace std {
   class ios_base::Init {
   public:
    Init();
    ~Init();
   private:
    // static int init_cnt;
   };
}

Draft
The class \texttt{Init} describes an object whose construction ensures the construction of the eight objects declared in \texttt{<iostream>} (27.3) that associate file stream buffers with the standard C streams provided for by the functions declared in \texttt{<cstdio>} (27.8.2).

For the sake of exposition, the maintained data is presented here as:

\begin{itemize}
  \item \texttt{static int \_init\_cnt}, counts the number of constructor and destructor calls for class \texttt{Init}, initialized to zero.
\end{itemize}

\begin{verbatim}
\texttt{Init();}
\end{verbatim}

\textbf{Effects:} Constructs an object of class \texttt{Init}. If \texttt{\_init\_cnt} is zero, the function stores the value one in \texttt{\_init\_cnt}, then constructs and initializes the objects \texttt{cin, cout, cerr, clog (27.3.1)}, \texttt{wcin, wcout, wcerr, and wclog (27.3.2)}. In any case, the function then adds one to the value stored in \texttt{\_init\_cnt}.

\begin{verbatim}
\texttt{\_Init();}
\end{verbatim}

\textbf{Effects:} Destroys an object of class \texttt{Init}. The function subtracts one from the value stored in \texttt{\_init\_cnt} and, if the resulting stored value is one, calls \texttt{cout.flush(), cerr.flush(), clog.flush(), wcout.flush(), wcerr.flush(), wclog.flush()}.

\subsection*{27.4.2.2 \texttt{ios\_base} state functions} [fmtflags.state]

\texttt{fmtflags flags() const;}

\textit{Returns:} The format control information for both input and output.

\begin{verbatim}
\texttt{fmtflags flags(fmtflags fmtfl);}\end{verbatim}

\textit{Postcondition: fmtfl \texttt{== flags().}}

\textit{Returns:} The previous value of \texttt{flags().}

\begin{verbatim}
\texttt{fmtflags setf(fmtflags fmtfl);}\end{verbatim}

\textit{Effects:} Sets \texttt{fmtfl} in \texttt{flags().}

\textit{Returns:} The previous value of \texttt{flags().}

\begin{verbatim}
\texttt{fmtflags setf(fmtflags fmtfl, fmtflags mask);}\end{verbatim}

\textit{Effects:} Clears \texttt{mask} in \texttt{flags()}, sets \texttt{fmtfl \& mask} in \texttt{flags()}.\n
\textit{Returns:} The previous value of \texttt{flags().}

\begin{verbatim}
\texttt{void unsetf(fmtflags mask);}\end{verbatim}

\textit{Effects:} Clears \texttt{mask} in \texttt{flags()}.\n
\begin{verbatim}
\texttt{streamsize precision() const;}
\end{verbatim}

\textit{Returns:} The precision to generate on certain output conversions.

Draft
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().

27.4.2.3 ios_base functions

locale imbue(const locale& loc);
   // Effects: Calls each registered callback pair (fn, index) (27.4.2.6) as (*fn)(imbue_event,*this,index) at such a time that a call to ios_base::getloc() from within fn returns the new locale value loc.
   // Returns: The previous value of getloc().
   // Postcondition: loc == getloc().

locale getloc() const;
   // Returns: If no locale has been imbued, a copy of the global C++ locale, locale(), in effect at the time of construction. Otherwise, returns the imbued locale, to be used to perform locale-dependent input and output operations.

27.4.2.4 ios_base static members

bool sync_with_stdio(bool sync = true);
   // Returns: true if the previous state of the standard iostream objects (27.3) was synchronized and otherwise returns false. The first time it is called, the function returns true.
   // Effects: If any input or output operation has occurred using the standard streams prior to the call, the effect is implementation-defined. Otherwise, called with a false argument, it allows the standard streams to operate independently of the standard C streams.

When a standard iostream object str is synchronized with a standard stdio stream f, the effect of inserting a character c by
   fputc(f, c);

is the same as the effect of
   str.rdbuf()->sputc(c);
for any sequences of characters; the effect of extracting a character \( c \) by
\[
c = fgetc(f);
\]
is the same as the effect of
\[
c = str.rdbuf()->sbumpc(c);
\]
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.\(^{293}\)

### 27.4.2.5 `ios_base` storage functions \(^{[ios.base.storage]}\)

```cpp
class ios_base
{
public:
  static int xalloc();

  long& iword(int idx);

  void* & pword(int idx);
};
```

1. **Returns**: `index++`

2. **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.\(^{294}\) 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\(^{295}\) 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`).

3. **Returns**: On success `iarray[idx]`. On failure, a valid `long` initialized to 0.

4. **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\(^{296}\) and `*this` is a base subobject of a `basic_ios<>` object or subobject,

---

\(^{293}\)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.

\(^{294}\)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.

\(^{295}\)For example, because it cannot allocate space.

\(^{296}\)For example, because it cannot allocate space.
the effect is equivalent to calling `basic_ios<>::setstate(badbit)` on the derived object (which may throw failure).

5 Returns: On success `parray[idx]`. On failure a valid `void*` initialized to 0.

6 Remarks: After a subsequent call to `pword(int)` for the same object, the earlier return value may no longer be valid.

### 27.4.2.6 `ios_base` callbacks

```cpp
void register_callback(event_callback fn, int index);
```

1 Effects: Registers the pair `(fn, index)` such that during calls to `imbue()` (27.4.2.3), `copyfmt()`, or `~ios_base()` (27.4.2.7), 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.

2 Requires: The function `fn` shall not throw exceptions.

Remarks: Identical pairs are not merged. A function registered twice will be called twice.

### 27.4.2.7 `ios_base` constructors/destructors

```cpp
ios_base();
```

1 Effects: Each `ios_base` member has an indeterminate value after construction. These members shall be initialized by calling `basic_ios::init`. If an `ios_base` object is destroyed before these initializations have taken place, the behavior is undefined.

```cpp
~ios_base()
```

2 Effects: Destroys an object of class `ios_base`. Calls each registered callback pair `(fn, index)` (27.4.2.6) as `(*fn)(erase_event, *this, index)` at such time that any `ios_base` member function called from within `fn` has well defined results.

### 27.4.3 Class template `fpos`

```cpp
namespace std {
    template <class stateT> class fpos {
        public:
        // 27.4.3.1 Members
        stateT state() const;
        void state(stateT);
        private;
        // stateT st;
        // exposition only
    }
}
```

### 27.4.3.1 `fpos` Members

Draft
void state(stateT s);

\hspace*{5em} \textit{Effects: Assign} s \textit{to st.}
\hspace*{5em}

stateT state() const;

\hspace*{5em} \textit{Returns: Current value of st.}

\subsection*{27.4.3.2 fpos requirements} \hfill [fpos.operations]

Operations specified in Table 116 are permitted. In that table,

- \( P \) refers to an instance of \texttt{fpos},
- \( p \) and \( q \) refer to values of type \( P \),
- \( O \) refers to type \texttt{streamoff},
- \( o \) refers to a value of type \texttt{streamoff},
- \( sz \) refers to a value of type \texttt{streamsize} and
- \( i \) refers to a value of type \texttt{int}.

\begin{table}[h]
\centering
\begin{tabular}{|l|c|l|}
\hline
expression & return type & operational semantics & assertion/note \\hline
\( P(i) \) & \texttt{fpos} & converts from offset & pre/post-condition \\hline
\( P p(i); \) & \texttt{fpos} & converts from offset & \( p == P(i) \) \hspace*{5em}
\hspace*{5em} note: a destructor is assumed. \\hline
\( P p = i; \) & \texttt{fpos} & converts from offset & \( p == P(i) \) \hspace*{5em}
\hspace*{5em} post: \( p == P(i) \) \\hline
\( P o \) & \texttt{fpos} & converts from offset & \( P(O(p)) == p \) \\hline
\( p == q \) & \texttt{bool} & convertible to bool & \( p == q \) \hspace*{5em}
\hspace*{5em} \( == \) is an equivalence relation \\hline
\( p != q \) & \texttt{bool} & convertible to bool & \( !(p == q) \) \\hline
\( q = p + o \) & \texttt{fpos} & + offset & \( q - o == p \) \\hline
\( p += o \) & \texttt{fpos} & + offset & \( q - o == p \) \\hline
\( q = p - o \) & \texttt{fpos} & - offset & \( q + o == p \) \\hline
\( p -= o \) & \texttt{fpos} & - offset & \( q + o == p \) \\hline
\( o = p - q \) & \texttt{fpos} & distance & \( q + o == p \) \\hline
\( \text{streamsize}(o) \) & \texttt{streamsize} & converts & \texttt{streamsize}(O(sz)) == sz \\hline
\( O(sz) \) & \texttt{OFF_T} & converts & \texttt{streamsize}(O(sz)) == sz \\hline
\end{tabular}
\end{table}

\[ \text{Note: Every implementation is required to supply overloaded operators on fpos objects to satisfy the requirements of 27.4.3.2. It is unspecified whether these operators are members of fpos, global operators, or provided in some other way.} \hspace*{5em} \text{- end note} \]

Stream operations that return a value of type \texttt{traits::pos_type} return \( P(O(-1)) \) as an invalid value to signal an
error. If this value is used as an argument to any istream, ostream, or streambuf member that accepts a value of type traits::pos_type then the behavior of that function is undefined.

27.4.4 Class template basic_ios

namespace std {
    template <class charT, class traits = char_traits<charT> >
    class basic_ios : public ios_base {
    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;

        operator unspecified_bool_type() const;
        bool operator!() const;
        iostate rdstate() const;
        void clear(iostate state = goodbit);
        void setstate(iostate state);
        bool good() const;
        bool eof() const;
        bool fail() const;
        bool bad() const;

        iostate exceptions() const;
        void exceptions(iostate except);

        // 27.4.4.1 Constructor/destructor:
        explicit basic_ios(basic_streambuf<charT,traits>* sb);
        virtual ~basic_ios();

        // 27.4.4.2 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 default) const;

    } // basic_ios
} // namespace std

Draft
char_type widen(char c) const;

private:
    basic_ios(const basic_ios& ) = delete;        // not_defined
    basic_ios& operator=(const basic_ios&) = delete;       // not_defined

protected:
    basic_ios();
    void init(basic_streambuf<charT,traits>* sb);
    void move(basic_ios& rhs);
    void swap(basic_ios& rhs);
    void set_rdbuf(basic_streambuf<charT, traits>* sb);

};

27.4.4.1 basic_ios constructors [basic.ios.cons]

explicit basic_ios(basic_streambuf<charT,traits>* sb);

1     Effects: Constructs an object of class basic_ios, assigning initial values to its member objects by calling init(sb).

basic_ios();

2     Effects: Constructs an object of class basic_ios (27.4.2.7) leaving its member objects uninitialized. The object shall be initialized by calling its init member function. If it is destroyed before it has been initialized the behavior is undefined.

"basic_ios();

3     Remarks: The destructor does not destroy rdbuf().

void init(basic_streambuf<charT,traits>* sb);

    Postconditions: The postconditions of this function are indicated in Table 117.

27.4.4.2 Member functions [basic.ios.members]

basic_ostream<charT,traits>* tie() const;

1     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);

2     Postcondition: tiestr == tie().

3     Returns: The previous value of tie().

basic_streambuf<charT,traits>* rdbuf() const;

4     Returns: A pointer to the streambuf associated with the stream.
27.4 Iostreams base classes

Input/output library 926

Table 117: 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>

basic_streambuf<charT,traits>* rdbuf(basic_streambuf<charT,traits>* sb);

5 Postcondition: sb == rdbuf().
6 Effects: Calls clear().
7 Returns: The previous value of rdbuf().

locale imbue(const locale& loc);
8 Effects: Calls ios_base::imbue(loc) (27.4.2.3) and if rdbuf()=!0 then rdbuf()->pubimbue(loc) (27.5.2.2.1).
9 Returns: The prior value of ios_base::imbue().

char narrow(char_type c, char default) const;
10 Returns: use_facet< ctype<char_type> > (getloc()).narrow(c,default)

char_type widen(char c) const;
11 Returns: use_facet< ctype<char_type> > (getloc()).widen(c)

char_type fill() const;
12 Returns: The character used to pad (fill) an output conversion to the specified field width.

char_type fill(char_type fillch);
13 Postcondition: traits::eq(fillch, fill())
14 Returns: The previous value of fill().

basic_ios& copyfmt(const basic_ios& rhs);
15 Effects: If (this == &rhs) does nothing. Otherwise assigns to the member objects of *this the corresponding member objects of rhs, except that:

— rdstate() and rdbuf() are left unchanged;

Draft
exceptions() is altered last by calling exceptions(rhs.except).

— The contents of arrays pointed at by pword and iword are copied not the pointers themselves.297)

If any newly stored pointer values in *this point at objects stored outside the object rhs, and those objects are destroyed when rhs is destroyed, the newly stored pointer values are altered to point at newly constructed copies of the objects.

Before copying any parts of rhs, calls each registered callback pair (fn, index) as (*fn)(erase_event, *this, index). After all parts but exceptions() have been replaced, calls each callback pair that was copied from rhs as (*fn)(copyfmt_event,*this, index).

Remarks: The second pass permits a copied pword value to be zeroed, or its referent deep copied or reference counted or have other special action taken.

Returns: *this.

void move(basic_ios&& rhs);

Postconditions: *this shall have the state that rhs had before the function call, except that rdbuf () shall return 0. rhs shall be in a valid but unspecified state, except that rhs.rdbuf () shall return the same value as it returned before the function call, and rhs.tie () shall return 0.

void swap(basic_ios&& rhs);

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.

Throws: Nothing.

void set_rdbuf(basic_streambuf<charT, traits>* sb);

Effects: Associates the basic_streambuf object pointed to by sb with this stream without calling clear().

Postconditions: rdbuf () == sb.

Throws: Nothing.

27.4.4.3 basic_ios flags functions

operator \{unspecified-bool-type\}() const;

Returns: If fail () then a value that will evaluate false in a boolean context; otherwise a value that will evaluate true in a boolean context. The value type returned shall not be convertible to int.

[Note: This conversion can be used in contexts where a bool is expected (e.g., an if condition); however, implicit conversions (e.g., to int) that can occur with bool are not allowed, eliminating some sources of user error. One possible implementation choice for this type is pointer-to-member. — end note ]

bool operator!() const;

297) This suggests an infinite amount of copying, but the implementation can keep track of the maximum element of the arrays that is non-zero.
27.4 Iostreams base classes  

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(27.4.2.1.1), constructed with implementation-defined argument values.

void setstate(iostate state);

Effects: Calls clear(rdstate() | state) (which may throw basic_ios::failure(27.4.2.1.1)).

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()).

27.4.5 ios_base manipulators  

27.4.5.1 fmtflags manipulators

ios_base& boolalpha(ios_base& str);

Effects: Calls str.setf(ios_base::boolalpha).

Returns: str.

298) Checking badbit also for fail() is historical practice.
ios_base & noboolalpha(ios_base & str);
3     Effects: Calls str.unsetf(ios_base::boolalpha).
4     Returns: str.

ios_base & showbase(ios_base & str);
5     Effects: Calls str.setf(ios_base::showbase).
6     Returns: str.

ios_base & noshowbase(ios_base & str);
7     Effects: Calls str.unsetf(ios_base::showbase).
8     Returns: str.

ios_base & showpoint(ios_base & str);
9     Effects: Calls str.setf(ios_base::showpoint).
10    Returns: str.

ios_base & noshowpoint(ios_base & str);
11    Effects: Calls str.unsetf(ios_base::showpoint).
12    Returns: str.

ios_base & showpos(ios_base & str);
13    Effects: Calls str.setf(ios_base::showpos).
14    Returns: str.

ios_base & noshowpos(ios_base & str);
15    Effects: Calls str.unsetf(ios_base::showpos).
16    Returns: str.

ios_base & skipws(ios_base & str);
17    Effects: Calls str.setf(ios_base::skipws).
18    Returns: str.

ios_base & nospwsp(ios_base & str);
19    Effects: Calls str.unsetf(ios_base::skipws).
20    Returns: str.

ios_base & uppercase(ios_base & str);
21    Effects: Calls str.setf(ios_base::uppercase).
22    Returns: str.
27.4  Iostreams base classes

ios_base& nouppercase(ios_base& str);
   Effects: Calls str.unsetf(ios_base::uppercase).
   Returns: str.

ios_base& unitbuf(ios_base& str);
   Effects: Calls str.setf(ios_base::unitbuf).
   Returns: str.

ios_base& nounitbuf(ios_base& str);
   Effects: Calls str.unsetf(ios_base::unitbuf).
   Returns: str.

27.4.5.2 adjustfield manipulators

ios_base& internal(ios_base& str);
   Effects: Calls str.setf(ios_base::internal, ios_base::adjustfield).
   Returns: str.

ios_base& left(ios_base& str);
   Effects: Calls str.setf(ios_base::left, ios_base::adjustfield).
   Returns: str.

ios_base& right(ios_base& str);
   Effects: Calls str.setf(ios_base::right, ios_base::adjustfield).
   Returns: str.

27.4.5.3 basefield manipulators

ios_base& dec(ios_base& str);
   Effects: Calls str.setf(ios_base::dec, ios_base::basefield).
   Returns: str.

ios_base& hex(ios_base& str);
   Effects: Calls str.setf(ios_base::hex, ios_base::basefield).
   Returns: str.

[adjustfield.manip]

[basefield.manip]

299) 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.
ios_base& oct(ios_base& str);
Effects: Calls str.setf(ios_base::oct, ios_base::basefield).
Returns: str.

27.4.5.4 floatfield manipulators [floatfield.manip]
ios_base& fixed(ios_base& str);
Effects: Calls str.setf(ios_base::fixed, ios_base::floatfield).
Returns: str.

ios_base& scientific(ios_base& str);
Effects: Calls str.setf(ios_base::scientific, ios_base::floatfield).
Returns: str.

ios_base& hexfloat(ios_base& str);
Effects: Calls str.setf(ios_base::fixed | ios_base::scientific, ios_base::floatfield).
Returns: str.

[Note: The more obvious use of ios_base::hex to specify hexadecimal floating-point format would change the meaning of existing well defined programs. C++2003 gives no meaning to the combination of fixed and scientific. — end note]

ios_base& defaultfloat(ios_base& str);
Effects: Calls str.unsetf(ios_base::floatfield).
Returns: str.

27.5 Stream buffers [streambuffers]

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;
}

The header <streambuf> defines types that control input from and output to character sequences.

27.5.1 Stream buffer requirements [streambuf.reqts]
Stream buffers can impose various constraints on the sequences they control. Some constraints are:

— The controlled input sequence can be not readable.
— The controlled output sequence can be not writable.
— The controlled sequences can be associated with the contents of other representations for character sequences, such as external files.
— The controlled sequences can support operations directly to or from associated sequences.
— The controlled sequences can impose limitations on how the program can read characters from a sequence, write characters to a sequence, put characters back into an input sequence, or alter the stream position.

Each sequence is characterized by three pointers which, if non-null, all point into the same charT array object. The array object represents, at any moment, a (sub)sequence of characters from the sequence. Operations performed on a sequence alter the values stored in these pointers, perform reads and writes directly to or from associated sequences, and alter “the stream position” and conversion state as needed to maintain this subsequence relationship. The three pointers are:

— the beginning pointer, or lowest element address in the array (called \texttt{xbeg} here);
— the next pointer, or next element address that is a current candidate for reading or writing (called \texttt{xnext} here);
— the end pointer, or first element address beyond the end of the array (called \texttt{xend} here).

The following semantic constraints shall always apply for any set of three pointers for a sequence, using the pointer names given immediately above:

— If \texttt{xnext} is not a null pointer, then \texttt{xbeg} and \texttt{xend} shall also be non-null pointers into the same charT array, as described above; otherwise, \texttt{xbeg} and \texttt{xend} shall also be null.
— If \texttt{xnext} is not a null pointer and \texttt{xnext} < \texttt{xend} for an output sequence, then a write position is available. In this case, *\texttt{xnext} shall be assignable as the next element to write (to put, or to store a character value, into the sequence).
— If \texttt{xnext} is not a null pointer and \texttt{xbeg} < \texttt{xnext} for an input sequence, then a putback position is available. In this case, \texttt{xnext} [-1] shall have a defined value and is the next (preceding) element to store a character that is put back into the input sequence.
— If \texttt{xnext} is not a null pointer and \texttt{xnext} < \texttt{xend} for an input sequence, then a read position is available. In this case, *\texttt{xnext} shall have a defined value and is the next element to read (to get, or to obtain a character value, from the sequence).

27.5.2 Class template basic_streambuf<

class basic_streambuf {

// 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;

Draft
typedef traitstraits_type;

virtual ~basic_streambuf();

// 27.5.2.2.1 locales:
locale pubimbue(const locale& loc);
locale getloc() const;

// 27.5.2.2.2 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:
// 27.5.2.2.3 Get area:
streamsize in_avail();
int_type snextc();
int_type sbumpc();
int_type sgetc();
streamsize sgetn(char_type* s, streamsize n);

// 27.5.2.2.4 Putback:
int_type sputbackc(char_type c);
int_type sungetc();

// 27.5.2.2.5 Put area:
int_type sputc(char_type c);
streamsize sputn(const char_type* s, streamsize n);

protected:
basic_streambuf();
basic_streambuf(const basic_streambuf& rhs);
basic_streambuf& operator=(const basic_streambuf& rhs);

void swap(basic_streambuf& rhs);

// 27.5.2.3.2 Get area:
char_type* eback() const;
char_type* gptr() const;
char_type* egptr() const;
void gbump(int n);
void setg(char_type* beg, char_type* gnext, char_type* gend);

// 27.5.2.3.3 Put area:

Draft
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`:

- a character `input sequence`;
- a character `output sequence`.

[Note: This paragraph is intentionally empty. — end note]

[Note: This paragraph is intentionally empty. — end note]

### 27.5.2.1 `basic_streambuf` constructors

```cpp
basic_streambuf();
```
Effects: Constructs an object of class basic_streambuf<charT,traits> and initializes:\(^{300}\)
  — all its pointer member objects to null pointers,
  — the getloc() member to a copy the global locale, locale(), at the time of construction.

Remarks: Once the getloc() member is initialized, results of calling locale member functions, and of members of facets so obtained, can safely be cached until the next time the member imbue is called.

basic_streambuf(const basic_streambuf& rhs);

Effects: Constructs a copy of rhs.

Postconditions:
  — eback() == rhs.eback()
  — gptr() == rhs.gptr()
  — egptr() == rhs.egptr()
  — pbase() == rhs.pbase()
  — pptr() == rhs.pptr()
  — eptr() == rhs.eptr()
  — getloc() == rhs.getloc()

~basic_streambuf();

Effects: None.

27.5.2.2 basic_streambuf public member functions [streambuf.members]

27.5.2.2.1 Locales [streambuflocales]

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.

\(^{300}\) The default constructor is protected for class basic_streambuf to assure that only objects for classes derived from this class may be constructed.
27.5 Stream buffers

27.5.2.2 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 subsync();
4 Returns: sync().

27.5.2.2.3 Get area

streamsize in_avail();

1 Returns: If a read position is available, returns egptr() - gptr(). Otherwise returns showmanyc() (27.5.2.4.3).

int_type snextc();
2 Effects: Calls sbumpc().
3 Returns: if that function returns traits::eof(), returns traits::eof(). Otherwise, returns sgetc().

int_type sbumpc();
4 Returns: If the input sequence read position is not available, returns uflow(). Otherwise, returns traits::to_int_type(*gptr()) and increments the next pointer for the input sequence.

int_type sgetc();
5 Returns: If the input sequence read position is not available, returns underflow(). Otherwise, returns traits::to_int_type(*gptr()).

streamsize sgetn(char_type* s, streamsize n);
6 Returns: sgetn(s, n).

27.5.2.2.4 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();

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()).

27.5.2.2.5 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).

27.5.2.3 basic_streambuf protected member functions

27.5.2.3.1 Assignment

basic_streambuf& operator=(const basic_streambuf& rhs);

Effects: Assigns the data members of rhs to *this.

Postconditions:

— eback() == rhs.eback()
— gptr() == rhs.gptr()
— egptr() == rhs.egptr()
— pbase() == rhs.pbase()
— pptr() == rhs.pptr()
— eptr() == rhs.eptr()
— getloc() == rhs.getloc()

Returns: *this.

void swap(basic_streambuf& rhs);

Effects: Swaps the data members of rhs and *this.

27.5.2.3.2 Get area access

char_type* eback() const;

Returns: The beginning pointer for the input sequence.
char_type* gptr() const;
2
   *Returns:* The next pointer for the input sequence.

char_type* egptr() const;
3
   *Returns:* The end pointer for the input sequence.

void gbump(int n);
4
   *Effects:* Adds \( n \) to the next pointer for the input sequence.

void setg(char_type* gbeg, char_type* gnext, char_type* gend);
5
   *Postconditions:* \( gbeg == \texttt{eback}() \), \( gnext == \texttt{gptr}() \), and \( gend == \texttt{egptr}() \).

### 27.5.2.3.3 Put area access

char_type* pbase() const;
1
   *Returns:* The beginning pointer for the output sequence.

char_type* pptr() const;
2
   *Returns:* The next pointer for the output sequence.

char_type* epptr() const;
3
   *Returns:* The end pointer for the output sequence.

void pbump(int n);
4
   *Effects:* Adds \( n \) to the next pointer for the output sequence.

void setp(char_type* pbeg, char_type* pend);
5
   *Postconditions:* \( pbeg == \texttt{pbase}() \), \( pbeg == \texttt{pptr}() \), and \( pend == \texttt{epptr}() \).

### 27.5.2.4 basic_streambuf virtual functions

#### 27.5.2.4.1 Locales

void imbue(const locale&)
1
   *Effects:* Change any translations based on locale.

Remarks: Allows the derived class to be informed of changes in locale at the time they occur. Between invocations of this function a class derived from streambuf can safely cache results of calls to locale functions and to members of facets so obtained.

Default behavior: Does nothing.
27.5.2.4.2 Buffer management and positioning

basic_streambuf* setbuf(char_type* s, streamsize n);

**Effects:** Influences stream buffering in a way that is defined separately for each class derived from basic_streambuf in this clause (27.7.1.4, 27.8.1.5).

**Default behavior:** Does nothing. Returns this.

pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode which = ios_base::in | ios_base::out);

**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 (27.7.1.4, 27.8.1.5).

**Default behavior:** Returns pos_type(off_type(-1)).

pos_type seekpos(pos_type sp, ios_base::openmode which = ios_base::in | ios_base::out);

**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 (27.7.1, 27.8.1.1).

**Default behavior:** Returns pos_type(off_type(-1)).

int sync();

**Effects:** Synchronizes the controlled sequences with the arrays. That is, if pbase() is non-null the characters between pbase() and pptr() are written to the controlled sequence. The pointers may then be reset as appropriate.

**Returns:** -1 on failure. What constitutes failure is determined by each derived class (27.8.1.5).

**Default behavior:** Returns zero.

27.5.2.4.3 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);

---

301) The morphemes of showmanyc are "es-how-many-see", not "show-manic".

302) 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.”
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 \( \text{sbumpc()} \). Assigning stops when either \( n \) characters have been assigned or a call to \( \text{sbumpc()} \) would return \( \text{traits::eof()} \).

Returns: The number of characters assigned.\(^{303} \)

Remarks: Uses \( \text{traits::eof()} \).

\[ \text{int\_type \text{underflow}()} ; \]

Remarks: The public members of \texttt{basic\_streambuf} call this virtual function only if \( \text{gptr()} \) is null or \( \text{gptr()} \geq \text{egptr()} \).

Returns: \( \text{traits::to\_int\_type(c)} \), 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 \( \text{traits::eof()} \) to indicate failure.

The pending sequence of characters is defined as the concatenation of:

a) If \( \text{gptr()} \) is non-\( \text{NULL} \), then the \( \text{egptr()} - \text{gptr()} \) characters starting at \( \text{gptr()} \), 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:

a) If \( \text{eback()} \) is null then empty,

b) Otherwise the \( \text{gptr()} - \text{eback()} \) characters beginning at \( \text{eback()} \).

Effects: The function sets up the \( \text{gptr()} \) and \( \text{egptr()} \) satisfying one of:

a) If the pending sequence is non-empty, \( \text{egptr()} \) is non-\( \text{NULL} \) and \( \text{egptr()} - \text{gptr()} \) characters starting at \( \text{gptr()} \) are the characters in the pending sequence

b) If the pending sequence is empty, either \( \text{gptr()} \) is null or \( \text{gptr()} \) and \( \text{egptr()} \) are set to the same non-\( \text{NULL} \) pointer.

If \( \text{eback()} \) and \( \text{gptr()} \) are non-\( \text{NULL} \) 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 \( \text{gptr()} - \text{eback()} \) characters, then the \( \text{gptr()} - \text{eback()} \) characters starting at \( \text{eback()} \) agree with the last \( \text{gptr()} - \text{eback()} \) characters of the backup sequence.

b) Or the \( n \) characters starting at \( \text{gptr()} - n \) agree with the backup sequence (where \( n \) is the length of the backup sequence)

Default behavior: Returns \( \text{traits::eof()} \).

\(^{303} \) Classes derived from \texttt{basic\_streambuf} can provide more efficient ways to implement \text{xsgetn()} and \text{xsputn()} by overriding these definitions from the base class.
int_type uflow();

**Requires:** The constraints are the same as for underflow(), except that the result character is transferred from the pending sequence to the backup sequence, and the pending sequence may 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.

### 27.5.2.4.4 Putback

int_type pbackfail(int_type c = traits::eof());

**Remarks:** The public functions of basic_streambuf call this virtual function only when gptr() is null, gptr() == eback(), or traits::eq(traits::to_char_type(c),gptr()[−1]) returns false. Other calls shall also satisfy that constraint.

The pending sequence is defined as for underflow(), with the modifications that

- If traits::eq_int_type(c,traits::eof()) returns true, then the input sequence is backed up one character before the pending sequence is determined.
- If traits::eq_int_type(c,traits::eof()) return 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().

### 27.5.2.4.5 Put area

streamsize xsputn(const char_type* s, streamsize n);

**Effects:** Writes up to n characters to the output sequence as if by repeated calls to sputc(c). The characters written are obtained from successive elements of the array whose first element is designated by s. Writing stops when either n characters have been written or a call to sputc(c) would return traits::eof().

**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

Draft
a) if \( \text{pbase}() \) is NULL then the empty sequence otherwise, \( \text{pptr}() - \text{pbase}() \) characters beginning at \( \text{pbase}() \).

b) if \( \text{traits}::\text{eq_int_type}(c, \text{traits}::\text{eof})() \) returns true, then the empty sequence otherwise, the sequence consisting of \( c \).

**Remarks:** The member functions \( \text{sputc()} \) and \( \text{sputn()} \) 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.

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 NULL, or \( \text{pbase}() \) and \( \text{pptr}() \) are both set to the same NULL non-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:** \( \text{traits}::\text{eof}() \) or throws an exception if the function fails.

Otherwise, returns some value other than \( \text{traits}::\text{eof}() \) to indicate success.

**Default behavior:** Returns \( \text{traits}::\text{eof}() \).

### 27.6 Formatting and manipulators

**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;

template <class charT, class traits = char_traits<charT> >
    class basic_iostream;
    typedef basic_iostream<char> iostream;
    typedef basic_iostream<wchar_t> wiostream;

template <class charT, class traits>
    basic_istream<charT,traits>& ws(basic_istream<charT,traits>& is);
}
```

**Header `<ostream>` synopsis**

```cpp
namespace std {
    template <class charT, class traits = char_traits<charT> >
}
```

---

304) That is, for each class derived from an instance of `basic_streambuf` in this clause (27.7.1, 27.8.1.1), a specification of how consuming a character effects the associated output sequence is given. There is no requirement on a program-defined class.

305) Typically, `overflow` returns \( c \) to indicate success, except when \( \text{traits}::\text{eq_int_type}(c, \text{traits}::\text{eof})() \) returns true, in which case it returns `traits::not_eof(c)`. Draft
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);

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);
}

27.6.1 Input streams

The header <iostream> defines two types and a function signature that control input from a stream buffer.

27.6.1.1 Class template basic_istream

namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_istream : virtual public basic_ios<charT, traits> { 
  public:
    // Types (inherited from basic_ios (27.4.4)):
    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;

    // 27.6.1.1.1 Constructor/destructor:
    explicit basic_istream(basic_streambuf<charT, traits>* sb);
    basic_istream(basic_istream& rhs);
    virtual ~basic_istream();

    // 27.6.1.1.2 Assign/swap:
    basic_istream& operator=(basic_istream& rhs);
    void swap(basic_istream& rhs);

    
    Draft
27.6 Formatting and manipulators

// 27.6.1.3 Prefix/suffix:
class sentry;

// 27.6.1.2 Formatted input:

basic_istream<charT,traits>& operator>>(
    basic_istream<charT,traits>& (*pf)(basic_istream<charT,traits>&));
basic_istream<charT,traits>& operator>>(
    basic_istream<charT,traits>& (*pf)(basic_istream<charT,traits>&));
basic_istream<charT,traits>& operator>>(
    ios_base& (*pf)(ios_base&));
basic_istream<charT,traits>& operator>>(
    bool& n);
basic_istream<charT,traits>& operator>>(
    short& n);
basic_istream<charT,traits>& operator>>(
    unsigned short& n);
basic_istream<charT,traits>& operator>>(
    int& n);
basic_istream<charT,traits>& operator>>(
    unsigned int& n);
basic_istream<charT,traits>& operator>>(
    long& n);
basic_istream<charT,traits>& operator>>(
    unsigned long& n);
basic_istream<charT,traits>& operator>>(
    long long& n);
basic_istream<charT,traits>& operator>>(
    unsigned long long& n);
basic_istream<charT,traits>& operator>>(
    float& f);
basic_istream<charT,traits>& operator>>(
    double& f);
basic_istream<charT,traits>& operator>>(
    long double& f);
basic_istream<charT,traits>& operator>>(
    void*& p);
basic_istream<charT,traits>& operator>>(
    basic_streambuf<char_type,traits>* sb);

// 27.6.1.3 Unformatted input:

streamsize gcount() const;
int_type get();
basic_istream<charT,traits>& get(char_type& c);
basic_istream<charT,traits>& get(char_type* s, streamsize n);
basic_istream<charT,traits>& get(char_type* s, streamsize n,
    char_type delim);
basic_istream<charT,traits>& get(basic_streambuf<char_type,traits>& sb);
basic_istream<charT,traits>& get(basic_streambuf<char_type,traits>& sb,
    char_type delim);
basic_istream<charT,traits>& getline(char_type* s, streamsize n);
basic_istream<charT,traits>& getline(char_type* s, streamsize n,
    char_type delim);
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);
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()->sgetc()` or `rdbuf()->sbumpc()`. They may use other public members of `istream`. If `rdbuf()->sgetc()` or `rdbuf()->sbumpc()` 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` (27.4.4.3), before returning.

If one of these called functions throws an exception, then unless explicitly noted otherwise, the input function sets

Draft
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.

27.6.1.1.1 basic_istream constructors

```cpp
effect 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
2   basic_ios::init(sb)(27.4.4.1).
3   Postcondition: gcount() == 0

basic_istream(basic_istream&& rhs);
4   Effects: Move constructs from the rvalue rhs. This is accomplished by default constructing the base class, copying
5   the gcount() from rhs, calling basic_ios<charT, traits>::move(rhs) to initialize the base class, and
6   setting the gcount() for rhs to 0.

virtual ~basic_istream();
7   Effects: Destroys an object of class basic_istream.
8   Remarks: Does not perform any operations of rdbuf().
```

27.6.1.1.2 Class basic_istream assign and swap

```cpp
basic_istream& operator=(basic_istream&& rhs);
1   Effects: swap(rhs);
2   Returns: *this.

void swap(basic_istream&& rhs);
3   Effects: Calls basic_ios<charT, traits>::swap(rhs). Exchanges the values returned by gcount() and
4   rhs.gcount().

template <class charT, class traits>
void swap(basic_istream<charT, traits>& x, basic_istream<charT, traits>& y);
```

27.6.1.1.3 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;
    }
}
Draft
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 true, 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.\(^{306}\) 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(27.4.4.3)`).\(^{307}\)

```cpp
~sentry();
```

**Effects:** None.

```cpp
operator bool() const; = delete; // not defined
```

**Effects:** Returns `ok_`.

---

\(^{306}\) 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.

\(^{307}\) The sentry constructor and destructor can also perform additional implementation-dependent operations.
27.6 Formatting and manipulators

27.6.1.2 Formatted input functions

27.6.1.2.1 Common requirements

Each formatted input function begins execution by constructing an object of class `sentry` with the `noskipws` (second) argument `false`. If the `sentry` object returns `true`, when converted to a value of type `bool`, the function endeavors to obtain the requested input. If an exception is thrown during input then `ios::badbit` is turned on 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`.

27.6.1.2.2 Arithmetic Extractors

As in the case of the inserters, these extractors depend on the locale’s `num_get<>` (22.2.2.1) object to perform parsing the input stream data. These extractors behave as formatted input functions (as described in 27.6.1.2.1). 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 = 0;
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`.

```cpp
operator>>(short& val);
```

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 = 0;
long lval;
use_facet<numget>(loc).get(*this, 0, *this, err, lval);
if (err == 0)
    && (lval < numeric_limits<short>::min() || numeric_limits<short>::max() < lval))
```

308 This is done without causing an `ios::failure` to be thrown.
err = ios_base::failbit;
setstate(err);

operator>>(int& val);

The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

typedef num_get<CharT, istreambuf_iterator<CharT,traits> > numget;
iostate err = 0;
long lval;
use_facet<numget>(loc).get(*this, 0, *this, err, lval);
if (err == 0)
  && (lval < numeric_limits<int>::min()
      || numeric_limits<int>::max() < lval))
  err = ios_base::failbit;
setstate(err);

27.6.1.2.3 basic_istream::operator>>

[istream::extractors]

basic_istream& operator>>(basic_istream& (*pf)(basic_istream&))

Effects: None. This extractor does not behave as a formatted input function (as described in 27.6.1.2.1.)

Returns: *pf(*this).\(^{309}\)

basic_istream& operator>>(basic_istream& (*pf)(basic_istream&))

Effects: Calls *pf(*this). This extractor does not behave as a formatted input function (as described in 27.6.1.2.1.)

Returns: *this.

basic_istream& operator>>(ios_base& (*pf)(ios_base&))

Effects: Calls *pf(*this).\(^{310}\) This extractor does not behave as a formatted input function (as described in 27.6.1.2.1.)

Returns: *this.

template<class CharT, class Traits>
  basic_istream& operator>>(basic_istream&& in, CharT* s);

template<class Traits>
  basic_istream& operator>>(basic_istream&& in, unsigned char* s);

template<class Traits>

\(^{309}\) See, for example, the function signature `ws(basic_istream&)` (27.6.1.4).

\(^{310}\) See, for example, the function signature `dec(ios_base&)` (27.4.5.3).
basic_istream<char,traits>& operator>>(basic_istream<char,traits>&& in,  
    signed char* s);

Effects: Behaves like a formatted input member (as described in 27.6.1.2.1) of in. After a sentry object is  
constructed, operator>> extracts characters and stores them into successive locations of an array whose first  
element is designated by s. If width() is greater than zero, n is width(). Otherwise n is the number of  
elements of the largest array of char_type that can store a terminating charT(). n is the maximum number of  
characters stored.

Characters are extracted and stored until any of the following occurs:

- 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  
(27.4.4.3).

Returns: in.

template<class charT, class traits>  
basic_istream<charT,traits>& operator>>(basic_istream<charT,traits>&& in,  
    char& 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 27.6.1.2.1) 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 a formatted input function (as described in 27.6.1.2.1). If sb is null, calls setstate(fail-
bit), which may throw ios_base::failure (27.4.4.3). 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` (27.4.4.3). If it inserted no characters because it caught an exception thrown while extracting characters from `*this` and `failbit` is on in `exceptions()` (27.4.4.3), then the caught exception is rethrown.

**Returns:** `*this`.

### 27.6.1.3 Unformatted input functions [istream.unformatted]

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 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 27.6.1.3, 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 27.6.1.3, 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` (27.4.4.3).

**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 27.6.1.3, paragraph 1). After constructing a `sentry` object, extracts a character, if one is available, and assigns it to `c`. Otherwise, the function calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).

**Returns:** `*this`.

```cpp
basic_istream<charT,traits>& get(char_type* s, streamsize n, char_type delim);
```

---

(311) This is done without causing an `ios::failure` to be thrown.

(312) Note that this function is not overloaded on types `signed char` and `unsigned char`.

Draft
Effects: Behaves as an unformatted input function (as described in 27.6.1.3, 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 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`). In any case, if \( n \) is greater than zero it then stores a null character into the next successive location of the array.

Returns: *this.

basic_istream<charT,traits>& get(char_type* s, streamsize n)

Effects: Calls get(s, n, widen(‘\n’))

Returns: Value returned by the call.

basic_istream<charT,traits>& get(basic_streambuf<char_type,traits>& sb, char_type delim);

Effects: Behaves as an unformatted input function (as described in 27.6.1.3, 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:

- end-of-file occurs on the input sequence;
- inserting in the output sequence fails (in which case the character to be inserted is not extracted);
- traits::eq(c, delim) for the next available input character c (in which case c is not extracted);
- an exception occurs (in which case, the exception is caught but not rethrown).

If the function inserts no characters, it calls `setstate(failbit)`, which may throw `ios_base::failure`.

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 27.6.1.3, 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:

\[ \text{Note that this function is not overloaded on types signed char and unsigned char.} \]

\[ \text{Note that this function is not overloaded on types signed char and unsigned char.} \]
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);\(^{315}\)
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.\(^{316}\)

If the function extracts no characters, it calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).\(^{317}\)

In any case, if \(n\) is greater than zero, it then stores a null character (using `charT()`) into the next successive location of the array.

Returns: `*this`.

[Example:]
```cpp
#include <iostream>

int main()
{
    using namespace std;
    const int line_buffer_size = 100;

    char buffer[line_buffer_size];
    int line_number = 0;
    while (cin.getline(buffer, line_buffer_size, '\n') || cin.gcount()) {
        int count = cin.gcount();
        if (cin.eof())
            cout << "Partial final line"; // cin.fail() is false
        else if (cin.fail()) {
            cout << "Partial long line";
            cin.clear(cin.rdstate() & ~ios::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\'))`

\(^{315}\) Since the final input character is “extracted,” it is counted in the `gcount()`, even though it is not stored.

\(^{316}\) This allows an input line which exactly fills the buffer, without setting `failbit`. This is different behavior than the historical AT&T implementation.

\(^{317}\) This implies an empty input line will not cause `failbit` to be set.
### 27.6 Formatting and manipulators

**basic_istream<charT,traits>&**

```cpp
ignore(streamsize n = 1, int_type delim = traits::eof());
```

*Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1). After constructing a sentry object, extracts characters and discards them. Characters are extracted until any of the following occurs:

- if `n` != `numeric_limits<streamsize>::max()` (18.2.1), `n` characters are extracted
- end-of-file occurs on the input sequence (in which case the function calls `setstate(eofbit)`, which may throw `ios_base::failure (27.4.4.3)`);
- `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`.

```cpp
int_type peek();
```

*Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, 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()`.

```cpp
basic_istream<charT,traits>& read(char_type* s, streamsize n);
```

*Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, 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:

- `n` characters are stored;
- end-of-file occurs on the input sequence (in which case the function calls `setstate(failbit|eofbit)`, which may throw `ios_base::failure (27.4.4.3)`).

*Returns:* `*this`.

```cpp
streamsize readsome(char_type* s, streamsize n);
```

*Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, 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 (27.4.4.3)`), and extracts no characters;

- If `rdbuf()->in_avail() == 0`, extracts no characters
- If `rdbuf()->in_avail() > 0`, extracts `min(rdbuf()->in_avail(), n))`.

*Returns:* The number of characters extracted.

---

318) Note that this function is not overloaded on types `signed char` and `unsigned char`.

Draft
basic_istream<charT,traits>& putback(char_type c);

Effects: Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1). 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 (27.4.4.3)). [Note: this function extracts no characters, so the value returned by the next call to gcount() is 0. — end note]

Returns: *this.

basic_istream<charT,traits>& unget();

Effects: Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1). 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 (27.4.4.3)). [Note: this function extracts no characters, so the value returned by the next call to gcount() is 0. — end note]

Returns: *this.

int sync();

Effects: Behaves as an unformatted input function (as described in 27.6.1.3, 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 (27.4.4.3), and returns -1. Otherwise, returns zero.

pos_type tellg();

Effects: Behaves as an unformatted input function (as described in 27.6.1.3, 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).

basic_istream<charT,traits>& seekg(pos_type pos);

Effects: Behaves as an unformatted input function (as described in 27.6.1.3, 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()->pubseekpos(pos, ios_base::in). In case of failure, the function calls setstate(failbit) (which may throw ios_base::failure).

Returns: *this.

basic_istream<charT,traits>& seekg(off_type off, ios_base::seekdir dir);

Effects: Behaves as an unformatted input function (as described in 27.6.1.3, 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).
27.6 Formatting and manipulators

Returns: *this.

27.6.1.4 Standard basic_istream manipulators

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 27.6.1.3, 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 (27.6.1.1.3). If ws stops extracting characters because there are no more available it sets eofbit, but not failbit.

Returns: is.

27.6.1.5 Class template basic_iostream

namespace std {
    template <class charT, class traits = char_traits<charT>>
    class basic_iostream :
        public basic_istream<charT,traits>,
        public basic_ostream<charT,traits>{
        public:
            // 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
            explicit basic_iostream(basic_streambuf<charT,traits>* sb);
            basic_iostream(basic_iostream& rhs);
            virtual ~basic_iostream();

            // assign/swap
            basic_iostream& operator=(basic_iostream& rhs);
            void swap(basic_iostream& rhs);
        };

        template <class charT, class traits>
        void swap(basic_iostream<charT, traits>& x, basic_iostream<charT, traits>& y);
        template <class charT, class traits>
        void swap(basic_iostream<charT, traits>&& x, basic_iostream<charT, traits>& y);
        template <class charT, class traits>
        void swap(basic_iostream<charT, traits>&& x, basic_iostream<charT, traits>& y);

        template <class charT, class traits>

Draft
The class `basic_iostream` inherits a number of functions that allow reading input and writing output to sequences controlled by a stream buffer.

### 27.6.1.5.1 basic_iostream constructors [iostream.cons]

```cpp
effect 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)` (27.6.1.1) and `basic_ostream<charT,traits>(sb)` (27.6.2.1)

**Postcondition:** `rdbuf()==sb` and `gcount()==0`.

```cpp
basic_iostream(basic_iostream&& rhs);
```

**Effects:** Move constructs from the rvalue `rhs` by constructing the `basic_istream` base class with `move(rhs)`.

### 27.6.1.5.2 basic_iostream destructor [iostream.dest]

```cpp
virtual ~basic_iostream();
```

**Effects:** Destroys an object of class `basic_iostream`.

**Remarks:** Does not perform any operations on `rdbuf()`.

### 27.6.1.5.3 basic_iostream assign and swap [iostream.assign]

```cpp
basic_iostream& operator=(basic_iostream&& rhs);
```

**Effects:** `swap(rhs)`.

```cpp
void swap(basic_iostream&& rhs);
```

**Effects:** Calls `basic_istream<charT, traits>::swap(rhs)`.

```cpp
template <class charT, class traits>
void swap(basic_iostream<charT, traits>& x, basic_iostream<charT, traits>&& y);
template <class charT, class traits>
void swap(basic_iostream<charT, traits>&& x, basic_iostream<charT, traits>& y);
template <class charT, class traits>
void swap(basic_iostream<charT, traits>& x, basic_iostream<charT, traits>&& y);
```

**Effects:** `x.swap(y)`.

### 27.6.2 Output streams [output.streams]

The header `<ostream>` defines a type and several function signatures that control output to a stream buffer.

### 27.6.2.1 Class template basic_ostream [ostream]

Draft
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 (27.4.4)):
    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;

    // 27.6.2.2 Constructor/destructor:
    explicit basic_ostream(basic_streambuf<char_type, traits>* sb);
    basic_ostream(basic_ostream&& rhs);
    virtual ~basic_ostream();

    // 27.6.2.3 Assign/swap
    basic_ostream& operator=((basic_ostream&& rhs);
    void swap(basic_ostream&& rhs);

    // 27.6.2.4 Prefix/suffix:
    class sentry;

    // 27.6.2.6 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<<(float f);
    basic_ostream<charT, traits>& operator<<(double f);
    basic_ostream<charT, traits>& operator<<(long double f);
    basic_ostream<charT, traits>& operator<<(const void* p);
    basic_ostream<charT, traits>& operator<<
        (basic_streambuf<char_type, traits>* sb);

    // 27.6.2.7 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();

// 27.6.2.5 seeks:
pos_type tellp();
basic_ostream<charT, traits>& seekp(pos_type);
basic_ostream<charT, traits>& seekp(off_type, ios_base::seekdir);

// 27.6.2.6.4 character inserters
template<class charT, class traits>
  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&, charT);

template<class charT, class traits>
  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&&, charT);

template<class charT, class traits>
  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&, char);

template<class charT, class traits>
  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&&, char);

// signed and unsigned
template<class traits>
  basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, signed char);

template<class traits>
  basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&&, signed char);

template<class traits>
  basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, unsigned char);

template<class traits>
  basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&&, unsigned char);

template<class charT, class traits>
  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&, const charT*);

template<class charT, class traits>
  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&&, const charT*);

Draft
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()`.

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.

### 27.6.2.2 `basic_ostream` constructors

```cpp
explicit basic_ostream(basic_streambuf<charT,traits>* sb);
```
Effects: Constructs an object of class basic_ostream, assigning initial values to the base class by calling basic_ios<charT,traits>::init(sb) (27.4.4.1).

Postcondition: rdbuf() == sb.

virtual ~basic_ostream();

Effects: Destroys an object of class basic_ostream.

Remarks: Does not perform any operations on rdbuf().

basic_ostream(basic_ostream&& rhs);

Effects: Move constructs from the rvalue rhs. This is accomplished by default constructing the base class and calling basic_ios<charT, traits>::move(rhs) to initialize the base class.

27.6.2.3 Class basic_ostream assign and swap

basic_ostream& operator=((basic_ostream&& rhs);

Effects: swap(rhs).

Returns: *this.

void swap(basic_ostream&& rhs);

Effects: Calls basic_ios<charT, traits>::swap(rhs).

template <class charT, class traits>
void swap(basic_ostream<charT, traits>& x, basic_ostream<charT, traits>& y);
template <class charT, class traits>
void swap(basic_ostream<charT, traits>&& x, basic_ostream<charT, traits>& y);
template <class charT, class traits>
void swap(basic_ostream<charT, traits>& x, basic_ostream<charT, traits>&& y);

Effects: x.swap(y).

27.6.2.4 Class basic_ostream::sentry

namespace std {

template <class charT, class traits = char_traits<charT> >
class basic_ostream<charT,traits>::sentry {

// bool ok_;

public:
explicit sentry(basic_ostream<charT,traits>& os);
~sentry();
operator bool() const { return ok_; }

private:
    sentry(const sentry&) = delete; // not defined
    sentry& operator=(const sentry&) = delete; // not defined
};
}

Draft
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(27.4.4.3)`).  

```cpp
~sentry();
```

If `((os.flags() & ios_base::unitbuf) && !uncaught_exception())` is true, calls `os.flush()`.

```cpp
operator bool() const;
```

Effects: Returns `ok_`.

### 27.6.2.5 basic_ostream seek members

```cpp
pos_type tellp();
```

Returns: if `fail()` != false, returns `pos_type(-1)` to indicate failure. Otherwise, returns `rdbuf()->pubseekoff(0, cur, out)`.

```cpp
basic_ostream<charT,traits>& seekp(pos_type pos);
```

Effects: If `fail()` != true, executes `rdbuf()->pubseekpos(pos, ios_base::out)`.

Returns: `*this`.

```cpp
basic_ostream<charT,traits>& seekp(off_type off, ios_base::seekdir dir);
```

Effects: If `fail()` != true, executes `rdbuf()->pubseekoff(off, dir, ios_base::out)`.

Returns: `*this`.

### 27.6.6 Formatted output functions

#### 27.6.6.1 Common requirements

Each formatted output function begins execution by constructing an object of class `sentry`. If this object returns `true` when converted to a value of type `bool`, the function endeavors to generate the requested output. If the generation fails, then the formatted output function does `setstate(ios::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`.

---

319) The call `os.tie()->flush()` does not necessarily occur if the function can determine that no synchronization is necessary.

320) The `sentry` constructor and destructor can also perform additional implementation-dependent operations.

321) Without causing an `ios::failure` to be thrown.
The descriptions of the individual formatted output operations describe how they perform output and do not mention the sentry object.

### 27.6.2.6.2 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:
bool failed = use_facet<
    num_put<charT,ostreambuf_iterator<charT,traits> >
    >(getloc()).put(*this, *this, fill(),
    static_cast<unsigned long>(val)).failed();

When val is of type float the formatting conversion occurs as if it performed the following code fragment:

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 streams 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.

27.6.2.6.3 basic_ostream::operator<<

basic_ostream<
    charT,traits>& operator<<
    (basic_ostream<
        charT,traits>& (*pf)(basic_ostream<
            charT,traits>&))
1
    Effects: None. Does not behave as a formatted output function (as described in 27.6.2.6.1).
2
    Returns: pf(*this).\(^{322}\)

basic_ostream<
    charT,traits>& operator<<
    (basic_ios<
        charT,traits>& (*pf)(basic_ios<
            charT,traits>&))
3
    Effects: Calls pf(*this). This inserter does not behave as a formatted output function (as described in 27.6.2.6.1).
4
    Returns: *this.\(^{323}\)

basic_ostream<
    charT,traits>& operator<<
    (ios_base& (*pf)(ios_base&))
5
    Effects: Calls pf(*this). This inserter does not behave as a formatted output function (as described in 27.6.2.6.1).
6
    Returns: *this.

basic_ostream<
    charT,traits>& operator<<
    (basic_streambuf<
        charT,traits>* sb);
7
    Effects: Behaves as a formatted output function (as described in 27.6.2.6.1). After the sentry object is constructed, if sb is null calls setstate(badbit) (which may throw ios_base::failure).

\(^{322}\) See, for example, the function signature endl(basic_ostream&) (27.6.2.8).

\(^{323}\) See, for example, the function signature dec(ios_base&) (27.4.5.3).
Gets characters from `sb` and inserts them in `*this`. Characters are read from `sb` and inserted until any of the following occurs:

- end-of-file occurs on the input sequence;
- inserting in the output sequence fails (in which case the character to be inserted is not extracted);
- an exception occurs while getting a character from `sb`.

If the function inserts no characters, it calls `setstate(failbit)` (which may throw `ios_base::failure (27.4.4.3)`). 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`.

### 27.6.2.6.4 Character inserter function templates

```cpp
// specialization
template<class traits>
basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out, char c);
```

```cpp
// 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 like a formatted inserter (as described in 27.6.2.6.1) of `out`. After a `sentry` object is constructed it inserts characters. In case `c` has type `char` and the character type of the stream is not `char`, then the character to be inserted is `out.widen(c)`; otherwise the character is `c`. Padding is determined as described in 22.2.2.2. `width(0)` is called. The insertion character and any required padding are inserted into `out`.

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 charT* s);
```

Requires: `s` is non-null.

Effects: Behaves like a formatted inserter (as described in 27.6.2.6.1) of `out`. After a `sentry` object is constructed it inserts `n` characters starting at `s`, where `n` is the number that would be computed as if by:

- `traits::length(s)` for the overload where the first argument is of type `basic_ostream<charT, traits>&` and the second is of type `const charT*`, and also for the overload where the first argument is of type `basic_ostream<char, traits>&` and the second is of type `const char*`,
- `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.
Padding is determined as described in 22.2.2.2. The n characters starting at s are widened using out.widen (27.4.4.2). The widened characters and any required padding are inserted into out. Calls width(0).

Returns: out.

27.6.2.7 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 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 27.6.2.7, paragraph 1). After constructing a sentry object, inserts the character c, if possible. Otherwise, calls setstate(badbit) (which may throw ios_base::failure (27.4.4.3)).

Returns: *this.

basic_ostream& write(const char_type* s, streamsize n);

Effects: Behaves as an unformatted output function (as described in 27.6.2.7, paragraph 1). After constructing a sentry object, obtains characters to insert from successive locations of an array whose first element is designated by s. Characters are inserted until either of the following occurs:

— n characters are inserted;

— inserting in the output sequence fails (in which case the function calls setstate(badbit), which may throw ios_base::failure (27.4.4.3)).

Returns: *this.

basic_ostream& flush();

If rdbuf() is not a null pointer, calls rdbuf()->pubsync(). If that function returns -1 calls setstate(badbit) (which may throw ios_base::failure (27.4.4.3)). Does not behave as an unformatted output function (as described in 27.6.2.7, paragraph 1).

Returns: *this.

27.6.2.8 Standard basic_ostream manipulators

namespace std {
    template <class charT, class traits>
    basic_ostream<charT,traits>& endl(basic_ostream<charT,traits>& os);

324) without causing an ios::failure to be thrown.
325) Note that this function is not overloaded on types signed char and unsigned char.
326) Note that this function is not overloaded on types signed char and unsigned char.
### 27.6 Formatting and manipulators

The header `<iomanip>` defines a type and several related functions that use this type to provide extractors and inserters that alter information maintained by class `ios_base` and its derived classes.

The type designated `smanip` in each of the following function descriptions is implementation-defined and may be different for each function.

```cpp
smanip resetiosflags(ios_base::fmtflags mask);
```

**Returns:** An object `s` of unspecified type such that if `out` is an instance of `basic_ostream<charT,traits>` then the expression `out<<s` behaves as if `f(s, mask)` were called, or if `in` is an instance of `basic_istream<charT,traits>` then the expression `in>>s` behaves as if `f(s, mask)` were called. The function `f` can be defined as:

```cpp
ios_base& f(ios_base& str, ios_base::fmtflags mask)
{
    // reset specified flags
    str.setf(ios_base::fmtflags(0), mask);
    return str;
}
```

The expression `out<<s` has type `basic_ostream<charT,traits>&` and value `out`. The expression `in>>s` has type `basic_istream<charT,traits>&` and value `in`.

```cpp
smanip setiosflags(ios_base::fmtflags mask);
```

---

327) The expression `cin >> resetiosflags(ios_base::skipws)` clears `ios_base::skipws` in the format flags stored in the `basic_istream<charT,traits>` object `cin` (the same as `cin >> noskipws`), and the expression `cout << resetiosflags(ios_base::showbase)` clears `ios_base::showbase` in the format flags stored in the `basic_ostream<charT,traits>` object `cout` (the same as `cout << noshowbase`).
Returns: An object \( s \) of unspecified type such that if \( \text{out} \) is an instance of basic_ostream<\( \text{charT}, \text{traits} \)>, then the expression \( \text{out} \ll s \) behaves as if \( f(s, \text{mask}) \) were called, or if \( \text{in} \) is an instance of basic_istream<\( \text{charT}, \text{traits} \)>, then the expression \( \text{in} \gg s \) behaves as if \( f(s, \text{mask}) \) were called. The function \( f \) can be defined as:

\[
\begin{aligned}
\text{ios_base}&\ f(\text{ios_base}& \; \text{str}, \text{ios_base}::\text{fmtflags} \; \text{mask}) \\
&\{ \\
&\quad \text{// set specified flags} \\
&\quad \text{str}.\text{setf}(\text{mask}); \\
&\quad \text{return} \; \text{str}; \\
&\}
\end{aligned}
\]

The expression \( \text{out} \ll s \) has type basic_ostream<\( \text{charT}, \text{traits} \)& and value \( \text{out} \). The expression \( \text{in} \gg s \) has type basic_istream<\( \text{charT}, \text{traits} \)& and value \( \text{in} \).

\smanip ssetbase(int base);

Returns: An object \( s \) of unspecified type such that if \( \text{out} \) is an instance of basic_ostream<\( \text{charT}, \text{traits} \)>, then the expression \( \text{out} \ll s \) behaves as if \( f(s) \) were called, or if \( \text{in} \) is an instance of basic_istream<\( \text{charT}, \text{traits} \)>, then the expression \( \text{in} \gg s \) behaves as if \( f(s, \text{base}) \) were called. The function \( f \) can be defined as:

\[
\begin{aligned}
\text{ios_base}&\ f(\text{ios_base}& \; \text{str}, \text{int} \; \text{base}) \\
&\{ \\
&\quad \text{// set basefield} \\
&\quad \text{str}.\text{setf}(\text{base} \; \text{==} \; 8 \; ? \; \text{ios_base}::\text{oct} : \\
&\quad \quad \text{base} \; \text{==} \; 10 \; ? \; \text{ios_base}::\text{dec} : \\
&\quad \quad \quad \text{base} \; \text{==} \; 16 \; ? \; \text{ios_base}::\text{hex} : \\
&\quad \quad \quad \quad \text{ios_base}::\text{fmtflags}(0), \text{ios_base}::\text{basefield}); \\
&\quad \text{return} \; \text{str}; \\
&\}
\end{aligned}
\]

The expression \( \text{out} \ll s \) has type basic_ostream<\( \text{charT}, \text{traits} \)& and value \( \text{out} \). The expression \( \text{in} \gg s \) has type basic_istream<\( \text{charT}, \text{traits} \)& and value \( \text{in} \).

\smanip ssetfill(char_type c);

Returns: An object \( s \) of unspecified type such that if \( \text{out} \) is (or is derived from) basic_ostream<\( \text{charT}, \text{traits} \)>, and \( c \) has type \( \text{charT} \) then the expression \( \text{out} \ll s \) behaves as if \( f(s, c) \) were called. The function \( f \) can be defined as:

\[
\begin{aligned}
\text{template}&\langle\text{class} \; \text{charT}, \; \text{class} \; \text{traits} \rangle \\
\text{basic_ios}<\text{charT},\text{traits}>\& \; f(\text{basic_ios}<\text{charT},\text{traits}>\& \; \text{str}, \; \text{charT} \; c) \\
&\{ \\
&\quad \text{// set fill character} \\
&\quad \text{str}.\text{fill}(c); \\
&\quad \text{return} \; \text{str}; \\
&\}
\end{aligned}
\]

The expression \( \text{out} \ll s \) has type basic_ostream<\( \text{charT}, \text{traits} \)& and value \( \text{out} \).

\smanip ssetprecision(int n);
Returns: An object \( s \) of unspecified type such that if \( \text{out} \) is an instance of `basic_ostream<charT, traits>` then the expression \( \text{out} \ll s \) behaves as if \( f(s, n) \) were called, or if \( \text{in} \) is an instance of `basic_istream<charT, traits>` then the expression \( \text{in} \gg s \) behaves as if \( f(s, n) \) were called. The function \( f \) can be defined as:

```cpp
templates_base& f(templates_base& \text{str}, \text{int } n)
{
    // set precision
    \text{str}.precision(n);
    \text{return str;
}
```

The expression \( \text{out} \ll s \) has type `basic_ostream<charT, traits>&` and value `\text{out}`. The expression \( \text{in} \gg s \) has type `basic_istream<charT, traits>&` and value `\text{in}`.

smanip `setw(int n)`;

Returns: An object \( s \) of unspecified type such that if \( \text{out} \) is an (instance of) `basic_ostream<charT, traits>` then the expression \( \text{out} \ll s \) behaves as if \( f(s, n) \) were called, or if \( \text{in} \) is an instance of `basic_istream<charT, traits>` then the expression \( \text{in} \gg s \) behaves as if \( f(s, n) \) were called. The function \( f \) can be defined as:

```cpp
templates_base& f(templates_base& \text{str}, \text{int } n)
{
    // set width
    \text{str}.width(n);
    \text{return str;
}
```

The expression \( \text{out} \ll s \) has type `basic_ostream<charT, traits>&` and value `\text{out}`. The expression \( \text{in} \gg s \) has type `basic_istream<charT, traits>&` and value `\text{in}`.

### 27.6.4 Extended Manipulators

The header `<iomanip>` defines a type and several related functions that use this type to provide extractors and inserter

```cpp
templates smanip get_money(templates moneyT& mon, bool intl = false);
```

Requires: The type \( \text{moneyT} \) shall be either `long double` or a specialization of the `basic_string` template (Clause 21).

Returns: An object \( s \) of unspecified type such that if \( \text{in} \) is an instance of `basic_istream<charT, traits>` then the expression \( \text{in} \gg \text{get_money}(\text{mon}, \text{intl}) \) behaves as if \( f(\text{in}, \text{mon}, \text{intl}) \) were called. The function \( f \) can be defined as:

```cpp
templates charT, \text{class traits}, \text{class moneyT}>
void f(basic_ios<charT, traits>& \text{str}, moneyT& \text{mon}, bool intl)
{
    typedef istreambuf_iterator<charT> Iter;
    Iter it(\text{str});
    // parse and format
    f(it, \text{mon}, \text{intl});
}
```
typedef money_get<charT, Iter> MoneyGet;

ios_base::iostate err = ios_base::goodbit;
const MoneyGet &mg = use_facet<money_get>(str.getloc());

mg.get(Iter(str.rdbuf()), Iter(), intl, str, err, mon);

if (ios_base::goodbit != err)
    str.setstate(err);
}

The expression `in >> s` has type `basic_istream<charT, traits>&` and value `in`.

```cpp
template <class charT, class moneyT>
smnpp put_money(const moneyT& mon, bool intl = false;

1 Requires: The type `moneyT` shall be either `long double` or a specialization of the basic_string template (Clause 21).

2 Returns: An object `s` 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 if `f(out, mon, intl)` were called. The function `f` can be 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> Iter;
    typedef money_put<charT, Iter> MoneyPut;

    const MoneyPut &mp = use_facet<money_put>(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 << s` has type `basic_ostream<charT, traits>&` and value `out`.

### 27.7 String-based streams

The header `<sstream>` defines four class templates and eight types that associate stream buffers with objects of class `basic_string`, as described in 21.2.

**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;
```
27.7 String-based streams

namespace std {
    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;
}

27.7.1 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;

        // 27.7.1.1 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(basic_stringbuf&& rhs);

        // 27.7.1.2 Assign and swap:
        basic_stringbuf& operator=(basic_stringbuf&& rhs);
        void swap(basic_stringbuf& rhs);

        // 27.7.1.3 Get and set:
        basic_string<charT,traits,Allocator> str() const;
        void str(const basic_string<charT,traits,Allocator>& s);
    }
}
protected:
    // 27.7.1.4 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);

} // namespace

The class basic_stringbuf is derived from basic_streambuf to associate possibly the input sequence and possibly the output sequence with a sequence of arbitrary characters. The sequence can be initialized from, or made available as, an object of class basic_string.

For the sake of exposition, the maintained data is presented here as:

- ios_base::openmode mode, has in set if the input sequence can be read, and out set if the output sequence can be written.

### 27.7.1.1 basic_stringbuf constructors

[stringbuf.cons]

explicit basic_stringbuf(ios_base::openmode which =
                        ios_base::in | ios_base::out);

Effects: Constructs an object of class basic_stringbuf, initializing the base class with basic_streambuf() (27.5.2.1), and initializing mode with which.

Postcondition: str() == ".".

Draft
explicit basic_stringbuf(const basic_string<
charT, traits, Allocator>& str,
    ios_base::openmode which = ios_base::in | ios_base::out);

Effects: Constructs an object of class basic_stringbuf, initializing the base class with basic_streambuf() (27.5.2.1), and initializing mode with which. Then copies the content of str into the basic_stringbuf underlying character sequence. If which & ios_base::out is true, initializes the output sequence such that pbase() points to the first underlying character, epptr() points one past the last underlying character, and pptr() is equal to epptr() if which & ios_base::ate is true, otherwise pptr() is equal to pbase(). If which & ios_base::in is true, initializes the input sequence such that eback() and gptr() point to the first underlying character and egptr() points one past the last underlying character.

basic_stringbuf(basic_stringbuf&& rhs);

Effects: Move constructs from the rvalue rhs. It is implementation-defined whether the sequence pointers in *this (eback(), gptr(), egptr(), pbase(), pptr(), eptr()) obtain the values which rhs had. Whether they do or not, *this and rhs reference separate buffers (if any at all) after the construction. The openmode, locale and any other state of rhs is also copied.

Postconditions: Let rhs_p refer to the state of rhs just prior to this construction and let rhs_a refer to the state of rhs just after this construction.

- str() == rhs_p.str()
- gptr() - eback() == rhs_p.gptr() - rhs_p.eback()
- egptr() - eback() == rhs_p.egptr() - rhs_p.eback()
- pptr() - pbase() == rhs_p.pptr() - rhs_p.pbase()
- epptr() - pbase() == rhs_p.epptr() - rhs_p.pbase()
- if (eback()) eback() != rhs_a.eback()
- if (gptr()) gptr() != rhs_a.gptr()
- if (egptr()) egptr() != rhs_a.egptr()
- if (pbase()) pbase() != rhs_a.pbase()
- if (pptr()) pptr() != rhs_a.pptr()
- if (epptr()) epptr() != rhs_a.epptr()

27.7.1.2 Assign and swap

basic_stringbuf& operator=(basic_stringbuf&& rhs);

Effects: swap(rhs).

Returns: *this.

void swap(basic_stringbuf&& rhs);

Effects: Exchanges the state of *this and rhs.
template <class charT, class traits, class Allocator>
void swap(basic_stringbuf<charT, traits, Allocator>& x,
basic_stringbuf<charT, traits, Allocator>& y);

Effects: \( x . \text{swap}(y) \).

### 27.7.1.3 Member functions

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 \([\text{eback}(), \text{egptr}())\). 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 \([\text{pbase}(), \text{high_mark}()\), where \text{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 \text{str}(\text{basic_string}) member function. In the case of calling the \text{str}(\text{basic_string}) member function, all characters initialized prior to the call are now considered uninitialized (except for those characters re-initialized by the new basic_string). Otherwise the basic_stringbuf has been created in neither input nor output mode and a zero length basic_string is returned.

void str(const basic_string<charT,traits,Allocator>& s);

*Effects:* Copies the content of \( s \) into the basic_stringbuf underlying character sequence. If \( \text{mode} \) & ios_base::out is true, initializes the output sequence such that \text{pbase}() points to the first underlying character, \text{egptr}() points one past the last underlying character, and \text{pptr}() is equal to \text{eptr}() if \( \text{mode} \) & ios_base::in is true, otherwise \text{pptr}() is equal to \text{pbase}(). If \( \text{mode} \) & ios_base::in is true, initializes the input sequence such that \text{eback}() and \text{gptr}() point to the first underlying character and \text{egptr}() points one past the last underlying character.

### 27.7.1.4 Overridden virtual functions

int_type underflow();

*Returns:* If the input sequence has a read position available, returns traits::to_int_type(*\text{gptr}()). Otherwise, returns traits::eof(). Any character in the underlying buffer which has been initialized is considered to be part of the input sequence.

int_type pbackfail(int_type c = traits::eof());

*Effects:* Puts back the character designated by \( c \) to the input sequence, if possible, in one of three ways:
— If \texttt{traits::eq\_int\_type(c,traits::eof())} returns \texttt{false} and if the input sequence has a putback position available, and if \texttt{traits::eq(to\_char\_type(c),gptr()[-1])} returns \texttt{true}, assigns \texttt{gptr()} - 1 to \texttt{gptr()}.

Returns: \texttt{c}.

— If \texttt{traits::eq\_int\_type(c,traits::eof())} returns \texttt{false} and if the input sequence has a putback position available, and if \texttt{mode & ios\_base::out} is nonzero, assigns \texttt{c} to \texttt{*--gptr()}.

Returns: \texttt{c}.

— If \texttt{traits::eq\_int\_type(c,traits::eof())} returns \texttt{true} and if the input sequence has a putback position available, assigns \texttt{gptr()} - 1 to \texttt{gptr()}.

Returns: \texttt{traits::not\_eof(c)}.

Returns: \texttt{traits::eof()} to indicate failure.

Remarks: If the function can succeed in more than one of these ways, it is unspecified which way is chosen.

\begin{verbatim}
int\_type\_overflow(int\_type c = traits::eof());
\end{verbatim}

Effects: Appends the character designated by \texttt{c} to the output sequence, if possible, in one of two ways:

— If \texttt{traits::eq\_int\_type(c,traits::eof())} returns \texttt{false} and if either the output sequence has a write position available or the function makes a write position available (as described below), the function calls \texttt{sputc(c)}.

Signals success by returning \texttt{c}.

— If \texttt{traits::eq\_int\_type(c,traits::eof())} returns \texttt{true}, there is no character to append.

Signals success by returning a value other than \texttt{traits::eof()}.

Remarks: The function can alter the number of write positions available as a result of any call.

Returns: \texttt{traits::eof()} to indicate failure.

The function can make a write position available only if \texttt{(mode & ios\_base::out)} \texttt{!= 0}. To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements to hold the current array object (if any), plus at least one additional write position. If \texttt{(mode & ios\_base::in)} \texttt{!= 0}, the function alters the read end pointer \texttt{egptr()} to point just past the new write position.

\begin{verbatim}
pos\_type\_seekoff(off\_type off, ios\_base::seekdir way, 
   ios\_base::openmode which
   = ios\_base::in | ios\_base::out);
\end{verbatim}

Effects: Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 118.

For a sequence to be positioned, if its next pointer (either \texttt{gptr()} or \texttt{pptr()} ) is a null pointer and the new offset \texttt{newoff} is nonzero, the positioning operation fails. Otherwise, the function determines \texttt{newoff} as indicated in Table 119.

If \texttt{(newoff + off)} \texttt{< 0}, or if \texttt{newoff + off} refers to an uninitialized character (as defined in 27.7.1.3 paragraph 1), the positioning operation fails. Otherwise, the function assigns \texttt{xbeg + newoff + off} to the next
Table 118: `seekoff` positioning

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(which &amp; ios_base::in) == ios_base::in</code></td>
<td>positions the input sequence</td>
</tr>
<tr>
<td><code>(which &amp; ios_base::out) == ios_base::out</code></td>
<td>positions the output sequence</td>
</tr>
<tr>
<td>`(which &amp; (ios_base::in</td>
<td>ios_base::out)) == ios_base::beg or ios_base::end`</td>
</tr>
<tr>
<td>Otherwise</td>
<td>the positioning operation fails.</td>
</tr>
</tbody>
</table>

Table 119: `newoff` values

<table>
<thead>
<tr>
<th>Condition</th>
<th>newoff Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>way == ios_base::beg</code></td>
<td>0</td>
</tr>
<tr>
<td><code>way == ios_base::cur</code></td>
<td>the next pointer minus the beginning pointer (xnext - xbeg).</td>
</tr>
<tr>
<td><code>way == ios_base::end</code></td>
<td>the end pointer minus the beginning pointer (xend - xbeg).</td>
</tr>
</tbody>
</table>
27.7 String-based streams

Returns: pos_type(newoff), constructed from the resultant offset newoff (of type off_type), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is pos_type(off_type(-1)).

pos_type seekpos(pos_type sp, ios_base::openmode which = ios_base::in | ios_base::out);

Effect: Alters the stream position within the controlled sequences, if possible, to correspond to the stream position stored in sp (as described below).

— If (which & ios_base::in) != 0, positions the input sequence.
— If (which & ios_base::out) != 0, positions the output sequence.
— 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, seekpos, tellg, tellp) the effect is undefined.

Returns: sp to indicate success, or pos_type(off_type(-1)) to indicate failure.

basic_streambuf<charT,traits>* setbuf(charT* s, streamsize n);

Effect: implementation-defined, except that setbuf(0,0) has no effect.

Returns: this.

27.7.2 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;

            // 27.7.2.1 Constructors:
            explicit basic_istringstream(ios_base::openmode which = ios_base::in);
            explicit basic_istringstream(
                const basic_string<charT,traits,Allocator>& str,
                ios_base::openmode which = ios_base::in);
            basic_istringstream(basic_istringstream&& rhs);

            // 27.7.2.2 Assign and swap:
            basic_istringstream& operator=(basic_istringstream&& rhs);
            void swap(basic_istringstream&& rhs);

    } [istreamstream]

}
// 27.7.3 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;
};

//template <class charT, class traits, class Allocator>

void swap(basic_istringstream<charT, traits, Allocator>& x,
basic_istringstream<charT, traits, Allocator>& y);

//template <class charT, class traits, class Allocator>

void swap(basic_istringstream<charT, traits, Allocator>&& x,
basic_istringstream<charT, traits, Allocator>& y);

//template <class charT, class traits, class Allocator>

void swap(basic_istringstream<charT, traits, Allocator>& x,
basic_istringstream<charT, traits, Allocator>&& y);

The class basic_istringstream<charT, traits, Allocator> supports reading objects of class basic_string<charT, traits, Allocator>. It uses a basic_stringbuf<charT, traits, Allocator> object to control the associated storage. For the sake of exposition, the maintained data is presented here as:

— sb, the stringbuf object.

27.7.2.1 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)) (27.7.1.1).

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)) (27.7.1.1).

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.
27.7 String-based streams

27.7.2.2 Assign and swap

basic_istringstream& operator=(basic_istringstream&& rhs);

1 Effects: swap(rhs).
2 Returns: *this.

void swap(basic_istringstream&& rhs)

3 Effects: Exchanges the state of *this and rhs by calling basic_istream<charT, traits>::swap(rhs) and sb.swap(rhs.sb).

template <class charT, class traits, class Allocator>
void swap(basic_istringstream<charT, traits, Allocator>& x,
          basic_istringstream<charT, traits, Allocator>& y);

template <class charT, class traits, class Allocator>
void swap(basic_istringstream<charT, traits, Allocator>& x,
          basic_istringstream<charT, traits, Allocator>& y);

template <class charT, class traits, class Allocator>
void swap(basic_istringstream<charT, traits, Allocator>& x,
          basic_istringstream<charT, traits, Allocator>& y);

4 Effects: x.swap(y).

27.7.2.3 Member functions

basic_stringbuf<charT,traits,Allocator>* rdbuf() const;

1 Returns: const_cast<basic_stringbuf<charT,traits,Allocator>*(&sb).

basic_string<charT,traits,Allocator> str() const;
2 Returns: rdbuf()->str().

void str(const basic_string<charT,traits,Allocator>& s);
3 Effects: Calls rdbuf()->str(s).

27.7.3 Class basic_ostringstream

namespace std {

   template <class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT> >
   class basic_ostringstream : public basic_ostream<charT,traits> {

   public:

      // Types:
      typedef charT char_type;
      typedef typename traits::int_type int_type;
      typedef typename traits::pos_type pos_type;

Draft
typedef typename traits::off_type off_type;
typedef traits traits_type;
typedef Allocator allocator_type;

// 27.7.3.1 Constructors/destructor:
explicit basic_ostringstream(ios_base::openmode which = ios_base::out);
explicit basic_ostringstream(
    const basic_string<charT,traits,Allocator>& str,
    ios_base::openmode which = ios_base::out);
basic_ostringstream(basic_ostringstream&& rhs);

// 27.7.3.2 Assign/swap:
basic_ostringstream& operator=(basic_ostringstream&& rhs);
void swap(basic_ostringstream&& rhs);

// 27.7.3.3 Members:
basic_stringbuf<charT,traits,Allocator>* rdbuf() const;
basic_string<charT,traits,Allocator> str() const;
void str(const basic_string<charT,traits,Allocator>& s);
private:
    // basic_stringbuf<charT,traits,Allocator> sb; exposition only
};

template <class charT, class traits, class Allocator>
void swap(basic_ostringstream<charT, traits, Allocator>& x,
    basic_ostringstream<charT, traits, Allocator>& y);
template <class charT, class traits, class Allocator>
void swap(basic_ostringstream<charT, traits, Allocator>&& x,
    basic_ostringstream<charT, traits, Allocator>&& y);
template <class charT, class traits, class Allocator>
void swap(basic_ostringstream<charT, traits, Allocator>& x,
    basic_ostringstream<charT, traits, Allocator>&& y);

The class basic_ostringstream<charT, traits, Allocator> supports writing objects of class basic_string<charT, traits, Allocator>. It uses a basic_stringbuf object to control the associated storage. For the sake of exposition, the maintained data is presented here as:

— sb, the stringbuf object.

### 27.7.3.1 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) (27.7.1.1).
27.7 String-based streams

explicit basic_ostringstream(
    const basic_string<charT,traits,Allocator>& str,
    ios_base::openmode which = ios_base::out);

Effects: Constructs an object of class basic_ostringstream<charT, traits>, initializing the base class with basic_ostream<charT, traits>(&sb) and initializing sb with basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::out)) (27.7.1.1).

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.

27.7.3.2 Assign and swap [ostringstream.assign]

basic_ostringstream& operator=(basic_ostringstream&& rhs);

Effects: swap(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);

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).

27.7.3.3 Member functions [ostringstream.members]

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).
27.7.4 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/destructors
        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(basic_stringstream&& rhs);

        // 27.7.5.1 Assign/swap:
        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);
    template <class charT, class traits, class Allocator>
    void swap(basic_stringstream<charT, traits, Allocator>&& x,
              basic_stringstream<charT, traits, Allocator>& y);
    template <class charT, class traits, class Allocator>
    void swap(basic_stringstream<charT, traits, Allocator>&& x,
              basic_stringstream<charT, traits, Allocator>&& y);
}

1 The class template basic_stringstream<charT, traits> supports reading and writing from objects of class basic_1.
27.7 String-based streams

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

— `sb`, the stringbuf object.

### 27.7.5 basic_stringstream constructors

[stringstream.cons]

```cpp
explicit basic_stringstream(
    ios_base::openmode which = ios_base::out|ios_base::in);
```

*Effects:* Constructs an object of class `basic_stringstream<charT, traits>`, initializing the base class with `basic_iostream(&sb)` and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(which)`.

```cpp
explicit basic_stringstream(
    const basic_string<charT, traits, Allocator>& str,
    ios_base::openmode which = ios_base::out|ios_base::in);
```

*Effects:* Constructs an object of class `basic_stringstream<charT, traits>`, initializing the base class with `basic_iostream(&sb)` and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which)`.

```cpp
basic_stringstream(basic_stringstream&& rhs);
```

*Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_stringbuf`. Next `basic_iostream<charT, traits>::set_rdbuf(&sb)` is called to install the contained `basic_stringbuf`.

### 27.7.5.1 Assign and swap

[stringstream.assign]

```cpp
basic_stringstream& operator=(basic_stringstream&& rhs);
```

*Effects:* `swap(rhs)`.

*Returns:* `*this`.

```cpp
void swap(basic_stringstream&& rhs);
```

*Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_iostream<charT, traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```cpp
template <class charT, class traits, class Allocator>
void swap(basic_stringstream<charT, traits, Allocator>& x,
         basic_stringstream<charT, traits, Allocator>& y);
template <class charT, class traits, class Allocator>
void swap(basic_stringstream<charT, traits, Allocator>&& x,
          basic_stringstream<charT, traits, Allocator>&& y);
template <class charT, class traits, class Allocator>
void swap(basic_stringstream<charT, traits, Allocator>& x,
          basic_stringstream<charT, traits, Allocator>&& y);
template <class charT, class traits, class Allocator>
void swap(basic_stringstream<charT, traits, Allocator>&& x,
          basic_stringstream<charT, traits, Allocator>&& y);
```

*Effects:* `x.swap(y)`.

Draft
27.7.6 Member functions

basic_stringbuf<charT,traits,Allocator>* rdbuf() const;

1. **Returns:** const_cast<basic_stringbuf<charT,traits,Allocator>*>(&sb)

basic_string<charT,traits,Allocator> str() const;

2. **Returns:** rdbuf()->str().

void str(const basic_string<charT,traits,Allocator>& str);

3. **Effects:** Calls rdbuf()->str(str).

27.8 File-based streams

27.8.1 File streams

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**

```cpp
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;
}
```

In this subclause, the type name `FILE` refers to the type `FILE` declared in `<cstdio>` (27.8.2).\(^{328}\)

**File** A File provides an external source/sink stream whose *underlaid character type* is `char` (byte).\(^{329}\)

\(^{328}\) In C, `FILE` must be a typedef. In C++, it may be a typedef or other type name.

\(^{329}\) A File is a sequence of multibyte characters. In order to provide the contents as a wide character sequence, `filebuf` should convert between wide character sequences and multibyte character sequences.
27.8 File-based streams

Multibyte character and Files A File provides byte sequences. So the streambuf (or its derived classes) treats a file as the external source/sink byte sequence. In a large character set environment, multibyte character sequences are held in files. In order to provide the contents of a file as wide character sequences, wide-oriented filebuf, namely wfilebuf should convert wide character sequences.

27.8.1.1 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;

      // 27.8.1.2 Constructors/destructor:
      basic_filebuf();
      basic_filebuf(basic_filebuf&& rhs);
      virtual ~basic_filebuf();

      // 27.8.1.3 Assign/swap:
      basic_filebuf& operator=(basic_filebuf&& rhs);
      void swap(basic_filebuf&& rhs);

      // 27.8.1.4 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);
      basic_filebuf<charT,traits>* close();

      protected:
        // 27.8.1.5 Overridden virtual functions:
        virtual streamsize showmanyc();
        virtual int_type underflow();
        virtual int_type uflow();
        virtual int_type pbackfail(int_type c = traits::eof());
        virtual int_type overflow (int_type c = traits::eof());

        virtual basic_streambuf<charT,traits>*
          setbuf(char_type* s, streamsize n);
        virtual pos_type seekoff(off_type off, ios_base::seekdir way,
          ios_base::openmode which = ios_base::in | ios_base::out);
        virtual pos_type seekpos(pos_type sp,
          ios_base::openmode which = ios_base::in | ios_base::out);
        virtual int sync();
  } // class basic_filebuf
} // namespace std
virtual void imbue(const locale& loc);
};

template <class charT, class traits>
void swap(basic_filebuf<charT, traits>& x,
          basic_filebuf<charT, traits>& y);

template <class charT, class traits>
void swap(basic_filebuf<charT, traits>&& x,
          basic_filebuf<charT, traits>& y);

template <class charT, class traits>
void swap(basic_filebuf<charT, traits>& x,
          basic_filebuf<charT, traits>&& y);

The class basic_filebuf<charT, traits> associates both the input sequence and the output sequence with a file.

The restrictions on reading and writing a sequence controlled by an object of class basic_filebuf<charT, traits> are the same as for reading and writing with the Standard C library FILEs.

In particular:

— If the file is not open for reading the input sequence cannot be read.
— If the file is not open for writing the output sequence cannot be written.
— A joint file position is maintained for both the input sequence and the output sequence.

An instance of basic_filebuf behaves as described in 27.8.1.1 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());
```

27.8.1.2 basic_filebuf constructors

basic_filebuf();

Effects: Constructs an object of class basic_filebuf<charT,traits>, initializing the base class with basic_streambuf<charT,traits>() (27.5.2.1).

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.

- is_open() == rhs_p.is_open()
- rhs_a.is_open() == false
- gptr() - eback() == rhs_p.gptr() - rhs_p.eback()
- egptr() - eback() == rhs_p.egptr() - rhs_p.eback()
- pptr() - pbase() == rhs_p.pptr() - rhs_p.pbase()
- epptr() - pbase() == rhs_p.epptr() - rhs_p.pbase()
- if (eback()) eback() != rhs_a.eback()
- if (gptr()) gptr() != rhs_a.gptr()
- if (egptr()) egptr() != rhs_a.egptr()
- if (pbase()) pbase() != rhs_a.pbase()
- if (pptr()) pptr() != rhs_a.pptr()
- if (epptr()) epptr() != rhs_a.epptr()

virtual ~basic_filebuf();

Effects: Destroys an object of class basic_filebuf<charT,traits>. Calls close().

27.8.1.3 Assign and swap

basic_filebuf& operator=(basic_filebuf&& rhs);

Effects: swap(rhs).

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);

template <class charT, class traits>
void swap(basic_filebuf<charT, traits>&& x,
          basic_filebuf<charT, traits>&& y);

template <class charT, class traits>
void swap(basic_filebuf<charT, traits>& x,
          basic_filebuf<charT, traits>&& y);

template <class charT, class traits>
void swap(basic_filebuf<charT, traits>&& x,
          basic_filebuf<charT, traits>&& y);

Effects: x.swap(y).
27.8.1.4 Member functions

```cpp
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.

```cpp
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 120. If mode is not some combination of flags shown in the table then the open fails.

<table>
<thead>
<tr>
<th>ios_base Flag combination</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>in</td>
</tr>
<tr>
<td>+</td>
<td>&quot;w&quot;</td>
</tr>
<tr>
<td>+</td>
<td>&quot;a&quot;</td>
</tr>
<tr>
<td>+</td>
<td>&quot;w&quot;</td>
</tr>
<tr>
<td>+</td>
<td>&quot;x&quot;</td>
</tr>
<tr>
<td>+</td>
<td>&quot;x+&quot;</td>
</tr>
<tr>
<td>+</td>
<td>&quot;w+b&quot;</td>
</tr>
<tr>
<td>+</td>
<td>&quot;ab&quot;</td>
</tr>
<tr>
<td>+</td>
<td>&quot;wb&quot;</td>
</tr>
<tr>
<td>+</td>
<td>&quot;r&quot;</td>
</tr>
<tr>
<td>+</td>
<td>&quot;r+b&quot;</td>
</tr>
<tr>
<td>+</td>
<td>&quot;w+b&quot;</td>
</tr>
</tbody>
</table>

3 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)).

4 If the repositioning operation fails, calls close() and returns a null pointer to indicate failure.

```cpp
basic_filebuf<charT,traits>* open(const string& s, ios_base::openmode mode);
```

*Returns:* open(s.c_str(), mode);

```cpp
basic_filebuf<charT,traits>* close();
```

Draft

---

3 The macro SEEK_END is defined, and the function signatures fopen(const char*, const char*) and fseek(FILE*, long, int) are declared in <cstdio> (27.8.2).
Effects: If _is_open()_ == false, returns a null pointer. If a put area exists, calls _overflow(traits::eof())_ to flush characters. If the last virtual member function called on this (between underflow, overflow, seekoff, and seekpos) was overflow then calls _a_codecvt.unshift_(possibly several times) to determine a termination sequence, inserts those characters and calls _overflow(traits::eof())_ again. Finally it closes the file (“as if” by calling _std::fclose(file)_). If any of the calls to overflow or _std::fclose_ fails then close fails.

Returns: this on success, a null pointer otherwise.

Postcondition: _is_open()_ == false.

### 27.8.1.5 Overridden virtual functions

**streamsize showmanyc();**

Effects: Behaves the same as _basic_streambuf::showmanyc_ (27.5.2.4).

Remarks: An implementation might well provide an overriding definition for this function signature if it can determine that more characters can be read from the input sequence.

**int_type underflow();**

Effects: Behaves according to the description of _basic_streambuf<charT,traits>::underflow_(), with the specialization that a sequence of characters is read from the input sequence “as if” by reading from the associated file into an internal buffer (extern_buf) and then “as if” doing

```c
char extern_buf[XSIZE];
char* extern_end;
charT intern_buf[ISIZE];
charT* intern_end;
codecvt_base::result r =
a_codecvt.in(state, extern_buf, extern_buf+XSIZE, extern_end,
           intern_buf, intern_buf+ISIZE, intern_end);
```

This shall be done in such a way that the class can recover the position (_fpos_t_) corresponding to each character between _intern_buf_ and _intern_end_. If the value of _r_ indicates that _a_codecvt.in_() ran out of space in _intern_buf_, retry with a larger _intern_buf_.

**int_type uflow();**

Effects: Behaves according to the description of _basic_streambuf<charT,traits>::uflow_(), with the specialization that a sequence of characters is read from the input with the same method as used by _underflow_.

**int_type pbackfail(int_type c = traits::eof());**

Effects: Puts back the character designated by _c_ to the input sequence, if possible, in one of three ways:

- If _traits::eq_int_type(c,traits::eof())_ returns false and if the function makes a putback position available and if _traits::eq(to_char_type(c),gptr()[-1])_ returns true, decrements the next pointer for the input sequence, gptr().

---

331) The function signature _fclose(FILE*)_ is declared in <cstdio> (27.8.2).

Draft
Returns: $c$.

— If $\text{traits::eq_int_type}(c, \text{traits::eof})$ returns false and if the function makes a putback position available and if the function is permitted to assign to the putback position, decrements the next pointer for the input sequence, and stores $c$ there.

Returns: $c$.

— If $\text{traits::eq_int_type}(c, \text{traits::eof})$ returns true, and if either the input sequence has a putback position available or the function makes a putback position available, decrements the next pointer for the input sequence, $\text{gptr()}$.

Returns: $\text{traits::not_eof}(c)$.

6 Returns: $\text{traits::eof()}$ to indicate failure.

7 Remarks: If $\text{is_open()} == \text{false}$, 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.

```c
int_type overflow(int_type c = traits::eof());
```

Effects: Behaves according to the description of `basic_streambuf<charT,traits>::overflow(c)`, except that the behavior of “consuming characters” is performed by first converting “as if” by:

```c
char* b = pbase();
char* p = pptr();
char* end;
char xbuf[XSIZE];
char* xbuf_end;
codecvt_base::result r =
    a_codecvt.out(state, b, p, end, xbuf, xbuf+XSIZE, xbuf_end);
```

and then

— If $r ==$ codecvt_base::error then fail.

— If $r ==$ codecvt_base::noconv then output characters from b up to (and not including) p.

— If $r ==$ codecvt_base::partial then output to the file characters from xbuf up to xbuf_end, and repeat using characters from end to p. If output fails, fail (without repeating).

— Otherwise output from xbuf to xbuf_end, and fail if output fails. At this point if b != p and b == end (xbuf isn’t large enough) then increase XSIZE and repeat from the beginning.

11 Returns: $\text{traits::not_eof}(c)$ to indicate success, and $\text{traits::eof()}$ to indicate failure. If $\text{is_open()} == \text{false}$, the function always fails.

```c
basic_streambuf* setbuf(char_type* s, streamsize n);
```
Effects: If setbuf(0,0) is called on a stream before any I/O has occurred on that stream, the stream becomes unbuffered. Otherwise the results are implementation-defined. “Unbuffered” means that pbase() and pptr() always return null and output to the file should appear as soon as possible.

pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode which = ios_base::in | ios_base::out);

Effects: Let width denote a_codecvt.encoding(). If is_open() == false, or off != 0 && width <= 0, then the positioning operation fails. Otherwise, if way != basic_ios::cur or off != 0, and if the last operation was output, then update the output sequence and write any unshift sequence. Next, seek to the new position: if width > 0, call 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 121.

Table 121: seekoff effects

<table>
<thead>
<tr>
<th>way</th>
<th>Value</th>
<th>stdio Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic_ios::beg</td>
<td>SEEK_SET</td>
<td></td>
</tr>
<tr>
<td>basic_ios::cur</td>
<td>SEEK_CUR</td>
<td></td>
</tr>
<tr>
<td>basic_ios::end</td>
<td>SEEK_END</td>
<td></td>
</tr>
</tbody>
</table>

Returns: a newly constructed pos_type object that stores the resultant stream position, if possible. If the positioning operation fails, or if the object cannot represent the resultant stream position, returns pos_type(off_type(-1)).

pos_type seekpos(pos_type sp, ios_base::openmode which = ios_base::in | ios_base::out);

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.

```cpp
void imbue(const locale& loc);
```

Precondition: 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 (22.2.1.4.2) 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.

### 27.8.1.6 Class template `basic_ifstream`

```cpp
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;

            // 27.8.1.7 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(basic_ifstream&& rhs);

            // 27.8.1.8 Assign/swap:
            basic_ifstream& operator=(basic_ifstream&& rhs);
            void swap(basic_ifstream&& rhs);

            // 27.8.1.9 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
        };
    }
}
```
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.

### 27.8.1.7 `basic_ifstream` constructors

#### `basic_ifstream()`

```cpp
basic_ifstream();
```

**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>()` (27.6.1.1.1, 27.8.1.2).

#### `explicit basic_ifstream(const char* s, ios_base::openmode mode = ios_base::in)`

```cpp
explicit basic_ifstream(const char* s, 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>()` (27.6.1.1.1, 27.8.1.2), 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)`

```cpp
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)`

```cpp
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`.

### 27.8.1.8 Assign and swap

#### `basic_ifstream& operator=(basic_ifstream&& rhs)`

```cpp
basic_ifstream& operator=(basic_ifstream&& rhs);
```

**Effects:** `swap(rhs)`.

**Returns:** `*this`.

```cpp
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).

```cpp
template <class charT, class traits, class Allocator>
void swap(basic_ifstream<charT, traits, Allocator>& x,
          basic_ifstream<charT, traits, Allocator>& y);
template <class charT, class traits, class Allocator>
void swap(basic_ifstream<charT, traits, Allocator>&& x,
          basic_ifstream<charT, traits, Allocator>& y);
template <class charT, class traits, class Allocator>
void swap(basic_ifstream<charT, traits, Allocator>& x,
          basic_ifstream<charT, traits, Allocator>&& y);
```

Effects: x.swap(y).

### 27.8.1.9 Member functions

```cpp
basic_filebuf<charT, traits>* rdbuf() const;

bool is_open() const;

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 (27.4.4.3)). A successful open does not change the error state.

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 false, calls setstate(failbit) (which may throw ios_base::failure (27.4.4.3)).

### 27.8.1.10 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;
```
27.8 File-based streams

```cpp
typedef typename traits::off_type off_type;
typedef traits traits_type;

// 27.8.1.11 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(basic_ofstream&& rhs);

// 27.8.1.12 Assign/swap:
basic_ofstream& operator=(basic_ofstream&& rhs);
void swap(basic_ofstream&& rhs);

// 27.8.1.13 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
};
```

The class `basic_ofstream<charT, traits>` supports writing to named files. It uses a `basic_filebuf<charT, traits>` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

— `sb`, the filebuf object.

27.8.1.11 `basic_ofstream` constructors

```cpp
basic_ofstream();
```

**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>()` (27.6.2.2, 27.8.1.2).
explicit basic_ofstream(const char* s,
  ios_base::openmode mode = ios_base::out);

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>()) (27.6.2.2, 27.8.1.2), then calls rdbuf() -> open(s, mode | ios_base::out). If that function returns a null pointer, calls setstate(fail-bit).

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.

27.8.1.12 Assign and swap

basic_ofstream& operator=(basic_ofstream&& rhs);

Effects: swap(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, class Allocator>
void swap(basic_ofstream<charT, traits, Allocator>& x,
  basic_ofstream<charT, traits, Allocator>& y);
template <class charT, class traits, class Allocator>
void swap(basic_ofstream<charT, traits, Allocator>&& x,
  basic_ofstream<charT, traits, Allocator>& y);
template <class charT, class traits, class Allocator>
void swap(basic_ofstream<charT, traits, Allocator>& x,
  basic_ofstream<charT, traits, Allocator>&& y);

Effects: x.swap(y).

27.8.1.13 Member functions

basic_filebuf<charT,traits>* rdbuf() const;

Returns: const_cast<basic_filebuf<charT,traits>*(&sb).

bool is_open() const;

Draft
Returns: rdbuf()->is_open().

```cpp
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 (27.4.4.3)). A successful open does not change the error state.

```cpp
void close();
```

Effects: Calls rdbuf()->close() and, if that function fails (returns a null pointer), calls setstate(failbit) (which may throw ios_base::failure (27.4.4.3)).

```cpp
void open(const string& s, ios_base::openmode mode = ios_base::out);
```

Effects: calls open(s.c_str(), mode);

27.8.1.14 Class template basic_fstream

```cpp
namespace std {
    template <class charT, class traits=char_traits<charT> >
    class basic_fstream
        : public basic_iostream<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(basic_fstream&& rhs);

        // 27.8.1.16 Assign/swap:
        basic_fstream& operator=(basic_fstream&& rhs);
        void swap(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,
```
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.

### 27.8.1.15 `basic_fstream` constructors

#### `basic_fstream()`

Effects: Constructs an object of class `basic_fstream<charT, traits>`, initializing the base class with `basic_istream(&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_iostream(&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`. 

Draft
27.8.1.16 Assign and swap

basic_fstream& operator=(basic_fstream&& rhs);

Effects: swap(rhs).
Returns: *this.

void swap(basic_fstream& rhs);

Effects: Exchanges the state of *this and rhs by calling basic_iostream<charT,traits>::swap(rhs) and sb.swap(rhs.sb).

template <class charT, class traits, class Allocator> void swap(basic_fstream<charT, traits, Allocator>& x, basic_fstream<charT, traits, Allocator>& y);
template <class charT, class traits, class Allocator> void swap(basic_fstream<charT, traits, Allocator>&& x, basic_fstream<charT, traits, Allocator>& y);
template <class charT, class traits, class Allocator> void swap(basic_fstream<charT, traits, Allocator>& x, basic_fstream<charT, traits, Allocator>&& y);

Effects: x.swap(y).

27.8.1.17 Member functions

basic_filebuf<charT,traits>* rdbuf() const;

Returns: const_cast<basic_filebuf<charT,traits>*>(sb).

bool is_open() const;

Effects: 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)(27.4.4.3).

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 false, calls setstate(failbit)(27.4.4.3) (which may throw ios_base::failure).
27.8.2 C Library files

Table 122 describes header `<cstdio>`.

Table 122: Header `<cstdio>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td></td>
</tr>
<tr>
<td>EOF</td>
<td>L_tmpnam</td>
</tr>
<tr>
<td>FILENAME_MAX</td>
<td>NULL</td>
</tr>
<tr>
<td>Types:</td>
<td>FILE</td>
</tr>
<tr>
<td>Functions:</td>
<td>clearerr</td>
</tr>
<tr>
<td>fclose</td>
<td>fprintf</td>
</tr>
<tr>
<td>feof</td>
<td>fputc</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>fgets</td>
<td>fseek</td>
</tr>
</tbody>
</table>

See also: ISO C subclause 7.9, Amendment 1 subclause 4.6.2.

Table 123 describes header `<cinttypes>`.

Table 123: Header `<cinttypes>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PRI{d i o u x}{MAX PTR}</td>
</tr>
<tr>
<td></td>
<td>SCN{d i o u x}{FAST LEAST}{8 16 32 64}</td>
</tr>
<tr>
<td></td>
<td>SCN{d i o u x}{MAX PTR}</td>
</tr>
<tr>
<td>Types:</td>
<td>imaxdiv_t</td>
</tr>
<tr>
<td>Functions:</td>
<td>abs</td>
</tr>
<tr>
<td></td>
<td>div</td>
</tr>
</tbody>
</table>

Draft
Chapter 28  Regular expressions library

This clause describes components that C++ programs may use to perform operations involving regular expression matching and searching.

28.1  Definitions

The following definitions shall apply to this clause:

28.1.1  collating element
        [defns.regex.collating.element]
        a sequence of one or more characters within the current locale that collate as if they were a single character.

28.1.2  finite state machine
        [defns.regex.finite.state.machine]
        an unspecified data structure that is used to represent a regular expression, and which permits efficient matches against
        the regular expression to be obtained.

28.1.3  format specifier
        [defns.regex.format.specifier]
        a sequence of one or more characters that is to be replaced with some part of a regular expression match.

28.1.4  matched
        [defns.regex.matched]
        a sequence of zero or more characters is matched by a regular expression when the characters in the sequence correspond
        to a sequence of characters defined by the pattern.

28.1.5  primary equivalence class
        [defns.regex.primary.equivalence.class]
        a set of one or more characters which share the same primary sort key: that is the sort key weighting that depends only
        upon character shape, and not accentation, case, or locale specific tailorings.
regular expression
a pattern that selects specific strings from a set of character strings.

28.1.7
sub-expression
a subset of a regular expression that has been marked by parenthesis.

28.2 Requirements

1 This subclause defines requirements on classes representing regular expression traits. [Note: The class template regex_traits, defined in clause 28.7, satisfies these requirements. — end note]

2 The class template 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 traits used by the basic_regex class template. This subclause defines the semantics guaranteed by these members.

3 To specialize class template basic_regex for a character container CharT and its related regular expression traits class Traits, use basic_regex<CharT, Traits>.

4 In Table 124 X denotes a traits class defining types and functions for the character container type charT; u is an object of type X; v is an object of type const X; p is a value of type const charT*; I1 and I2 are Input Iterators; F1 and F2 are forward iterators; c is a value of type const charT; s is an object of type X::string_type; cs is an object of type const X::string_type; b is a value of type bool; I is a value of type int; cl is an object of type X::char_class_type, and loc is an object of type X::locale_type.

<table>
<thead>
<tr>
<th>expression</th>
<th>Return Type</th>
<th>Assertion / Note / Pre / Post condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::char_type</td>
<td>charT</td>
<td>The character container type used in the implementation of class template basic_regex.</td>
</tr>
<tr>
<td>X::string_type</td>
<td>std::basic_string&lt;CharT&gt;</td>
<td></td>
</tr>
<tr>
<td>X::locale_type</td>
<td>A copy constructible type</td>
<td>A type that represents the locale used by the traits class.</td>
</tr>
<tr>
<td>X::char_class_type</td>
<td>A bitmask type [17.3.2.1.2].</td>
<td>A bitmask type representing a particular character classification.</td>
</tr>
<tr>
<td>X::length(p)</td>
<td>std::size_t</td>
<td>Yields the smallest i such that p[i] == 0. Complexity is linear in i.</td>
</tr>
<tr>
<td>v.translate(c)</td>
<td>X::char_type</td>
<td>Returns a character such that for any character d that is to be considered equivalent to c then v.translate(c) == v.translate(d).</td>
</tr>
<tr>
<td>v.translate_nocase(c)</td>
<td>X::char_type</td>
<td>For all characters C that are to be considered equivalent to c when comparisons are to be performed without regard to case, then v.translate_nocase(c) == v.translate_nocase(C).</td>
</tr>
</tbody>
</table>

Table 124: 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>v.transform(F1, F2)</td>
<td>X::string_type</td>
<td>Returns a sort key for the character sequence designated by the iterator range [F1,F2) such that if the character sequence [G1,G2,s) sorts before the character sequence [H1,H2,t) then v.transform(G1, G2) &lt; v.transform(H1, H2).</td>
</tr>
<tr>
<td>v.transform_primary(F1, F2)</td>
<td>X::string_type</td>
<td>Returns a sort key for the character sequence designated by the iterator range [F1,F2) such that if the character sequence [G1,G2) sorts before the character sequence [H1,H2) when character case is not considered then v.transform_primary(G1, G2) &lt; v.transform_primary(H1, H2).</td>
</tr>
<tr>
<td>v.lookup_collatename(F1, F2)</td>
<td>X::string_type</td>
<td>Returns a sequence of characters that represents the collating element consisting of the character sequence designated by the iterator range [F1,F2). Returns an empty string if the character sequence is not a valid collating element.</td>
</tr>
<tr>
<td>v.lookup_classname(F1, F2)</td>
<td>X::char_class_type</td>
<td>Converts the character sequence designated by the iterator range [F1,F2) into a value of a bitmask type that can subsequently be passed to isctype. Values returned from lookup_classname can be bitwise or’ed together; the resulting value represents membership in either of the corresponding character classes. Returns 0 if the character sequence is not the name of a character class recognized by X. The value returned shall be independent of the case of the characters in the sequence.</td>
</tr>
<tr>
<td>v.isctype(c, cl)</td>
<td>bool</td>
<td>Returns true if character c is a member of one of the character classes designated by cl, false otherwise.</td>
</tr>
<tr>
<td>v.value(c, I)</td>
<td>int</td>
<td>Returns the value represented by the digit c in base I if the character c is a valid digit in base I; otherwise returns -1. [Note: the value of I will only be 8, 10, or 16. — end note]</td>
</tr>
<tr>
<td>u.imbue(loc)</td>
<td>X::locale_type</td>
<td>Imbues u with the locale loc and returns the previous locale used by u if any.</td>
</tr>
<tr>
<td>v.getloc()</td>
<td>X::locale_type</td>
<td>Returns the current locale used by v, if any.</td>
</tr>
</tbody>
</table>

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.]

Draft
28.3 Regular expressions summary

The header `<regex>` defines a basic regular expression class template and its traits that can handle all char-like template arguments ([21]).

The header `<regex>` defines a class template that holds the result of a regular expression match.

The header `<regex>` defines a series of algorithms that allow an iterator sequence to be operated upon by a regular expression.

The header `<regex>` defines two specific template classes, `regex` and `wregex`, and their special traits.

The header `<regex>` also defines two iterator types for enumerating regular expression matches.

28.4 Header `<regex>` synopsis

```cpp
namespace std {
    // 28.5, regex constants:
    namespace regex_constants {
        typedef bitmask_type syntax_option_type;
        typedef bitmask_type match_flag_type;
        typedef implementation-defined 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<const_iterator> ssub_match;
    typedef sub_match<wstring::const_iterator> wssub_match;

    // 28.9.2, sub_match non-member operators:

    Draft
```
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);

Draft
template <class BiIter, class ST, class SA>
bool operator!=(
    const sub_match<BiIter>& lhs,
    const basic_string<
        typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template <class BiIter, class ST, class SA>
bool operator<(const sub_match<BiIter>& lhs, const basic_string<
    typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template <class BiIter, class ST, class SA>
bool operator>(const sub_match<BiIter>& lhs, const basic_string<
    typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template <class BiIter, class ST, class SA>
bool operator>=(const sub_match<BiIter>& lhs, const basic_string<
    typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template <class BiIter, class ST, class SA>
bool operator<=(const sub_match<BiIter>& lhs, const basic_string<
    typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template <class BiIter>
bool operator==(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<=(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);

Draft
typedef 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==(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<=(typename iterator_traits<BiIter>::value_type const& lhs,
                const sub_match<BiIter>& rhs);

Draft
28.4 Header `<regex>` synopsis

```cpp

typedef 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);

// 28.10.6, 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<
```
typedef basic_string<charT, ST, SA>::const_iterator,
    Allocator>& m,
const basic_regex<charT, traits>& e,
regex_constants::match_flag_type flags =
regex_constants::match_default);

template <class charT, class 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,
28.5 Namespace std::regex_constants

const basic_regex<charT, traits>& e,
regex_constants::match_flag_type flags =
regex_constants::match_default);

// 28.11.4, function template regex_replace:
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 basic_string<charT>& fmt,
regex_constants::match_flag_type flags =
regex_constants::match_default);

template <class traits, class charT>
basic_string<charT>
regex_replace(const basic_string<charT>& s,
const basic_regex<charT, traits>& e,
const basic_string<charT>& 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:
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;
} // namespace std

28.5 Namespace std::regex_constants

The namespace std::regex_constants holds symbolic constants used by the regular expression library. This name-
space 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 regex_constants {
  typedef bitmask_type syntax_option_type;
  static const syntax_option_type icase;
  static const syntax_option_type nosubs;
  static const syntax_option_type optimize;
  static const syntax_option_type collate;
  static const syntax_option_type ECMAScript;
  static const syntax_option_type basic;
  static const syntax_option_type extended;
  static const syntax_option_type awk;
  static const syntax_option_type grep;
  static const syntax_option_type egrep;
} // namespace regex_constants
```

The type `syntax_option_type` is an implementation defined bitmask type ([17.3.2.1.2]). Setting its elements has the effects listed in table 125. A valid value of type `syntax_option_type` shall have exactly one of the elements `ECMAScript`, `basic`, `extended`, `awk`, `grep`, `egrep`, set.

### 28.5.2 Bitmask Type `regex_constants::match_flag_type`

```cpp
namespace regex_constants {
  typedef bitmask_type regex_constants::match_flag_type;

  static const match_flag_type match_default = 0;
  static const match_flag_type match_not_bol;
  static const match_flag_type match_not_eol;
  static const match_flag_type match_not_bow;
  static const match_flag_type match_not_eow;
  static const match_flag_type match_any;
  static const match_flag_type match_not_null;
  static const match_flag_type match_continuous;
  static const match_flag_type match_prev_avail;
  static const match_flag_type format_default = 0;
  static const match_flag_type format_sed;
  static const match_flag_type format_no_copy;
  static const match_flag_type format_first_only;
} // namespace regex_constants
```

The type `regex_constants::match_flag_type` is an implementation defined bitmask type ([17.3.2.1.2]). Matching a regular expression against a sequence of characters `[first,last)` proceeds according to the rules of the grammar specified for the regular expression object, modified according to the effects listed in table 126 for any bitmask elements set.

<table>
<thead>
<tr>
<th>Table 126: <code>regex_constants::match_flag_type</code> effects when obtaining a match against a character container sequence <code>[first,last)</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Draft</td>
</tr>
</tbody>
</table>
### 28.5 Namespace `std::regex_constants`

<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 <code>[first, last)</code> shall be treated as though it is not at the beginning of a line, so the character <code>^</code> in the regular expression shall not match <code>[first, first)</code>.</td>
</tr>
<tr>
<td>match_not_eol</td>
<td>The last character in the sequence <code>[first, last)</code> shall be treated as though it is not at the end of a line, so the character <code>&quot;$&quot;</code> in the regular expression shall not match <code>[last, last)</code>.</td>
</tr>
<tr>
<td>match_not_bow</td>
<td>The expression <code>\b</code> shall not match the sub-sequence <code>[first, first)</code>.</td>
</tr>
<tr>
<td>match_not_eow</td>
<td>The expression <code>\b</code> shall not match the sub-sequence <code>[last, last)</code>.</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 <code>first</code>.</td>
</tr>
<tr>
<td>match_prev_avail</td>
<td><code>--first</code> is a valid iterator position. When this flag is set the flags match_not_bol and match_not_bow shall be ignored by the regular expression algorithms 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 ECMA/Script replace function in ECMA-262, part 15.4.11 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 regex_constants {
    typedef implementation defined error_type;

    static const error_type error_collate;
    static const error_type error ctype;
    static const error_type error escape;
    static const error_type error backref;
    static const error_type error brack;
    static const error_type error paren;
    static const error_type error brace;
    static const error_type error badbrace;
    static const error_type error range;
    static const error_type error space;
    static const error_type error badrepeat;
```

Draft
<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 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>
28.6 Class `regex_error`  

```cpp
static const error_type error_complexity;
static const error_type error_stack;
} // namespace regex_constants
```

1. The type `error_type` is an implementation defined enumeration type ([[17.3.1.1]]). Values of type `error_type` represent the error conditions described in table 127:

Table 127: `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;
```
Returns: The error code that was passed to the constructor.

28.7 Class template regex_traits

```cpp
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);
        charT translate(charT c) const;
        charT translate_nocase(charT c) const;
        template <class ForwardIterator>
        string_type transform(ForwardIterator first, ForwardIterator last) const;
        template <class ForwardIterator>
        string_type transform_primary(ForwardIterator first, ForwardIterator last) const;
        template <class ForwardIterator>
        string_type lookup_collatename(ForwardIterator first, ForwardIterator last) const;
        template <class ForwardIterator>
        char_class_type lookup_classname(ForwardIterator first, ForwardIterator last) const;
        bool isctype(charT c, char_class_type f) const;
        int value(charT ch, int radix) const;
        locale_type imbue(locale_type l);
        locale_type getloc() const;
};
```

The specializations regex_traits<char> and regex_traits<wchar_t> shall be valid and shall satisfy the requirements for a regular expression traits class (28.2).

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<char>::length(p);

charT translate(charT c) const;

Returns: (c).

charT translate_nocase(charT c) const;

Draft
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.end());

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) const;

Returns: an unspecified value that represents the character classification named by the character sequence designated by the iterator range [first,last). The value returned shall be independent of the case of the characters in the character sequence. If the name is not recognized then returns a value that compares equal to 0.

Remarks: For regex_traits<char>, at least the names "d", "w", "s", "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper" and "xdigit" shall be recognized. For regex_traits<wchar_t>, at least the names L"d", L"w", L"s", L"alnum", L"alpha", L"blank", L"cntrl", L"digit", L"graph", L"lower", L"print", L"punct", L"space", L"upper" and L"xdigit" 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: Converts f into a value m of type std::ctype_base::mask in an unspecified manner, and returns true if use_facet<ctype<charT> >(getloc()).is(c, m) is true. Otherwise returns true if f bitwise or’ed with the result of calling lookup_classname with an iterator pair that designates the character sequence "w" is not equal to 0 and c == ' ', or if f bitwise or’ed with the result of calling lookup_classname with an iterator pair that designates the character sequence "blank" is not equal to 0 and c is one of an implementation-defined subset of the characters for which isspace(c, getloc()) returns true, otherwise returns false.

int value(charT ch, int radix) const;
Precondition: The value of \( \text{radix} \) shall be 8, 10, or 16.

Returns: the value represented by the digit \( \text{ch} \) in base \( \text{radix} \) if the character \( \text{ch} \) is a valid digit in base \( \text{radix} \); otherwise returns -1.

```cpp
locale_type imbue(locale_type loc);
```

Effects: Imbues this with a copy of the locale \( \text{loc} \). \[Note: calling imbue with a different locale than the one currently in use invalidates all cached data held by \*this. \text{— end note}\]

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.

```cpp
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.

### 28.8 Class template `basic_regex`

For a char-like type \( \text{charT} \), specializations of class template `basic_regex` represent regular expressions constructed from character sequences of \( \text{charT} \) characters. In the rest of 28.8, \( \text{charT} \) 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 \( \text{charT} \) 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. \[Note: implementations will typically declare some function templates as friends of \text{basic_regex} to achieve this \text{— end note}\]

The functions described in this clause report errors by throwing exceptions of type `regex_error`.

```cpp
template <class charT, 
           class traits = regex_traits<charT> >
class basic_regex
{
public:
    // types:
    typedef charT value_type;
    typedef regex_constants::syntax_option_type flag_type;
    typedef typename traits::locale_type locale_type;

    // 28.8.1, constants:
    static const regex_constants::syntax_option_type icase = regex_constants::icase;
    static const regex_constants::syntax_option_type nosubs = regex_constants::nosubs;
    static const regex_constants::syntax_option_type optimize = regex_constants::optimize;
    static const regex_constants::syntax_option_type collate = regex_constants::collate;

    Draft
28.8 Class template `basic_regex` Regular expressions library

```cpp
static const regex_constants::syntax_option_type
    ECMAScript = regex_constants::ECMAScript;
static const regex_constants::syntax_option_type
    basic = regex_constants::basic;
static const regex_constants::syntax_option_type
    extended = regex_constants::extended;
static const regex_constants::syntax_option_type
    awk = regex_constants::awk;
static const regex_constants::syntax_option_type
    grep = regex_constants::grep;
static const 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);
basic_regex(const basic_regex&);
template <class ST, class SA>
    explicit basic_regex(const basic_string<charT, ST, SA>& p,
        flag_type f = regex_constants::ECMAScript);
template <class InputIterator>
    basic_regex(InputIterator first, InputIterator last,
        flag_type f = regex_constants::ECMAScript);
~basic_regex();
basic_regex& operator=(const basic_regex&);
basic_regex& operator=(const charT* ptr);
    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(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:
Draft
```
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 const regex_constants::syntax_option_type icase = regex_constants::icase;
static const regex_constants::syntax_option_type nosubs = regex_constants::nosubs;
static const regex_constants::syntax_option_type optimize = regex_constants::optimize;
static const regex_constants::syntax_option_type collate = regex_constants::collate;
static const regex_constants::syntax_option_type ECMAScript = regex_constants::ECMAScript;
static const regex_constants::syntax_option_type basic = regex_constants::basic;
static const regex_constants::syntax_option_type extended = regex_constants::extended;
static const regex_constants::syntax_option_type awk = regex_constants::awk;
static const regex_constants::syntax_option_type grep = regex_constants::grep;
static const 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

basic_regex();

Effects: Constructs an object of class basic_regex that does not match any character sequence.

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);
28.8  Class template basic_regex

Regular expressions library  1022

6  Requires: \( p \) shall not be a null pointer.
7  Throws: regex_error if \( p \) is not a valid regular expression.
8  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 \( f \).
9  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 \( e \).
11  Postconditions: flags() and mark_count() return \( e \).flags() and \( e \).mark_count(), respectively.

template <class ST, class SA>
basic_regex(const basic_string<charT, ST, SA>& s,
   flag_type f = regex_constants::ECMAScript);

   Throws: regex_error if \( s \) is not a valid regular expression.
13  Effects: Constructs an object of class basic_regex; the object’s internal finite state machine is constructed from the regular expression contained in the string \( s \), and interpreted according to the flags specified in \( f \).
14  Postconditions: flags() returns \( f \). mark_count() returns the number of marked sub-expressions within the expression.

template <class ForwardInputIterator>
basic_regex(ForwardInputIterator first, ForwardInputIterator last,
   flag_type f = regex_constants::ECMAScript);

   Throws: regex_error if the sequence \([first,last)\) is not a valid regular expression.
16  Effects: Constructs an object of class basic_regex; the object’s internal finite state machine is constructed from the regular expression contained in the sequence of characters \([first,last)\), and interpreted according to the flags specified in \( f \).
17  Postconditions: flags() returns \( f \). mark_count() returns the number of marked sub-expressions within the expression.

basic_regex& operator=(const basic_regex& e);

   Effects: Returns the result of assign(e).

basic_regex& operator=(const charT* ptr);

   Requires: \( ptr \) shall not be a null pointer.
19  Effects: Returns the result of assign(ptr).

template <class ST, class SA>
basic_regex& operator=(const basic_string<charT, ST, SA>& p);

Draft
28.8.3 basic_regex assign

basic_regex& assign(const basic_regex& that);

Effects: Copies that into *this and returns *this.

Postconditions: flags() and mark_count() return that.flags() and that.mark_count(), respectively.

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 corresponds to the Input Iterator requirements ([24.1.1]).

Returns: assign(string_type(first, last), f).

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.
28.9 Class template sub_match

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.

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;

  bool matched;

  difference_type length() const;
}
operator basic_string<value_type>() const;

basic_string<value_type> str() const;

int compare(const sub_match& s) const;
int compare(const basic_string<value_type>& s) const;
int compare(const value_type* s) const;
};

28.9.1 sub_match members

difference_type length();
  Returns: (matched ? distance(first, second) : 0).

operator basic_string<value_type>() const;
  Returns: matched ? basic_string<value_type>(first, second) : basic_string<value_type>().

basic_string<value_type> str() const;
  Returns: matched ? basic_string<value_type>(first, second) : basic_string<value_type>().

int compare(const sub_match& s) const;
  Returns: str().compare(s.str()).

int compare(const basic_string<value_type>& s) const;
  Returns: str().compare(s).

int compare(const value_type* s) const;
  Returns: str().compare(s).

28.9.2 sub_match non-member operators

template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
  Returns: lhs.compare(rhs) == 0.

template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
  Returns: lhs.compare(rhs) != 0.

template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
  Returns: lhs.compare(rhs) < 0.

template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
template <class BiIter>
bool operator>=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs.compare(rhs) >= 0.

template <class BiIter>
bool operator>(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs.compare(rhs) > 0.

template <class BiIter, class ST, class SA>
bool operator==(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs == rhs.str().

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.str().

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.str().

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.str().

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.str().

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.str().
const basic_string<
  typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
const sub_match<BiIter>& rhs);

Returns: lhs <= rhs.str().

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.str() == 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: lhs.str() != 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: lhs.str() < 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: lhs.str() > 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: lhs.str() >= 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: lhs.str() <= rhs.

template <class BiIter>
bool operator==(typename iterator_traits<BiIter>::value_type const* lhs,
const sub_match<BiIter>& rhs);

Returns: lhs == rhs.str().

Draft
Class template sub_match

Regular expressions library

28.9

template <class BiIter>
bool operator!=(typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

Returns: lhs != rhs.str().

template <class BiIter>
bool operator<(typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

Returns: lhs < rhs.str().

template <class BiIter>
bool operator>(typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

Returns: lhs > rhs.str().

template <class BiIter>
bool operator>=(typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

Returns: lhs >= rhs.str().

template <class BiIter>
bool operator<=(typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

Returns: lhs <= rhs.str().

template <class BiIter>
bool operator==(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const* rhs);

Returns: lhs.str() == rhs.

template <class BiIter>
bool operator!=(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const* rhs);

Returns: lhs.str() != rhs.

template <class BiIter>
bool operator<(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const* rhs);

Returns: lhs.str() < rhs.

template <class BiIter>
bool operator>(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const* rhs);

Returns: lhs.str() > rhs.
bool operator>=(const sub_match<BiIter>& lhs,  
   typename iterator_traits<BiIter>::value_type const* rhs);  

Returns: lhs.str() >= rhs.

template <class BiIter>  
bool operator<==(const sub_match<BiIter>& lhs,  
   typename iterator_traits<BiIter>::value_type const* rhs);  

Returns: lhs.str() <= rhs.

template <class BiIter>  
bool operator==(typename iterator_traits<BiIter>::value_type const& lhs,  
   const sub_match<BiIter>& rhs);  

Returns: lhs == rhs.str().

template <class BiIter>  
bool operator!=(typename iterator_traits<BiIter>::value_type const& lhs,  
   const sub_match<BiIter>& rhs);  

Returns: lhs != rhs.str().

template <class BiIter>  
bool operator<(typename iterator_traits<BiIter>::value_type const& lhs,  
   const sub_match<BiIter>& rhs);  

Returns: lhs < rhs.str().

template <class BiIter>  
bool operator>(typename iterator_traits<BiIter>::value_type const& lhs,  
   const sub_match<BiIter>& rhs);  

Returns: lhs > rhs.str().

template <class BiIter>  
bool operator>=(typename iterator_traits<BiIter>::value_type const& lhs,  
   const sub_match<BiIter>& rhs);  

Returns: lhs >= rhs.str().

template <class BiIter>  
bool operator<=(typename iterator_traits<BiIter>::value_type const& lhs,  
   const sub_match<BiIter>& rhs);  

Returns: lhs <= rhs.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 a Sequence, as specified in [23.1.1], except that only operations defined for const-qualified Sequences are supported.

The `sub_match` object stored at index 0 represents sub-expression 0, i.e., the whole match. In this case the `sub_match` member matched is always true. The `sub_match` object stored at index n denotes what matched the marked sub-expression n within the matched expression. If the sub-expression n participated in a regular expression match then the `sub_match` member matched evaluates to true, and members `first` and `second` denote the range of characters `[first, second)` which formed that match. Otherwise matched is false, and members `first` and `second` point to the end of the sequence that was searched. [Note: The `sub_match` objects representing different sub-expressions that did not participate in a regular expression match need not be distinct. — end note]

```cpp
template <class BidirectionalIterator,  
          class Allocator = allocator<sub_match<BidirectionalIterator>>>

class Allocator = allocator<sub_match<BidirectionalIterator>>;
```

Draft
class match_results
{
public:
    typedef sub_match<BidirectionalIterator> value_type;
    typedef typename Allocator::const_reference const_reference;
    typedef const_reference reference;
    typedef implementation defined const_iterator;
    typedef const_iterator iterator;
    typedef typename iterator_traits<BidirectionalIterator>::
        difference_type difference_type;
    typedef typename Allocator::size_type size_type;
    typedef Allocator Allocator;
    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& operator=(const match_results& m);
    ~match_results();

    // 28.10.2, size:
    size_type size() const;
    size_type max_size() const;
    bool empty() const;

    // 28.10.3 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;

    // 28.10.4, format:
    template <class OutputIter>
    OutputIter
    format(OutputIter out,
        const string_type& fmt,
        regex_constants::match_flag_type flags =
            regex_constants::format_default) const;
    string_type
    format(const string_type& fmt,
        regex_constants::match_flag_type flags =
            regex_constants::format_default) const;

Draft
28.10 Class template match_results

Allocator

allocator_type get_allocator() const;

// 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: size() returns 0, str() returns basic_string<charT>().

match_results(const match_results& m);

Effects: Constructs an object of class match_results, as a copy of m.

match_results& operator=(const match_results& m);

Effects: Assigns m to *this. The postconditions of this function are indicated in Table 128

### Table 128: match_results assignment operator effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>size()</td>
<td>m.size()</td>
</tr>
<tr>
<td>str(n)</td>
<td>m.str(n) for all integers n &lt; m.size()</td>
</tr>
<tr>
<td>prefix()</td>
<td>m.prefix()</td>
</tr>
<tr>
<td>suffix()</td>
<td>m.suffix()</td>
</tr>
<tr>
<td>(*this)[n]</td>
<td>m[n] for all integers n &lt; m.size()</td>
</tr>
<tr>
<td>length(n)</td>
<td>m.length(n) for all integers n &lt; m.size()</td>
</tr>
<tr>
<td>position(n)</td>
<td>m.position(n) for all integers n &lt; m.size()</td>
</tr>
</tbody>
</table>

28.10.2 match_results size

size_type size() const;

Returns: One plus the number of marked sub-expressions in the regular expression that was matched if *this represents the result of a successful match. Otherwise returns 0. [Note: The state of a match_results object can be modified only by passing that object to regex_match or regex_search. Sections 28.11.2 and 28.11.3 specify the effects of those algorithms on their match_results arguments. —end note]

size_type max_size() const;

Returns: The maximum number of sub_match elements that can be stored in *this.
bool empty() const;

Returns: size() == 0.

28.10.3 match_results element access

difference_type length(size_type sub = 0) const;

Returns: (*this)[sub].length().

difference_type position(size_type sub = 0) const;

Returns: The distance from the start of the target sequence to (*this)[sub].first.

string_type str(size_type sub = 0) const;

Returns: string_type((*this)[sub]).

const_reference operator[](size_type n) const;

Returns: A reference to the sub_match object representing the character sequence that matched marked sub-expression n. If n == 0 then returns a reference to a sub_match object representing the character sequence that matched the whole regular expression. If n >= size() then returns a sub_match object representing an unmatched sub-expression.

const_reference prefix() const;

Returns: A reference to the sub_match object representing the character sequence from the start of the string being matched/searched to the start of the match found.

const_reference suffix() const;

Returns: A reference to the sub_match object representing the character sequence from the end of the match found to the end of the string being matched/searched.

const_iterator begin() const;

Returns: A starting iterator that enumerates over all the sub-expressions stored in *this.

const_iterator end() const;

Returns: A terminating iterator that enumerates over all the sub-expressions stored in *this.

28.10.4 match_results formatting

template <class OutputIter>
OutputIter format(OutputIter out,
        const string_type& fmt,
        regex_constants::match_flag_type flags =
        regex_constants::format_default) const;

Draft
28.11  Regular expression algorithms

28.10.5  match_results allocator

allocator_type get_allocator() const;

Effects: Returns a copy of the Allocator that was passed to the object’s constructor.

28.10.6  match_results swap

void swap(match_results& that);

Effects: Swaps the contents of the two sequences.

Postcondition: *this contains the sequence of matched sub-expressions that were in that, that contains the
sequence of matched sub-expressions that were in *this.

Complexity: constant time.

template <class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1,
          match_results<BidirectionalIterator, Allocator>& m2);

Effects: m1.swap(m2).

28.11  Regular expression algorithms

28.11.1  exceptions

The algorithms described in this subclause may throw an exception of type regex_error. If such an exception e
is thrown, e.code() shall return either regex_constants::error_complexity or regex_constants::error_stack.

28.11.2  regex_match
template <class BidirectionalIterator, class Allocator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
    match_results<BidirectionalIterator, Allocator>& m,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags =
    regex_constants::match_default);

Requires: The type BidirectionalIterator shall satisfy the requirements of a Bidirectional Iterator ([24.1.4]).

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: If the function returns false, then the effect on parameter m is unspecified except that m.size() returns 0 and m.empty() returns true. Otherwise the effects on parameter m are given in table 129.

Table 129: Effects of regex_match algorithm

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>m.size()</td>
<td>1 + e.mark_count()</td>
</tr>
<tr>
<td>m.empty()</td>
<td>false</td>
</tr>
<tr>
<td>m.prefix().first</td>
<td>first</td>
</tr>
<tr>
<td>m.prefix().second</td>
<td>first</td>
</tr>
<tr>
<td>m.prefix().matched</td>
<td>false</td>
</tr>
<tr>
<td>m.suffix().first</td>
<td>last</td>
</tr>
<tr>
<td>m.suffix().second</td>
<td>last</td>
</tr>
<tr>
<td>m.suffix().matched</td>
<td>false</td>
</tr>
<tr>
<td>m[0].first</td>
<td>first</td>
</tr>
<tr>
<td>m[0].second</td>
<td>last</td>
</tr>
<tr>
<td>m[0].matched</td>
<td>true if a full match was found.</td>
</tr>
<tr>
<td>m[n].first</td>
<td>For all integers 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 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 n &lt; m.size(), true if sub-expression n participated in the match, false otherwise.</td>
</tr>
</tbody>
</table>

Effects: Behaves “as if” by constructing an instance of match_results<BidirectionalIterator> what, and
then returning the result of \texttt{regex_match(first, last, what, e, flags)}.

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

5 \textit{Returns}: \texttt{regex_match(str, str + char_traits<charT>::length(str), m, e, flags)}.

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

6 \textit{Returns}: \texttt{regex_match(s.begin(), s.end(), m, e, flags)}.

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

7 \textit{Returns}: \texttt{regex_match(str, str + char_traits<charT>::length(str), e, flags)}.

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

8 \textit{Returns}: \texttt{regex_match(s.begin(), s.end(), e, flags)}.

\subsection{28.11.3 \texttt{regex_search}}

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

1 \textit{Requires}: Type \texttt{BidirectionalIterator} shall satisfy the requirements of a Bidirectional Iterator (24.1.4).

2 \textit{Effects}: Determines whether there is some sub-sequence within \texttt{[first, last)} that matches the regular expression \texttt{e}. The parameter \texttt{flags} is used to control how the expression is matched against the character sequence.

Draft
Returns true if such a sequence exists, false otherwise.

**Postconditions:** If the function returns false, then the effect on parameter m is unspecified except that m.size() returns 0 and m.empty() returns true. Otherwise the effects on parameter m are given in table 130.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>m.size()</td>
<td>1 + e.mark_count()</td>
</tr>
<tr>
<td>m.empty()</td>
<td>false</td>
</tr>
<tr>
<td>m.prefix().first</td>
<td>first</td>
</tr>
<tr>
<td>m.prefix().second</td>
<td>m[0].first</td>
</tr>
<tr>
<td>m.prefix().matched</td>
<td>m.prefix().first != m.prefix().second</td>
</tr>
<tr>
<td>m.suffix().first</td>
<td>m[0].second</td>
</tr>
<tr>
<td>m.suffix().second</td>
<td>last</td>
</tr>
<tr>
<td>m.suffix().matched</td>
<td>m.suffix().first != m.suffix().second</td>
</tr>
<tr>
<td>m[0].first</td>
<td>The start of the sequence of characters that matched the regular expression</td>
</tr>
<tr>
<td>m[0].second</td>
<td>The end of the sequence of characters that matched the regular expression</td>
</tr>
<tr>
<td>m[0].matched</td>
<td>true if a match was found, and false otherwise.</td>
</tr>
<tr>
<td>m[n].first</td>
<td>For all integers 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 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 n &lt; m.size(), true if sub-expression n participated in the match, false otherwise.</td>
</tr>
</tbody>
</table>

template <class charT, class Allocator, class traits>
bool regex_search(const charT* str, match_results<const charT*, Allocator>& m, const basic_regex<charT, traits>& e, regex_constants::match_flag_type flags = regex_constants::match_default);

**Returns:** The result of regex_search(str, str + char_traits<charT>::length(str), m, e, flags).

Draft
28.11  Regular expression algorithms

Returns: The result of \texttt{regex\_search(s.begin(), s.end(), m, e, flags)}.

\begin{verbatim}
5 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);
\end{verbatim}

Effects: Behaves “as if” by constructing an object what of type match\_results<iteratorBidirectionalI
terator> and then returning the result of \texttt{regex\_search(first, last, what, e, flags)}.

\begin{verbatim}
6 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);
\end{verbatim}

Returns: \texttt{regex\_search(str, str + char\_traits<charT>::length(str), e, flags)}

\begin{verbatim}
7 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);
\end{verbatim}

Returns: \texttt{regex\_search(s.begin(), s.end(), e, flags)}.

28.11.4  regex\_replace

\begin{verbatim}
8 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 basic_string<charT>& fmt,
                               regex_constants::match_flag_type flags =
                               regex_constants::match_default);
\end{verbatim}

Effects: Constructs a regex\_iterator object as if by regex\_iterator<BidirectionalIterator, charT,
traits> i(first, last, e, flags), and uses i to enumerate through all of the matches m of type match\_results<BidirectionalIterator> that occur within the sequence [first, last). If no such matches are found and !(flags & regex\_constants::format\_no\_copy) then calls std::copy(first, last, out). If any matches are found then, for each such match, if !(flags & regex\_constants::format\_no\_copy) calls std::copy(m.prefix().first, m.prefix().second, out), and then calls m.format(out, fmt, flags). Finally, if such a match is found and !(flags & regex\_constants::format\_no\_copy), calls std::copy(last_m.suffix().first, last_m.suffix().second, out) where last_m is a copy of the last match found. If flags & regex\_constants::format\_first\_only is non-zero then only the first match found is replaced.

Draft
template <class traits, class charT>
    basic_string<charT>
    regex_replace(const basic_string<charT>& s,
                  const basic_regex<charT, traits>& e,
                  const basic_string<charT>& fmt,
                  regex_constants::match_flag_type flags =
                  regex_constants::match_default);

Effects: Constructs an empty string result of type basic_string<charT>, calls regex_replace(back_inserter(result), s.begin(), s.end(), e, fmt, flags), and then returns result.

28.12  Regular expression Iterators  [re.iter]

28.12.1  Class template regex_iterator  [re.regiter]

The class template regex_iterator is an iterator adapter. It represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence. A regex_iterator uses regex_search to find successive regular expression matches within the sequence from which it was constructed. After the iterator is constructed, and every time operator++ is used, the iterator finds and stores a value of match_results<BidirectionalIterator>. If the end of the sequence is reached (regex_search returns false), the iterator becomes equal to the end-of-sequence iterator value. The default constructor constructs an end-of-sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of operator* on an end-of-sequence iterator is not defined. For any other iterator value a const match_results<BidirectionalIterator>& is returned. The result of operator-> on an end-of-sequence iterator is not defined. For any other iterator value a const match_results<BidirectionalIterator>* is returned. It is impossible to store things into regex_iterators. Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.
28.12 Regular expression Iterators

regex_iterator(const regex_iterator&);
regex_iterator& operator=(const regex_iterator&);
bool operator==(const regex_iterator&);
bool operator!=(const regex_iterator&);
const value_type& operator*();
const value_type* operator->();
regex_iterator& operator++();
regex_iterator operator++(int);

private:
    // these members are shown for exposition only:
    BidirectionalIterator begin;
    BidirectionalIterator end;
    const regex_type* pregex;
    regex_constants::match_flag_type flags;
    match_results<BidirectionalIterator> match;
};

2 A regex_iterator object 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

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.

28.12.1.2 regex_iterator comparisons

bool operator==(const regex_iterator& right);

Returns: true if *this and right are both end-of-sequence iterators or if begin == right.begin, end == right.end, pregex == right.pregex, flags == right.flags, and match[0] == right.match[0], otherwise false.

bool operator!=(const regex_iterator& right);

Returns: !(this == right).

28.12.1.3 regex_iterator dereference

Draft
const value_type& operator*();

Returns: match.

const value_type* operator->();

Returns: &match.

28.12.1.4 regex_iterator increment

regex_iterator& operator++();

Effects: Constructs a local variable start of type BidirectionalIterator and initializes it with the value of match[0].second.

If the iterator holds a zero-length match and start == end the operator sets *this to the end-of-sequence iterator and returns *this.

Otherwise, if the iterator holds a zero-length match the operator calls regex_search(start, end, match, *pregex, flags | regex_constants::match_not_null | regex_constants::match_continuous). If the call returns true the operator returns *this. Otherwise the operator increments start and continues as if the most recent match was not a zero-length match.

If the most recent match was not a zero-length match, the operator sets flags to flags | regex_constants::match_prev_avail and calls regex_search(start, end, match, *pregex, flags). If the call returns false the iterator sets *this to the end-of-sequence iterator. The iterator then returns *this.

In all cases in which the call to regex_search returns true, match.prefix().first shall be equal to the previous value of match[0].second, and for each index i in the half-open range [0, match.size()) for which match[i].matched is true, match[i].position() shall return distance(begin, match[i].first).

[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]

It is unspecified how the implementation makes these adjustments.

[Note: this means that a compiler may call an implementation-specific search function, in which case a 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

The class template regex_token_iterator is an iterator adapter; 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.

2 When class `regex_token_iterator` is used to enumerate a single sub-expression with index -1 the iterator performs field splitting: that is to say it enumerates one sub-expression for each section of the character container sequence that does not match the regular expression specified.

3 After it is constructed, the iterator finds and stores a value `match_results<BidirectionalIterator>` position and sets the internal count $N$ to zero. It also maintains a sequence `subs` which contains a list of the sub-expressions which will be enumerated. Every time `operator++` is used the count $N$ is incremented; if $N$ exceeds or equals `subs.size()`, then the iterator increments member `position` and sets count $N$ to zero.

4 If the end of sequence is reached (position is equal to the end of sequence iterator), the iterator becomes equal to the end-of-sequence iterator value, unless the sub-expression being enumerated has index -1, in which case the iterator enumerates one last sub-expression that contains all the characters from the end of the last regular expression match to the end of the input sequence being enumerated, provided that this would not be an empty sub-expression.

5 The default constructor constructs an end-of-sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end-of-sequence iterator is not defined. For any other iterator value a `const sub_match<BidirectionalIterator>&` is returned. The result of `operator->` on an end-of-sequence iterator is not defined. For any other iterator value a `const sub_match<BidirectionalIterator>*` is returned.

6 It is impossible to store things into `regex_iterators`. Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.

```cpp
template <class BidirectionalIterator, 
        class charT = typename iterator_traits<BidirectionalIterator>::value_type, 
        class traits = regex_traits<charT> >

class regex_token_iterator
{
public:
    typedef basic_regex<charT, traits> regex_type;
    typedef sub_match<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_token_iterator();
    regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b, 
                        const regex_type& re, 
                        int submatch = 0,
                        regex_constants::match_flag_type m = 
                        regex_constants::match_default);
    regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b, 
                        const regex_type& re, 
                        const std::vector<int>& submatches, 
                        regex_constants::match_flag_type m = 
                        regex_constants::match_default);

    Draft
```
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);

regex_token_iterator(const regex_token_iterator&);
regex_token_iterator& operator=(const regex_token_iterator&);

bool operator==(const regex_token_iterator&);
bool operator!=(const regex_token_iterator&);

const value_type& operator*();
const value_type* operator->();

regex_token_iterator& operator++();
regex_token_iterator operator++(int);

private:
    // data members for exposition only:
    typedef regex_iterator<BidirectionalIterator, charT, traits> position_iterator;
    position_iterator position;
    const value_type *result;
    value_type suffix;
    std::size_t N;
    std::vector<int> subs;
};

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

regex_token_iterator();

Effects: Constructs the end-of-sequence iterator.

regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
    const regex_type& re,
    int submatch = 0,
    regex_constants::match_flag_type m =
    regex_constants::match_default);

regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
    const regex_type& re,
    const std::vector<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);

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 constructor initializes the member subs to hold a copy of the sequence of integer values pointed to by the iterator range [&submatches, &submatches + N).

Each constructor then sets N to 0, and position to position_iterator(a, b, re, m). If position is not an end-of-sequence iterator the constructor sets result to the address of the current match. Otherwise if any of the values stored in subs is equal to -1 the constructor sets *this to a suffix iterator that points to the range [a,b), otherwise the constructor sets *this to an end-of-sequence iterator.

28.12.2.2 regex_token_iterator comparisons

bool operator==(const regex_token_iterator& right);

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);

Returns: !(*this == right).

28.12.2.3 regex_token_iterator dereference

const value_type& operator*();

Returns: *result.

const value_type* operator->();

Returns: result.

28.12.2.4 regex_token_iterator increment

regex_token_iterator& operator++();

Effects: Constructs a local variable prev of type position_iterator, initialized with the value of position.

If *this is a suffix iterator, sets *this to an end-of-sequence iterator.

Otherwise, if N + 1 < subs.size(), increments N and sets result to the address of the current match.
Otherwise, sets \( N \) to 0 and increments position. If position is not an end-of-sequence iterator the operator sets result to the address of the current match.

Otherwise, if any of the values stored in \( \text{subs} \) is equal to -1 and \( \text{prev} \rightarrow \text{suffix}(). \text{length}() \) is not 0 the operator sets \( * \text{this} \) to a suffix iterator that points to the range \( \text{[prev} \rightarrow \text{suffix}(). \text{first}, \text{prev} \rightarrow \text{suffix}(). \text{second}) \).

Otherwise, sets \( * \text{this} \) to an end-of-sequence iterator.

\[ \text{Returns: } * \text{this} \]

\[ \text{regex_token_iterator& operator++(int);} \]

\[ \text{Effects: Constructs a copy tmp of } * \text{this, then calls } ++(* \text{this}). \]

\[ \text{Returns: tmp.} \]

### 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.

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 object 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:

\[ \text{CharacterClass ::} \]
\[ \quad [ \text{[lookahead} \notin \{\}]} \text{ClassRanges } \]
\[ \quad [ \ ^ \text{ClassRanges } \]

\[ \text{ClassAtom ::} \]
\[ \quad - \]
\[ \text{ClassAtomNoDash} \]
\[ \text{ClassAtomExClass} \]
\[ \text{ClassAtomCollatingElement} \]
\[ \text{ClassAtomEquivalence} \]

The following new productions are then added:

\[ \text{ClassAtomExClass ::} \]
\[ \quad [: \text{ClassName :]} \]

\[ \text{ClassAtomCollatingElement ::} \]
\[ \quad [. \text{ClassName .}] \]

\[ \text{ClassAtomEquivalence ::} \]
\[ \quad [= \text{ClassName =}] \]

\[ \text{ClassName ::} \]

Draft
28.13 Modified ECMAScript regular expression grammar

```
ClassNameCharacter
ClassNameCharacter ClassName

ClassNameCharacter ::
  SourceCharacter but not one of "," "=" ":"
```

5 The productions ClassAtomExClass, ClassAtomCollatingElement and ClassAtomEquivalence provide functionality equivalent to that of the same features in regular expressions in POSIX.

6 The regular expression grammar may be modified by any `regex_constants::syntax_option_type` flags specified when constructing an object of type specialization of `basic_regex` according to the rules in table 125.

7 A `ClassName` production, when used in `ClassAtomExClass`, is not valid if `traits_inst.lookup_classname` returns zero for that name. The names recognized as valid `ClassName`s are determined by the type of the traits class, but at least the following names shall be recognized: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`, `d`, `s`, `w`. In addition the following expressions shall be equivalent:

```
\d and [[:digit:]]
\D and [^[:digit:]]
\s and [[:space:]]
\S and [^[:space:]]
\w and [[:alnum:]]
\W and [^[:alnum:]]
```

8 A `ClassName` production when used in a `ClassAtomCollatingElement` production is not valid if the value returned by `traits_inst.lookup_classname` for that name is an empty string.

9 The results from multiple calls to `traits_inst.lookup_classname` can be bitwise OR'ed together and subsequently passed to `traits_inst.isctype`.

10 A `ClassName` production when used in a `ClassAtomEquivalence` production is not valid if the value returned by `traits_inst.lookup_classname` 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_classname` is an empty string.

11 When the sequence of characters being transformed to a finite state machine contains an invalid class name the translator shall throw an exception object of type `regex_error`.

12 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]

13 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`.

14 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:

— 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.

— 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());
```

— 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.

— 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)) is true.

Draft
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

New context-dependent keywords are introduced into a program by \texttt{typedef} (7.1.3), \texttt{namespace} (7.3.1), class (clause 9), enumeration (7.2), and \texttt{template} (clause 14) declarations.

\begin{verbatim}
typedef-name:
    identifier
namespace-name:
    original-namespace-name
    namespace-alias
original-namespace-name:
    identifier
namespace-alias:
    identifier
class-name:
    identifier
    template-id
enum-name:
    identifier
template-name:
    identifier
\end{verbatim}

Note that a \texttt{typedef-name} naming a class is also a \texttt{class-name} (9.1).

A.2 Lexical conventions

\begin{verbatim}
hex-quad:
    hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit
\end{verbatim}
universal-character-name:
  \u hex-quad
  \U hex-quad hex-quad

preprocessing-token:
  header-name
  identifier
  pp-number
  character-literal
  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 nondigit
  pp-number 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

literal:
  integer-literal
  character-literal
  floating-literal
  string-literal
  boolean-literal

integer-literal:
  decimal-literal integer-suffix\textsubscript{opt}
  octal-literal integer-suffix\textsubscript{opt}
  hexadecimal-literal integer-suffix\textsubscript{opt}

decimal-literal:
  nonzero-digit
  decimal-literal digit

octal-literal:
  0
  octal-literal octal-digit

hexadecimal-literal:
  0x hexadecimal-digit
  0X hexadecimal-digit
  hexadecimal-literal hexadecimal-digit

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

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

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\textsubscript{opt}
  unsigned-suffix long-long-suffix\textsubscript{opt}
  long-suffix unsigned-suffix\textsubscript{opt}
  long-long-suffix unsigned-suffix\textsubscript{opt}

unsigned-suffix: one of
  u U

long-suffix: one of
  l L

long-long-suffix: one of
  ll LL
character-literal:
' c-char-sequence '
'u' c-char-sequence '
'U' c-char-sequence '
'L' c-char-sequence '
c-char-sequence:
c-char
c-char-sequence c-char
c-char:
any member of the source character set except
the single-quote ' , backslash \, or new-line character
escape-sequence
universal-character-name
escape-sequence:
simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence
simple-escape-sequence: one of
' ' '" '? '\ '
\a \b \f \n \r \t \v
octal-escape-sequence:
\ octal-digit
\ octal-digit octal-digit
\ octal-digit octal-digit octal-digit
hexadecimal-escape-sequence:
\x hexadecimal-digit
hexadecimal-escape-sequence hexadecimal-digit
floating-literal:
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:
+ sign opt digit-sequence
- sign opt digit-sequence
sign: one of
+ -
digit-sequence:
digit
digit-sequence digit
floating-suffix: one of
f F L

Draft
string-literal:
   " s-char-sequence opt "
   u" s-char-sequence opt "
   U" s-char-sequence opt "
   L" 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

boolean-literal:
   false
   true

A.3 Basic concepts

translation-unit:
   declaration-seq opt

A.4 Expressions

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

id-expression:
   unqualified-id
   qualified-id

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

qualified-id:
   ::opt nested-name-specifier template opt unqualified-id
   :: identifier
   :: operator-function-id
   :: template-id

nested-name-specifier:
   type-name ::
   namespace-name ::
   nested-name-specifier identifier ::
   nested-name-specifier template opt simple-template-id ::
### A.4 Expressions

postfix-expression:
- primary-expression
- postfix-expression [ expression ]
- postfix-expression ( expression-listopt )
- simple-type-specifier ( expression-listopt )
- typename-specifier ( expression-listopt )
- postfix-expression . templateopt id-expression
- postfix-expression -> templateopt 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 )
- typeid ( expression )
- typeid ( type-id )

expression-list:
- assignment-expression ...opt
- expression-list , assignment-expression ...opt

cast-expression:
- expression
- sizeof ... ( identifier )
- alignof ( type-id )
- new-expression
- delete-expression

ew-placement:
- ( expression-list )

type-specifier-seq:
- type-specifier-seq new-declaratoropt

new-declarator:
  ptr-operator new-declarator_opt
  direct-new-declarator

direct-new-declarator:
  [ expression ]
  direct-new-declarator [ constant-expression ]

new-initializer:
  ( expression-list_opt )
delete-expression:
  ::opt delete cast-expression
  ::opt delete [ ] cast-expression
cast-expression:
  unary-expression
  ( type-id ) cast-expression

pm-expression:
  cast-expression
  pm-expression . * cast-expression
  pm-expression ->* cast-expression

multiplicative-expression:
  pm-expression
  multiplicative-expression * pm-expression
  multiplicative-expression / pm-expression
  multiplicative-expression % pm-expression

additive-expression:
  multiplicative-expression
  additive-expression + multiplicative-expression
  additive-expression - multiplicative-expression

shift-expression:
  additive-expression
  shift-expression << additive-expression
  shift-expression >> additive-expression

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
A.5 Statements

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

assignment-expression:
  conditional-expression
  logical-or-expression assignment-operator assignment-expression
  throw-expression

assignment-operator: one of
  = += -= *= /= %= += -= >>= <<= &= ^= |=

expression:
  assignment-expression
  expression , assignment-expression

constant-expression:
  conditional-expression

constant-expression:
  conditional-expression

A.5 Statements

statement:
  labeled-statement
  expression-statement
  compound-statement
  selection-statement
  iteration-statement
  jump-statement
  declaration-statement
  try-block

labeled-statement:
  identifier : statement
  case constant-expression : statement
  default : statement
expression-statement:
  expression_opt ;

compound-statement:
  { statement-seq_opt }

statement-seq:
  statement
  statement-seq statement

selection-statement:
  if ( condition ) statement
  if ( condition ) statement else statement
  switch ( condition ) statement

condition:
  expression
  type-specifier-seq declarator = assignment-expression

iteration-statement:
  while ( condition ) statement
  do statement while ( expression ) ;
  for ( for-init-statement condition_opt ; expression_opt ) statement

for-init-statement:
  expression-statement
  simple-declaration

jump-statement:
  break ;
  continue ;
  return expression_opt ;
  goto identifier ;

declaration-statement:
  block-declaration

A.6 Declarations

declaration-seq:
  declaration
  declaration-seq declaration

declaration:
  block-declaration
  function-definition
  template-declaration
  explicit-instantiation
  explicit-specialization
  linkage-specification
  namespace-definition
A.6 Declarations

**Grammar summary**

- **block-declaration:**
  - simple-declaration
  - asm-definition
  - namespace-alias-definition
  - using-declaration
  - using-directive
  - static_assert-declaration
  - alias-declaration

- **alias-declaration:**
  - using identifier = type-id

- **simple-declaration:**
  - decl-specifier-seqopt init-declarator-listopt ;

- **static_assert-declaration:**
  - static assert (constant-expression, string-literal) ;

- **decl-specifier:**
  - storage-class-specifier
  - type-specifier
  - function-specifier
  - friend
dtypedef
  - constexpr
  - alignment-specifier

- **decl-specifier-seq:**
  - decl-specifier-seqopt decl-specifier

- **storage-class-specifier:**
  - register
  - static
  - extern
  - mutable

- **function-specifier:**
  - inline
  - virtual
  - explicit

- **typedef-name:**
  - identifier

- **type-specifier:**
  - simple-type-specifier
  - class-specifier
  - enum-specifier
  - elaborated-type-specifier
typename-specifier
cv-qualification

Draft
simple-type-specifier:
  ::opt nested-name-specifieropt type-name
  ::opt nested-name-specifier template simple-template-id
  char
  char16_t
  char32_t
  wchar_t
  bool
  short
  int
  long
  signed
  unsigned
  float
  double
  void
  auto
dcltype ( expression )

type-name:
  class-name
  enum-name
  typedef-name

elaborated-type-specifier:
  class-key ::opt nested-name-specifieropt identifier
  class-key ::opt nested-name-specifieropt templateopt simple-template-id
  enum enum-key ::opt nested-name-specifieropt identifier

alignment-specifier:
  alignas ( constant-expression )
  alignas ( type-id )

enum-name:
  identifier

enum-specifier:
  enum enum-key identifieropt enum-baseopt { enumerator-listopt }
  enum enum-key identifieropt enum-baseopt { enumerator-list , }

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
A.7 Declarators

namespace-name:
    original-namespace-name
    namespace-alias
original-namespace-name:
    identifier
namespace-definition:
    named-namespace-definition
    unnamed-namespace-definition
named-namespace-definition:
    original-namespace-definition
    extension-namespace-definition
original-namespace-definition:
    namespace identifier { namespace-body }
extension-namespace-definition:
    namespace original-namespace-name { namespace-body }
unnamed-namespace-definition:
    namespace { namespace-body }
namespace-body:
    declaration-seq opt
namespace-alias:
    identifier
namespace-alias-definition:
    namespace identifier = qualified-name-specifier ;
qualified-name-specifier:
    :: opt nested-name-specifier opt namespace-name
using-declaration:
    using typename opt :: opt nested-name-specifier unqualified-id ;
    using :: unqualified-id ;
using-directive:
    using namespace :: opt nested-name-specifier opt namespace-name ;
asm-definition:
    asm ( string-literal ) ;
linkage-specification:
    extern string-literal { declaration-seq opt }
    extern string-literal declaration

A.7 Declarators

init-declarator-list:
    init-declarator
    init-declarator-list , init-declarator
init-declarator:
    declarator initializer opt
declarator:
    direct-declarator
    ptr-operator declarator

Draft
direct-declarator:
  declarator-id
  direct-declarator ( parameter-declaration-clause ) cv-qualifier-seq_opt exception-specification_opt
direct-declarator ( constant-expression_opt )
  ( declarator )

ptr-operator:
  * cv-qualifier-seq_opt
  &
  &&
  :: opt nested-name-specifier * cv-qualifier-seq_opt

cv-qualifier-seq:
  cv-qualifier cv-qualifier-seq_opt

cv-qualifier:
  const
  volatile
deaרator-id:
  . . . opt id-expression
  . . . opt nested-name-specifier_opt class-name
type-id:
  type-specifier-seq abstract-declarator_opt
type-specifier-seq:
  type-specifier type-specifier-seq_opt
abstract-declarator:
  ptr-operator abstract-declarator_opt
direct-abstract-declarator
  . . .
direct-abstract-declarator:
  direct-abstract-declarator_opt
  ( parameter-declaration-clause ) cv-qualifier-seq_opt exception-specification_opt
direct-abstract-declarator_opt ( constant-expression_opt )
  ( abstract-declarator )
parameter-declaration-clause:
  parameter-declaration-list_opt . . . opt
  parameter-declaration-list , . . .
parameter-declaration-list:
  parameter-declaration
  parameter-declaration-list , parameter-declaration
parameter-declaration:
  decl-specifier-seq declarator
decl-specifier-seq declarator = assignment-expression
decl-specifier-seq abstract-declarator_opt
decl-specifier-seq abstract-declarator_opt = assignment-expression
A.8 Classes

function-definition:
  decl-specifier-seq_opt declarator ctor-initializer_opt function-body
  decl-specifier-seq_opt declarator function-try-block
  decl-specifier-seq_opt declarator = default ;
  decl-specifier-seq_opt declarator = delete ;

function-body:
  ctor-initializer_opt compound-statement
  function-try-block

initializer:
  = initializer-clause
  ( expression-list )

initializer-clause:
  assignment-expression
  { { initializer-list ,opt } }
  { }

initializer-list:
  initializer-clause ...opt
  initializer-list , initializer-clause ...opt

A.8 Classes

class-name:
  identifier
  simple-template-id

class-specifier:
  class-head { member-specification_opt }

class-head:
  class-key identifier_opt base-clause_opt
  class-key nested-name-specifier identifier base-clause_opt
  class-key nested-name-specifier_opt simple-template-id base-clause_opt

class-key:
  class
  struct
  union

member-specification:
  member-declaration member-specification_opt
  access-specifier : member-specification_opt

member-declaration:
  decl-specifier-seq_opt member-declarator-list_opt ;
  function-definition ;opt
  ::opt nested-name-specifier template_opt unqualified-id ;
  using-declaration
  static_assert-declaration
  template-declaration

member-declarator-list:
  member-declarator
  member-declarator-list , member-declarator
A.9 Derived classes

member-declarator:
  declarator pure-specifier_opt
  declarator constant-initializer_opt
  identifier_opt : constant-expression

pure-specifier:
  = 0

constant-initializer:
  = constant-expression

A.9 Derived classes

base-clause:
  : base-specifier-list

base-specifier-list:
  base-specifier ... opt
  base-specifier-list , base-specifier ... opt

base-specifier:
  :: opt nested-name-specifier_opt class-name
  virtual access-specifier_opt :: opt nested-name-specifier_opt class-name
  access-specifier virtual_opt :: opt nested-name-specifier_opt class-name

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

ctor-initializer:
  : mem-initializer-list

mem-initializer-list:
  mem-initializer ... opt
  mem-initializer , mem-initializer-list ... opt

mem-initializer:
  mem-initializer-id ( expression-list_opt )

mem-initializer-id:
  :: opt nested-name-specifier_opt class-name
  identifier
A.11 Overloading

operator-function-id:
  operator operator

A.12 Templates

template-declaration:
  export_opt template < template-parameter-list > declaration

template-parameter-list:
  template-parameter
  template-parameter-list , template-parameter

template-parameter:
  type-parameter
  parameter-declaration

type-parameter:
   class . . . opt identifier_opt
   class identifier_opt = type-id
   typename . . . opt identifier_opt
   typename identifier_opt = type-id
   template < template-parameter-list > class . . . opt identifier_opt
   template < template-parameter-list > class identifier_opt = id-expression

simple-template-id:
  template-name < template-argument-list_opt >

template-id:
  simple-template-id
  operator-function-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

typename-specifier:
  typename :: opt nested-name-specifier identifier
  typename :: opt nested-name-specifier template_opt simple-template-id

external_opt template declaration

eexplicit-specialization:
  template < > declaration

Draft
A.13 Exception handling

```
try-block:
    try compound-statement handler-seq

function-try-block:
    try ctor-initializer_opt function-body compound-statement handler-seq

handler-seq:
    handler handler-seq_opt

handler:
    catch ( exception-declaration ) compound-statement

exception-declaration:
    type-specifier-seq declarator
    type-specifier-seq abstract-declarator
    type-specifier-seq ...

throw-expression:
    throw assignment-expression_opt

exception-specification:
    throw ( type-id-list_opt )

type-id-list:
    type-id ... opt
    type-id-list , type-id ... opt
```

A.14 Preprocessing directives

```
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 lparen identifier-list_opt ) replacement-list new-line
    # define identifier lparen identifier . . . ) replacement-list new-line
    # define identifier lparen identifier-list, . . . ) replacement-list new-line
    # undef identifier new-line
    # line pp-tokens new-line
    # error pp-tokens_opt new-line
    # pragma pp-tokens_opt new-line
    # new-line

text-line:
    pp-tokens_opt new-line

non-directive:
    pp-tokens_opt new-line

lparen:
    a ( character not immediately preceded by white-space

identifier-list:
    identifier
    identifier-list , identifier

replacement-list:
    pp-tokens_opt

pp-tokens:
    preprocessing-token
    pp-tokens preprocessing-token

new-line:
    the new-line character
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.

- Nesting levels of compound statements, iteration control structures, and selection control structures [256].
- Nesting levels of conditional inclusion [256].
- Pointer, array, and function declarators (in any combination) modifying a class, arithmetic, or incomplete type in a declaration [256].
- Nesting levels of parenthesized expressions within a full expression [256].
- Number of characters in an internal identifier or macro name [1 024].
- Number of characters in an external identifier [1 024].
- External identifiers in one translation unit [65 536].
- Identifiers with block scope declared in one block [1 024].
- Macro identifiers simultaneously defined in one translation unit [65 536].
- Parameters in one function definition [256].
- Arguments in one function call [256].
- Parameters in one macro definition [256].
- Arguments in one macro invocation [256].
- Characters in one logical source line [65 536].
- Characters in a character string literal or wide string literal (after concatenation) [65 536].
— Size of an object [262 144].
— Nesting levels for #include files [256].
— Case labels for a switch statement (excluding those for any nested switch statements) [16 384].
— Data members in a single class [16 384].
— Enumeration constants in a single enumeration [4 096].
— Levels of nested class definitions in a single member-specification [256].
— Functions registered by atexit() [32].
— Direct and indirect base classes [16 384].
— Direct base classes for a single class [1 024].
— Members declared in a single class [4 096].
— Final overriding virtual functions in a class, accessible or not [16 384].
— Direct and indirect virtual bases of a class [1 024].
— Static members of a class [1 024].
— Friend declarations in a class [4 096].
— Access control declarations in a class [4 096].
— Member initializers in a constructor definition [6 144].
— Scope qualifications of one identifier [256].
— Nested external specifications [1 024].
— Template arguments in a template declaration [1 024].
— Recursively nested template instantiations [17].
— Handlers per try block [256].
— Throw specifications on a single function declaration [256].
Appendix C
(informative)
Compatibility

C.1 C++ and ISO C

The subclauses of this subclause list the differences between C++ and ISO C, by the chapters of this document.

C.1.1 Clause 2: lexical conventions

2.3
Change: C++ style comments (//) are added
A pair of slashes now introduce a one-line comment.
Rationale: This style of comments is a useful addition to the language.
Effect on original feature: Change to semantics of well-defined feature. A valid ISO C expression containing a division operator followed immediately by a C-style comment will now be treated as a C++ style comment. For example:

```c
{ 
    int a = 4;
    int b = 8       // divide by a*/ a;
    *a;
}
```

Difficulty of converting: Syntactic transformation. Just add white space after the division operator.
How widely used: The token sequence //* probably occurs very seldom.

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.
2.13.2
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 2.13.4:
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: Simple syntactic transformation, because string literals can be converted to char*; (4.2). The most common cases are handled by a new but deprecated standard conversion:

```c
char* p = "abc";  // valid in C, deprecated in C++
char* q = expr ? "abc" : "de";  // valid in C, invalid in C++
```

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,
```
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,
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 built-in 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.

Draft
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.

4.10 Change: Only pointers to non-const and non-volatile objects may be implicitly converted to void*

**Rationale:** This improves type safety.

**Effect on original feature:** Deletion of semantically well-defined feature.

**Difficulty of converting:** Could be automated. A C program containing such an implicit conversion from (e.g.) pointer-to-const-object to void* will receive a diagnostic message. The correction is to add an explicit cast.

**How widely used:** Seldom.

C.1.3 Clause 5: expressions

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.

Draft
Difficulty of converting: Syntactic transformation.
How widely used: Seldom.
5.16, 5.17, 5.18

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 values.
Effect on original feature: Change to semantics of well-defined feature. Some C expressions that implicitly rely on lvalue-to-rvalue conversions will yield different results. For example,

    char arr[100];
    sizeof(0, arr)

yields 100 in C++ and sizeof(char*) in C.
Difficulty of converting: Programs must add explicit casts to the appropriate rvalue.
How widely used: Rare.

C.1.4 Clause 6: statements [diff.stat]

6.4.2, 6.6.4 (switch and goto statements)
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, e.g. zero.
How widely used: Seldom. For several years, many existing C implementations have produced warnings in this case.

C.1.5 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
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:

```c
typedef struct name1 { /*...*/ } name1; // valid C and C++
struct name { /*...*/ }; // valid C, invalid C++
```

**Rationale:** For ease of use, C++ doesn’t require that a type name be prefixed with the keywords `class`, `struct` or `union` when used in object declarations or type casts.

Example:

```c
class name { /*...*/ }; // i has type class name
```

**Effect on original feature:** Deletion of semantically well-defined feature.

**Difficulty of converting:** Semantic transformation. One of the 2 types has to be renamed.

**How widely used:** Seldom.

### 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 (type specifiers)

**Change:** Banning implicit int

In C++ a `decl-specifier-seq` must contain a `type-specifier`. In the following example, the left-hand column presents valid C; the right-hand column presents equivalent C++:

```c
```
void f(const parm); void f(const int parm);
const n = 3; const int n = 3;
main() int main()
/* ... */ /* ... */

Rationale: In C++, implicit int creates several opportunities for ambiguity between expressions involving function-like casts and declarations. Explicit declaration is increasingly considered to be proper style. Liaison with WG14 (C) indicated support for (at least) deprecating implicit int in the next revision of C.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation. Could be automated.
How widely used: Common.

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:
enum color { red, blue, green };
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:
enum e { A };
sizeof(A) == sizeof(int) // in C
sizeof(A) == sizeof(e) // in C++
/* and sizeof(int) is not necessarily equal to sizeof(e) */

Rationale: In C++, an enumeration is a distinct type.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Seldom. The only time this affects existing C code is when the size of an enumerator is taken. Taking the size of an enumerator is not a common C coding practice.

C.1.6 Clause 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++
// int f( unknown ) in C
```

**Rationale:** This is to avoid erroneous function calls (i.e. function calls with the wrong number or type of arguments).

**Effect on original feature:** Change to semantics of well-defined feature. This feature was marked as “obsolescent” in C.

**Difficulty of converting:** Syntactic transformation. The function declarations using C incomplete declaration style must be completed to become full prototype declarations. A program may need to be updated further if different calls to the same (non-prototype) function have different numbers of arguments or if the type of corresponding arguments differed.

**How widely used:** Common.

8.3.5 [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.1.7 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 built-in 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.

9.7

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;
};
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 ‘I’ really is.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation. Either the type or the struct member has to be renamed.

How widely used: Seldom.

C.1.8 Clause 12: special member functions

12.8 (copying class objects)

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; };
struct X x1, x2;
v volatile struct X x3 = {0};
x1 = x3;        // invalid C++
x2 = x3;        // 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.1.9 Clause 16: preprocessing directives

16.8 (predefined names)

**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 Standard C 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 ISO/IEC 9899:1990 and ISO/IEC 9899:1990/DAM 1 noted in other subclauses (17.4.1.2, 18.1, 21.4).

The C++ Standard library provides 56 standard macros from the C library, as shown in Table 131.

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 131: Standard Macros

<table>
<thead>
<tr>
<th>assert</th>
<th>HUGE_VAL</th>
<th>NULL &lt;cstdlib&gt;</th>
<th>SIG_ERR</th>
<th>TMP_MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFSIZ</td>
<td>L_tmpnam</td>
<td>NULL &lt;cstring&gt;</td>
<td>SIG_IGN</td>
<td>va_arg</td>
</tr>
<tr>
<td>CLOCKS_PER_SEC</td>
<td>LC_ALL</td>
<td>NULL &lt;ctime&gt;</td>
<td>SIGABRT</td>
<td>va_end</td>
</tr>
<tr>
<td>EDOM</td>
<td>LC_COLLATE</td>
<td>NULL &lt;cwchar&gt;</td>
<td>SIGFPE</td>
<td>va_start</td>
</tr>
<tr>
<td>EILSEQ</td>
<td>LC_CTYPE</td>
<td>offsetof</td>
<td>SIGILL</td>
<td>WCHAR_MAX</td>
</tr>
<tr>
<td>EOF</td>
<td>LC_MONETARY</td>
<td>RAND_MAX</td>
<td>SIGINT</td>
<td>WCHAR_MIN</td>
</tr>
<tr>
<td>ERANGE</td>
<td>LC_NUMERIC</td>
<td>SEEK_CUR</td>
<td>SIGSEGV</td>
<td>WEOF &lt;cwchar&gt;</td>
</tr>
<tr>
<td>errno</td>
<td>LC_TIME</td>
<td>SEEK_END</td>
<td>SIGTERM</td>
<td>WEOF &lt;cwctype&gt;</td>
</tr>
<tr>
<td>EXIT_FAILURE</td>
<td>MB_CUR_MAX</td>
<td>SEEK_SET</td>
<td>stderr</td>
<td>_IOFBF</td>
</tr>
<tr>
<td>EXIT_SUCCESS</td>
<td>NULL &lt;locale&gt;</td>
<td>setjmp</td>
<td>stdin</td>
<td>_IOLBF</td>
</tr>
<tr>
<td>FILENAME_MAX</td>
<td>NULL &lt;cstdlib&gt;</td>
<td>SIG_DFL</td>
<td>stdout</td>
<td>_IONBF</td>
</tr>
<tr>
<td>FOPEN_MAX</td>
<td>NULL &lt;cstdio&gt;</td>
<td>SIG_DFL</td>
<td>stdout</td>
<td>_IONBF</td>
</tr>
</tbody>
</table>

The C++ Standard library provides 57 standard values from the C library, as shown in Table 132.

The C++ Standard library provides 20 standard types from the C library, as shown in Table 133.

The C++ Standard library provides 2 standard structs from the C library, as shown in Table 134.

The C++ Standard library provides 209 standard functions from the C library, as shown in Table 135.
Table 132: Standard Values

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
<th>Value 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR_BIT</td>
<td>FLT_DIG</td>
<td>INT_MIN</td>
<td>MB_LEN_MAX</td>
<td></td>
</tr>
<tr>
<td>CHAR_MAX</td>
<td>FLT_EPSILON</td>
<td>LDBL_DIG</td>
<td>SCHAR_MAX</td>
<td></td>
</tr>
<tr>
<td>CHAR_MIN</td>
<td>FLT_MANT_DIG</td>
<td>LDBL_EPSILON</td>
<td>SCHAR_MIN</td>
<td></td>
</tr>
<tr>
<td>DBL_DIG</td>
<td>FLT_MAX</td>
<td>LDBL_MANT_DIG</td>
<td>SHRT_MAX</td>
<td></td>
</tr>
<tr>
<td>DBL_EPSILON</td>
<td>FLT_MAX_10_EXP</td>
<td>LDBL_MAX</td>
<td>SHRT_MIN</td>
<td></td>
</tr>
<tr>
<td>DBL_MANT_DIG</td>
<td>FLT_MAX_EXP</td>
<td>LDBL_MAX_10_EXP</td>
<td>UCHAR_MAX</td>
<td></td>
</tr>
<tr>
<td>DBL_MAX</td>
<td>FLT_MIN</td>
<td>LDBL_MAX_EXP</td>
<td>UINT_MAX</td>
<td></td>
</tr>
<tr>
<td>DBL_MAX_10_EXP</td>
<td>FLT_MIN_10_EXP</td>
<td>LDBL_MIN</td>
<td>ULONG_MAX</td>
<td></td>
</tr>
<tr>
<td>DBL_MAX_EXP</td>
<td>FLT_MIN_EXP</td>
<td>LDBL_MIN_10_EXP</td>
<td>USRT_MAX</td>
<td></td>
</tr>
<tr>
<td>DBL_MIN</td>
<td>FLT_RADIX</td>
<td>LDBL_MIN_EXP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBL_MIN_10_EXP</td>
<td>FLT_ROUNDS</td>
<td></td>
<td>LONG_MAX</td>
<td></td>
</tr>
<tr>
<td>DBL_MIN_EXP</td>
<td>INT_MAX</td>
<td></td>
<td>LONG_MIN</td>
<td></td>
</tr>
</tbody>
</table>

Table 133: Standard Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock_t</td>
<td></td>
</tr>
<tr>
<td>ldiv_t</td>
<td></td>
</tr>
<tr>
<td>size_t</td>
<td>&lt;cstdio&gt;</td>
</tr>
<tr>
<td>va_list</td>
<td></td>
</tr>
<tr>
<td>div_t</td>
<td></td>
</tr>
<tr>
<td>mbstate_t</td>
<td></td>
</tr>
<tr>
<td>size_t</td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>wctrans_t</td>
<td></td>
</tr>
<tr>
<td>FILE</td>
<td></td>
</tr>
<tr>
<td>ptdiff_t</td>
<td></td>
</tr>
<tr>
<td>size_t</td>
<td>&lt;cstring&gt;</td>
</tr>
<tr>
<td>wctype_t</td>
<td></td>
</tr>
<tr>
<td>fpos_t</td>
<td></td>
</tr>
<tr>
<td>sig_atomic_t</td>
<td></td>
</tr>
<tr>
<td>size_t</td>
<td>&lt;ctime&gt;</td>
</tr>
<tr>
<td>wint_t</td>
<td>&lt;cwchar&gt;</td>
</tr>
<tr>
<td>jmp_buf</td>
<td></td>
</tr>
<tr>
<td>size_t</td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>time_t</td>
<td></td>
</tr>
<tr>
<td>wint_t</td>
<td>&lt;cwctype&gt;</td>
</tr>
</tbody>
</table>

Table 134: Standard Structs

<table>
<thead>
<tr>
<th>Struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>lconv</td>
</tr>
<tr>
<td>tm</td>
</tr>
</tbody>
</table>

Draft
### Table 135: Standard Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>fmod</td>
<td>isupper</td>
<td>mktime</td>
<td>strftime</td>
</tr>
<tr>
<td>abs</td>
<td>fopen</td>
<td>iswalnum</td>
<td>modf</td>
<td>strlen</td>
</tr>
<tr>
<td>acos</td>
<td>fprintf</td>
<td>iswalpha</td>
<td>perror</td>
<td>strncat</td>
</tr>
<tr>
<td>asctime</td>
<td>fputc</td>
<td>iswcntrl</td>
<td>pow</td>
<td>wscmp</td>
</tr>
<tr>
<td>asin</td>
<td>fputs</td>
<td>iswctype</td>
<td>printf</td>
<td>wcsncpy</td>
</tr>
<tr>
<td>atan</td>
<td>fputwc</td>
<td>iswdigit</td>
<td>putc</td>
<td>wcsbrk</td>
</tr>
<tr>
<td>atan2</td>
<td>fputws</td>
<td>iswgraph</td>
<td>putchar</td>
<td>wcscspn</td>
</tr>
<tr>
<td>atexit</td>
<td>fread</td>
<td>iswlower</td>
<td>puts</td>
<td>wcsbspn</td>
</tr>
<tr>
<td>atof</td>
<td>free</td>
<td>iswprint</td>
<td>putwc</td>
<td>wcslen</td>
</tr>
<tr>
<td>atoi</td>
<td>freopen</td>
<td>iswpunct</td>
<td>putwchar</td>
<td>strtod</td>
</tr>
<tr>
<td>atol</td>
<td>frexp</td>
<td>iswspace</td>
<td>qsort</td>
<td>wcsncmp</td>
</tr>
<tr>
<td>bsearch</td>
<td>fscanf</td>
<td>iswupper</td>
<td>raise</td>
<td>wcscpy</td>
</tr>
<tr>
<td>bsearch</td>
<td>fseek</td>
<td>iswxdigit</td>
<td>rand</td>
<td>wcspbrk</td>
</tr>
<tr>
<td>calloc</td>
<td>fsetpos</td>
<td>isxdigit</td>
<td>realloc</td>
<td>wcsrchr</td>
</tr>
<tr>
<td>ceil</td>
<td>ftell</td>
<td>labs</td>
<td>remove</td>
<td>wcsrtombs</td>
</tr>
<tr>
<td>clearerr</td>
<td>fwrite</td>
<td>localecon</td>
<td>setbuf</td>
<td>tanh</td>
</tr>
<tr>
<td>clock</td>
<td>fprintf</td>
<td>ldiv</td>
<td>rewind</td>
<td>system</td>
</tr>
<tr>
<td>cos</td>
<td>fwrite</td>
<td>localeconv</td>
<td>scanf</td>
<td>wcstod</td>
</tr>
<tr>
<td>cosh</td>
<td>fscanf</td>
<td>localtime</td>
<td>setbuf</td>
<td>wcstok</td>
</tr>
<tr>
<td>ctime</td>
<td>getc</td>
<td>log</td>
<td>setlocale</td>
<td>time</td>
</tr>
<tr>
<td>difftime</td>
<td>getchar</td>
<td>log10</td>
<td>setvbuf</td>
<td>tmpfile</td>
</tr>
<tr>
<td>div</td>
<td>getenv</td>
<td>longjmp</td>
<td>signal</td>
<td>tmpnam</td>
</tr>
<tr>
<td>exit</td>
<td>gets</td>
<td>malloc</td>
<td>sin</td>
<td>tolower</td>
</tr>
<tr>
<td>exp</td>
<td>getwc</td>
<td>mblen</td>
<td>sinh</td>
<td>toupper</td>
</tr>
<tr>
<td>fabs</td>
<td>getwchar</td>
<td>mbstrlen</td>
<td>sprintf</td>
<td>towctrans</td>
</tr>
<tr>
<td>fclose</td>
<td>gmtime</td>
<td>mbtowc</td>
<td>sqrt</td>
<td>tolowerw</td>
</tr>
<tr>
<td>feof</td>
<td>isalnum</td>
<td>mbsinit</td>
<td>srand</td>
<td>toupperw</td>
</tr>
<tr>
<td>ferror</td>
<td>isalpha</td>
<td>mbstowcs</td>
<td>sscanf</td>
<td>ungetc</td>
</tr>
<tr>
<td>fflush</td>
<td>iscntrl</td>
<td>mbstowcs</td>
<td>strcat</td>
<td>ungetwc</td>
</tr>
<tr>
<td>fgetc</td>
<td>isdigit</td>
<td>mbtowc</td>
<td>strchr</td>
<td>vfprintf</td>
</tr>
<tr>
<td>fgetpos</td>
<td>isgraph</td>
<td>memchr</td>
<td>strcmp</td>
<td>vfwprintf</td>
</tr>
<tr>
<td>fgets</td>
<td>islower</td>
<td>memcmp</td>
<td>strcoll</td>
<td>vprintf</td>
</tr>
<tr>
<td>fgets</td>
<td>isprint</td>
<td>memcpy</td>
<td>strcpyp</td>
<td>vswprintf</td>
</tr>
<tr>
<td>fgetws</td>
<td>ispunct</td>
<td>memmove</td>
<td>strcspn</td>
<td>wscanf</td>
</tr>
<tr>
<td>floor</td>
<td>isspace</td>
<td>memset</td>
<td>strerror</td>
<td>vfprintf</td>
</tr>
</tbody>
</table>

---

Draft
C.2 Standard C library Compatibility

C.2.1 Modifications to headers

For compatibility with the Standard C library, the C++ Standard library provides the 18 C headers (D.5), but their use is deprecated in C++.

C.2.2 Modifications to definitions

C.2.2.1 Types char16_t and char32_t

The types char16_t and char32_t are distinct types rather than typedefs to existing integral types.

C.2.2.2 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 <cstddef>, <cstdlib>, or <cwchar> (21.4).

C.2.2.3 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.2.2.4 Macro NULL

The macro NULL, defined in any of <locale>, <cstdlib>, <cstdio>, <cstddl1ib>, <cstring>, <ctime>, or <cwchar>, is an implementation-defined C++ null pointer constant in this International Standard (18.1).

C.2.3 Modifications to declarations

Header <cstring>: The following functions have different declarations:

- strchr
- strpbrk
- strrchr
- strstr
- memchr

21.4 describes the changes.

C.2.4 Modifications to behavior

Header <cstddl1ib>: The following functions have different behavior:

- atexit
- exit
- abort

18.4 describes the changes.

Header <csetjmp>: The following functions have different behavior:
— `longjmp`

18.8 describes the changes.

**C.2.4.1 Macro `offsetof(type,member-designator)`**

The macro `offsetof`, defined in `<cstddef>`, accepts a restricted set of `type` arguments in this International Standard. 18.1 describes the change.

**C.2.4.2 Memory allocation functions**

The functions `calloc`, `malloc`, and `realloc` are restricted in this International Standard. 20.6.8 describes the changes.
Appendix D (normative)
Compatibility features

This clause describes features of the C++ Standard that are specified for compatibility with existing implementations.

These are deprecated features, where deprecated is defined as: Normative for the current edition of the Standard, but not guaranteed to be part of the Standard in future revisions.

D.1 Increment operator with bool operand
The use of an operand of type bool with the ++ operator is deprecated (see 5.3.2 and 5.2.6).

D.2 static keyword
The use of the static keyword is deprecated when declaring objects in namespace scope (see 3.3.5).

D.3 Access declarations
Access declarations are deprecated (see 11.3).

D.4 Implicit conversion from const strings
The implicit conversion from const to non-const qualification for string literals (4.2) is deprecated.

D.5 Standard C library headers
For compatibility with the Standard C library and the C Unicode TR, the C++ Standard library provides the 25 C headers, as shown in Table 136.

| <assert.h> | <float.h> | <math.h> | <stdbool.h> | <tgmath.h> |
| <complex.h> | <inttypes.h> | <setjmp.h> | <stdio.h> | <time.h> |
| <ctype.h> | <iso646.h> | <signal.h> | <stdlib.h> | <uchar.h> |
| <errno.h> | <limits.h> | <stdarg.h> | <stdlib.h> | <wchar.h> |
| <fenv.h> | <locale.h> | <stdbool.h> | <string.h> | <wctype.h> |

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 (3.3.5) of the namespace \texttt{std} and are then injected into the global namespace scope by explicit using-declarations (7.3.3).

[Example: The header \texttt{<cstdlib>} assuredly provides its declarations and definitions within the namespace \texttt{std}. It may also provide these names within the global namespace. The header \texttt{<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 \texttt{std}. — end example]

D.6 Old iostreams members

The following member names are in addition to names specified in clause 27:

namespace \texttt{std} {
  class \texttt{ios\_base} {
    public:
      typedef \texttt{T1} \texttt{io\_state};
      typedef \texttt{T2} open_mode;
      typedef \texttt{T3} seek_dir;
      typedef \texttt{OFF\_T} streamoff;
      typedef \texttt{POS\_T} streampos;
      // remainder unchanged
  };
}

The type \texttt{io\_state} is a synonym for an integer type (indicated here as \texttt{T1}) that permits certain member functions to overload others on parameters of type \texttt{iostate} and provide the same behavior.

The type \texttt{open\_mode} is a synonym for an integer type (indicated here as \texttt{T2}) that permits certain member functions to overload others on parameters of type \texttt{openmode} and provide the same behavior.

The type \texttt{seek\_dir} is a synonym for an integer type (indicated here as \texttt{T3}) that permits certain member functions to overload others on parameters of type \texttt{seekdir} and provide the same behavior.

The type \texttt{streamoff} is an implementation-defined type that satisfies the requirements of type \texttt{OFF\_T} (27.4.1).

The type \texttt{streampos} is an implementation-defined type that satisfies the requirements of type \texttt{POS\_T} (27.2).

An implementation may provide the following additional member function, which has the effect of calling \texttt{sbumpc()} (27.5.2.2.3):

namespace \texttt{std} {
  template<class charT, class traits = char_traits<charT> >
  class basic_streambuf {
    public:
      void stossc();
      // remainder unchanged
  };
}

An implementation may provide the following member functions that overload signatures specified in clause 27:
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
        }
    class ios_base {
        public:
            // remainder unchanged
    }
    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
    }
}
The effects of these functions is to call the corresponding member function specified in clause 27.

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

```cpp
namespace std {
    class strstreambuf : public basic_streambuf<char> {
        public:
            explicit strstreambuf(streamsize alsizet = 0);
            strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
            strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = 0);
            strstreambuf(const char* gnext_arg, streamsize n);
            strstreambuf(signed char* gnext_arg, streamsize n, signed char* pbegin_arg = 0);
            strstreambuf(const signed char* gnext_arg, streamsize n);
            strstreambuf(unsigned char* gnext_arg, streamsize n, unsigned char* pbegin_arg = 0);
            strstreambuf(const unsigned char* gnext_arg, streamsize n);
            virtual ~strstreambuf();
            void freeze(bool freeze = true);
            char* str();
            int pcount();

        protected:
            virtual int_type overflow (int_type c = EOF);
            virtual int_type pbackfail(int_type c = EOF);
            virtual int_type underflow();
            virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                ios_base::openmode which = ios_base::in | ios_base::out);
            virtual pos_type seekpos(pos_type sp, ios_base::openmode which
                = ios_base::in | ios_base::out);
            virtual streambuf* setbuf(char* s, streamsize n);

        private:
            // typedef T1 strstate;
            // static const strstate allocated;
            // static const strstate constant;
            // static const strstate dynamic;
            // static const strstate frozen;
            // strstate strmode;
            // streamsize alsizet;
            // void* (*palloc)(size_t);
            // void (*pfree)(void*);
```

---

Draft
The class `strstreambuf` associates the input sequence, and possibly the output sequence, with an object of some `character` array type, whose elements store arbitrary values. The array object has several attributes.

[Note: For the sake of exposition, these are represented as elements of a bitmask type (indicated here as `T1`) called `strstate`. The elements are:

- `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 ]

[Note: 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 alsiz`, the suggested minimum size for a dynamic array object;
- `void* palloc(size_t)`, points to the function to call to allocate a dynamic array object;
- `void (*pfree)(void*)`, points to the function to call to free a dynamic array object.

— end note ]

Each object of class `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

```cpp
explicit strstreambuf(streamsize alsiz_arg = 0);
```

*Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 137.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strmode</code></td>
<td><code>dynamic</code></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>

Draft
D.7 char* streams

strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));

Effects: Constructs an object of class strstreambuf, initializing the base class with streambuf(). The post-conditions of this function are indicated in Table 138.

Table 138: strstreambuf(void* (*)(size_t), void (*)(void*)) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>strmode</td>
<td>dynamic</td>
</tr>
<tr>
<td>alsize</td>
<td>an unspecified value</td>
</tr>
<tr>
<td>palloc</td>
<td>palloc_arg</td>
</tr>
<tr>
<td>pfree</td>
<td>pfree_arg</td>
</tr>
</tbody>
</table>

strstreambuf(char* gnext_arg, streamsize n, char *pbeg_arg = 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 post-conditions of this function are indicated in Table 139.

Table 139: strstreambuf(charT*, streamsize, charT*) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>strmode</td>
<td>0</td>
</tr>
<tr>
<td>alsize</td>
<td>an unspecified value</td>
</tr>
<tr>
<td>palloc</td>
<td>a null pointer</td>
</tr>
<tr>
<td>pfree</td>
<td>a null pointer</td>
</tr>
</tbody>
</table>

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 std::strlen(gnext_arg).
- If n < 0, N is INT_MAX.\(^{334}\)

If pbeg_arg is a null pointer, the function executes:

setg(gnext_arg, gnext_arg, gnext_arg + N);

Otherwise, the function executes:

setg(gnext_arg, gnext_arg, pbeg_arg);
setp(pbeg_arg, pbeg_arg + N);
strstreambuf(const char* gnext_arg, streamsize n);

\(^{334}\) The function signature strlen(const char*) is declared in <cstring> (21.4). The macro INT_MAX is defined in <climits> (18.2).
D.7.1.2 Member functions

```c
void freeze(bool freezefl = true);
```

*Effects:* If `strmode & dynamic` is non-zero, alters the freeze status of the dynamic array object as follows:
- If `freezefl` is true, the function sets `frozen` in `strmode`.
- Otherwise, it clears `frozen` in `strmode`.

```c
char* str();
```

*Effects:* Calls `freeze()`, then returns the beginning pointer for the input sequence, `gbeg`.

*Remarks:* The return value can be a null pointer.

```c
int pcount() const;
```

*Effects:* If the next pointer for the output sequence, `pnext`, is a null pointer, returns zero. Otherwise, returns the current effective length of the array object as the next pointer minus the beginning pointer for the output sequence, `pnext - pbeg`.

D.7.1.3 `strstreambuf` overridden virtual functions

```c
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:
- If `c != EOF` and if either the output sequence has a write position available or the function makes a write position available (as described below), assigns `c` to `*pnext++`.
  - Returns `(unsigned char)c`.
- If `c == EOF`, there is no character to append.
  - Returns a value other than EOF.
  - Returns EOF to indicate failure.

*Remarks:* The function can alter the number of write positions available as a result of any call.
To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements \( n \) to hold the current array object (if any), plus at least one additional write position. How many additional write positions are made available is otherwise unspecified.\(^\text{335)}\) If \( \text{palloc} \) is not a null pointer, the function calls \((\text{palloc})(n)\) to allocate the new dynamic array object. Otherwise, it evaluates the expression \text{new} \ charT[\( n \)]. In either case, if the allocation fails, the function returns EOF. Otherwise, it sets \( \text{allocated} \) in \text{strmode}.

To free a previously existing dynamic array object whose first element address is \( p \): If \( \text{pfree} \) is not a null pointer, the function calls \((\text{pfree})(p)\). Otherwise, it evaluates the expression \text{delete}[] \( p \).

If \( \text{strmode} \& \text{dynamic} == 0 \), or if \( \text{strmode} \& \text{frozen} != 0 \), the function cannot extend the array (reallocate it with greater length) to make a write position available.

\begin{verbatim}
int_type pbackfail(int_type c = EOF);
\end{verbatim}

Puts back the character designated by \( c \) to the input sequence, if possible, in one of three ways:

\begin{itemize}
  \item If \( c \neq \text{EOF} \), if the input sequence has a putback position available, and if \((\text{char})c == \text{gnext}[-1]\), assigns \( \text{gnext} - 1 \) to \( \text{gnext} \).
    \begin{verbatim}
    Returns c.
    \end{verbatim}
  \item If \( c \neq \text{EOF} \), if the input sequence has a putback position available, and if \( \text{strmode} \& \text{constant} \) is zero, assigns \( c \) to \(*--\text{gnext} \).
    \begin{verbatim}
    Returns c.
    \end{verbatim}
  \item If \( c == \text{EOF} \) and if the input sequence has a putback position available, assigns \( \text{gnext} - 1 \) to \( \text{gnext} \).
    \begin{verbatim}
    Returns a value other than EOF.
    \end{verbatim}
\end{itemize}

Returns EOF to indicate failure.

Remarks: If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

\begin{verbatim}
int_type underflow();
\end{verbatim}

Effects: Reads a character from the \textit{input sequence}, if possible, without moving the stream position past it, as follows:

\begin{itemize}
  \item If the input sequence has a read position available, the function signals success by returning \((\text{unsigned char})*\text{gnext} \).
    \begin{verbatim}
    Returns (unsigned char*)\text{gnext}.
    \end{verbatim}
  \item Otherwise, if the current write next pointer \( \text{pnext} \) is not a null pointer and is greater than the current read end pointer \( \text{gend} \), makes a \textit{read position} available by assigning to \( \text{gend} \) a value greater than \( \text{gnext} \) and no greater than \( \text{pnext} \).
    \begin{verbatim}
    Returns (unsigned char*)\text{gnext}.
    \end{verbatim}
\end{itemize}

Returns EOF to indicate failure.

Remarks: The function can alter the number of read positions available as a result of any call.

\(^{335)}\) An implementation should consider \textit{alsize} in making this decision.
pos_type seekoff(off_type off, seekdir way, openmode which = in | out);

Effects: Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 140.

Table 140: 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>
</tbody>
</table>

| Otherwise | 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 as indicated in Table 141.

Table 141: 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>
<tr>
<td>If (newoff + off) &lt; (seeklow - xbeg), or (seekhigh - xbeg) &lt; (newoff + off)</td>
<td>the positioning operation fails</td>
</tr>
</tbody>
</table>

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)).

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).

— If (which & ios::in) != 0, positions the input sequence.
— If (which & ios::out) != 0, positions the output sequence.
— 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():

— If newoff is an invalid stream position, has a negative value, or has a value greater than (seekhigh - seeklow), the positioning operation fails

— Otherwise, the function adds newoff to the beginning pointer xbeg and stores the result in the next pointer xnext.

Returns: pos_type(newoff), constructed from the resultant offset newoff (of type off_type), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is pos_type(off_type(-1)).

streambuf<char>* setbuf(char* s, streamsize n);

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

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:

— sb, the strstreambuf object.

D.7.2.1 istrstream constructors

explicit istrstream(const char* s);
explicit istrstream(char* s);

Effects: Constructs an object of class istrstream, initializing the base class with istream(sb) and initializing sb with strstreambuf(s,0). s shall designate the first element of an NTBS.
istrstream(const char* s, streamsize n);

Effects: Constructs an object of class istrstream, initializing the base class with istream(&sb) and initializing sb with strstreambuf(s,n)). s shall designate the first element of an array whose length is n elements, and n shall be greater than zero.

D.7.2.2 Member functions

strstreambuf* rdbuf() const;

Returns: const_cast<strstreambuf*>(&sb).

char* str();

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
        };
    }
}

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:

— sb, the strstreambuf object.

D.7.3.1 ostrstream constructors

ostrstream();

Effects: Constructs an object of class ostrstream, initializing the base class with ostream(&sb) and initializing sb with strstreambuf()).

ostrstream(char* s, int n, ios_base::openmode mode = ios_base::out);
D.7.3.2 Member functions

strstreambuf* rdbuf() const;

*Effects:* Returns: (strstreambuf*)&sb.

void freeze(bool freezefl = true);

*Effects:* Calls rdbuf()->freeze(freezefl).

char* str();

*Returns:* rdbuf()->str().

int pcount() const;

*Returns:* rdbuf()->pcount().
The class `strstream` supports reading and writing from objects of class `strstreambuf`. It supplies a `strstreambuf` object to control the associated array object. For the sake of exposition, the maintained data is presented here as

— `sb`, the `strstreambuf` object.

### D.7.4.1 strstream constructors

*strstream();*

*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);*

*Effects:* Constructs an object of class `strstream`, initializing the base class with `iostream(&sb)` and initializing `sb` with one of the two constructors:

— If `(mode & app) == 0`, then `s` shall designate the first element of an array of `n` elements. The constructor is `strstreambuf(s,n,s)`.

— 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();*

*Effects:* Destroys an object of class `strstream`.

*strstreambuf* rdbuf() const;

*Returns:* `&sb`.

### D.7.4.3 strstream operations

*void freeze(bool freezefl = true);*

*Effects:* Calls `rdbuf()->freeze(freezefl)`.

*char* str();*
D.8 Binders

Returns: rdbuf()->str().

int pcount() const;

Returns: rdbuf()->pcount().

D.8 Binders [depr.lib.binders]

The binders binder1st, bind1st, binder2nd, and bind2nd are deprecated. [Note: The template function bind (20.5.10) provides a better solution. —end note]

D.8.1 Class template binder1st [depr.lib.bind.1st]

template <class Fn>
class binder1st
  : public unary_function<typename Fn::second_argument_type,  
    typename Fn::result_type> {
protected:
  Fn fn;
  typename Fn::first_argument_type value;
public:
  binder1st(const Fn& x,  
    const typename Fn::first_argument_type& y);
  typename Fn::result_type  
    operator()(const typename Fn::second_argument_type& x) const;
  typename Fn::result_type  
    operator()(typename Fn::second_argument_type& x) const;
};

The constructor initializes fn with x and value with y.

operator() returns fn(value, x).

D.8.2 bind1st [depr.lib.bind.1st]

template <class Fn, class T>
  binder1st<Fn> bind1st(const Fn& fn, const T& z);

Returns: binder1st<Fn>(fn, typename Fn::first_argument_type(z)).

D.8.3 Class template binder2nd [depr.lib.bind.2nd]

template <class Fn>
class binder2nd
  : public unary_function<typename Fn::first_argument_type,  
    typename Fn::result_type> {
protected:
  Fn fn;
  typename Fn::second_argument_type value;

Draft
public:
    binder2nd(const Fn& x,
              const typename Fn::second_argument_type& y);
    typename Fn::result_type
        operator()(const typename Fn::first_argument_type& x) const;
    typename Fn::result_type
        operator()(typename Fn::first_argument_type& x) const;
};


1. The constructor initializes fn with x and value with y.
2. operator() returns fn(x, value).

D.8.4 bind2nd

[depr.lib.bind.2nd]

template <class Fn, class T>
binder2nd<Fn> bind2nd(const Fn& op, const T& x);

1. Returns: binder2nd<Fn>(op, typename Fn::second_argument_type(x)).
2. [Example:
    find_if(v.begin(), v.end(), bind2nd(greater<int>(), 5));

    finds the first integer in vector v greater than 5;
    find_if(v.begin(), v.end(), bind1st(greater<int>(), 5));

    finds the first integer in v less than 5. —end example ]

D.9 auto_ptr

[depr.auto.ptr]

The class template auto_ptr is deprecated. [Note: The class template unique_ptr (20.6.5) provides a better solution. —end note]

D.9.1 Class template auto_ptr

[auto.ptr]

1. Template auto_ptr stores a pointer to an object obtained via new and deletes that object when it itself is destroyed (such as when leaving block scope 6.7).
2. Template auto_ptr_ref holds a reference to an auto_ptr. It is used by the auto_ptr conversions to allow auto_ptr objects to be passed to and returned from functions.

namespace std {
    template <class Y> struct auto_ptr_ref {};

    template <class X> class auto_ptr {
        public:
            typedef X element_type;

            // D.9.1.1 construct/copy/destroy:
            explicit auto_ptr(X* p =0) throw();

Draft
auto_ptr(auto_ptr&) throw();
template<class Y> auto_ptr(auto_ptr<Y>&) throw();
auto_ptr& operator=(auto_ptr&) throw();
template<class Y> auto_ptr& operator=(auto_ptr<Y>&) throw();
auto_ptr& operator=(auto_ptr_ref<X> r) throw();
~auto_ptr() throw();

// D.9.1.2 members:
X& operator*() const throw();
X* operator->() const throw();
X* get() const throw();
X* release() throw();
void reset(X* p = 0) throw();

// D.9.1.3 conversions:
auto_ptr(auto_ptr_ref<X>) throw();
template<class Y> operator auto_ptr_ref<Y>() throw();
template<class Y> operator auto_ptr<Y>() throw();
};
template <> class auto_ptr<void>
{
public:
    typedef void element_type;
};

3 The auto_ptr provides a semantics of strict ownership. An auto_ptr owns the object it holds a pointer to. Copying an auto_ptr copies the pointer and transfers ownership to the destination. If more than one auto_ptr owns the same object at the same time the behavior of the program is undefined. [ Note: The uses of auto_ptr include providing temporary exception-safety for dynamically allocated memory, passing ownership of dynamically allocated memory to a function, and returning dynamically allocated memory from a function. auto_ptr does not meet the CopyConstructible and Assignable requirements for Standard Library container elements and thus instantiating a Standard Library container with an auto_ptr results in undefined behavior. — end note ]

D.9.1.1 auto_ptr constructors [auto.ptr.cons]

explicit auto_ptr(X* p = 0) throw();

Postconditions: *this holds the pointer p.

auto_ptr(auto_ptr& a) throw();

Effects: Calls a.release().

Postconditions: *this holds the pointer returned from a.release().

template<class Y> auto_ptr(auto_ptr<Y>& a) throw();

Requires: Y* can be implicitly converted to X*.
auto_ptr& operator=(auto_ptr& a) throw();

Requires: The expression delete get() is well formed.
Effects: reset(a.release())
Returns: *this.

auto_ptr& operator=(auto_ptr<Y>& a) throw();

Requires: Y* can be implicitly converted to X*. The expression delete get() is well formed.
Effects: reset(a.release())
Returns: *this.

auto_ptr() throw();

Requires: The expression delete get() is well formed.
Effects: delete get().

D.9.1.2 auto_ptr members

X& operator*() const throw();

Requires: get() != 0
Returns: *get()

X* operator->() const throw();

Returns: get()

X* get() const throw();

Returns: The pointer *this holds.

X* release() throw();

Returns: get()
Postcondition: *this holds the null pointer.

void reset(X* p=0) throw();

Effects: If get() != p then delete get().
Postconditions: *this holds the pointer p.
D.9.1.3 auto_ptr conversions

```cpp
auto_ptr(auto_ptr_ref<X> r) throw();
```

1. **Effects:** Calls `p.release()` for the `auto_ptr p` that `r` holds.
2. **Postconditions:** `*this` holds the pointer returned from `release()`.

```cpp
template<class Y> operator auto_ptr_ref<Y>() throw();
```

3. **Returns:** An `auto_ptr_ref<Y>` that holds `*this`.

```cpp
template<class Y> operator auto_ptr<Y>() throw();
```

4. **Effects:** Calls `release()`.
5. **Returns:** An `auto_ptr<Y>` that holds the pointer returned from `release()`.

```cpp
auto_ptr& operator=(auto_ptr_ref<X> r) throw()
```

6. **Effects:** Calls `reset(p.release())` for the `auto_ptr p` that `r` holds a reference to.
7. **Returns:** `*this`
Appendix E  
(normative)  
Universal-character-names  
[extendid]

This clause lists the complete set of hexadecimal code values that are valid in universal-character-names in C++ identifiers (2.10).

This table is reproduced unchanged from ISO/IEC PDTR 10176, produced by ISO/IEC JTC1/SC22/WG20, except that the ranges 0041–005a and 0061–007a designate the upper and lower case English alphabets, which are part of the basic source character set, and are not repeated in the table below.

Latin: 00c0–00d6, 00d8–00f6, 00f8–01f5, 01fa–0217, 0250–02a8, 1e00–1e9a, 1ea0–1ef9
Greek: 0384, 0388–038a, 038c, 038e–03a1, 03a3–03ce, 03d0–03d6, 03da, 03dc, 03de, 03e0, 03e2–03f3, 1f00–1f15, 1f18–1f1d, 1f20–1f45, 1f48–1f4d, 1f50–1f57, 1f59, 1f5b, 1f5d, 1f5f–1f7d, 1f80–1f84, 1f86–1fbc, 1fc2–1fc4, 1fc6–1fcc, 1fd0–1fd3, 1fd6–1fdb, 1fe0–1fec, 1ff2–1ff4, 1ff6–1ffc
Cyrillic: 0401–040d, 040f–044f, 0451–045c, 045e–0481, 0490–04c4, 04c7–04c8, 04cb–04cc, 04d0–04eb, 04ee–04f5, 04f8–04f9
Armenian: 0531–0556, 0561–0587
Hebrew: 05d0–05ea, 05f0–05f4
Arabic: 0621–063a, 0640–0652, 0670–06b7, 06ba–06be, 06c0–06ce, 06e5–06e7
Devanagari: 0905–0939, 0958–0962
Bengali: 0985–098c, 098f–0990, 0993–09a8, 09aa–09b0, 09b2, 09b6–09b9, 09dc–09dd, 09df–09e1, 09f0–09f1
Gurmukhi: 0a05–0a0a, 0a0f–0a10, 0a13–0a28, 0a2a–0a30, 0a32–0a33, 0a35–0a36, 0a38–0a39, 0a59–0a5c, 0a5e
Gujarati: 0a85–0a8b, 0a8d, 0a8f–0a91, 0a93–0a98, 0aaa–0ab0, 0ab2–0ab3, 0ab5–0ab9, 0ae0
Oriya: 0b05–0b0c, 0b0f–0b10, 0b13–0b28, 0b2a–0b30, 0b32–0b33, 0b36–0b39, 0b5c–0b5d, 0b5f–0b61
Tamil: 0b85–0b8a, 0b8e–0b90, 0b92–0b95, 0b99–0b9a, 0bc9, 0b9e–0b9f, 0ba3–0ba4, 0ba8–0baa, 0bae–0bb5, 0bb7–0bb9
Telugu: 0c05–0c0c, 0c0e–0c10, 0c12–0c28, 0c2a–0c33, 0c35–0c39, 0c60–0c61
Kannada: 0c85–0c8c, 0c8e–0c90, 0c92–0ca8, 0caa–0cb3, 0cb5–0cb9, 0ce0–0ce1
Malayalam: 0d05–0d0c, 0d0e–0d10, 0d12–0d28, 0d2a–0d39, 0d60–0d61
Thai: 0e01–0e30, 0e32–0e33, 0e40–0e46, 0e4f–0e5b
Lao: 0e81–0e82, 0e84, 0e87, 0e88, 0e8a, 0e8d, 0e94–0e97, 0e99–0e9f, 0ea1–0ea3, 0ea5, 0ea7, 0eaa, 0eab, 0ead–0e0b0, 0eb2, 0eb3, 0ebd, 0ec0–0ec4, 0ec6
Georgian: 10a0–10c5, 10d0–10f6
Hiragana: 3041–3094, 309b–309e
Katakana: 30a1–30fe
Bopomofo: 3105–312c
Hangul: 1100–1159, 1161–11a2, 11a8–11f9
CJK Unified Ideographs: f900–fa2d, fb1f–fb36, fb38–fb3c, fb3e, fb40–fb41, fb42–fb44, fb46–fbb1, fb3d–fd3f, fd50–fd8f, fd92–fdec, fdf0–fddf, fe70–fe72, fe74, fe76–fefe, ff21–ff3a, ff41–ff5a, ff66–ffbe, ffca–ffcf, ffd2–ff77, ffd9–ffdc, 4e00–9fa5
Appendix F
(informative)
Cross references

To refer to sections in the text, use section labels, which appear at the beginning of every section on the right-hand side of the page. While section numbers appear more natural, they change from time to time as material is added and removed from the draft.

This appendix lists each section label and the corresponding section number, in alphabetical order by label. It will be revised with each new draft, so can be used to find where a section is located in the current draft.

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 Appendix B has been changed from `[limits]` to `[implimits]`. The label `[limits]` refers to section 18.2.1.

<table>
<thead>
<tr>
<th>A</th>
<th>alg.accumulate</th>
<th>26.6.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjacent.difference</td>
<td>26.6.4</td>
<td></td>
</tr>
<tr>
<td>adjustfield.manip</td>
<td>27.4.5.2</td>
<td></td>
</tr>
<tr>
<td>alg.adjacent.find</td>
<td>25.1.5</td>
<td></td>
</tr>
<tr>
<td>alg.binary.search</td>
<td>25.3.3</td>
<td></td>
</tr>
<tr>
<td>alg.c.library</td>
<td>25.4</td>
<td></td>
</tr>
<tr>
<td>alg.copy</td>
<td>25.2.1</td>
<td></td>
</tr>
<tr>
<td>alg.count</td>
<td>25.1.6</td>
<td></td>
</tr>
<tr>
<td>alg.equal</td>
<td>25.1.8</td>
<td></td>
</tr>
<tr>
<td>alg.fill</td>
<td>25.2.6</td>
<td></td>
</tr>
<tr>
<td>alg.find</td>
<td>25.1.2</td>
<td></td>
</tr>
<tr>
<td>alg.find.end</td>
<td>25.1.3</td>
<td></td>
</tr>
<tr>
<td>alg.find.first.of</td>
<td>25.1.4</td>
<td></td>
</tr>
<tr>
<td>alg.foreach</td>
<td>25.1.1</td>
<td></td>
</tr>
<tr>
<td>alg.generate</td>
<td>25.2.7</td>
<td></td>
</tr>
<tr>
<td>alg.heap.operations</td>
<td>25.3.6</td>
<td></td>
</tr>
<tr>
<td>alg.lex.comparison</td>
<td>25.3.8</td>
<td></td>
</tr>
<tr>
<td>alg.merge</td>
<td>25.3.4</td>
<td></td>
</tr>
<tr>
<td>alg.min.max</td>
<td>25.3.7</td>
<td></td>
</tr>
<tr>
<td>alg.modifying.operations</td>
<td>25.2</td>
<td></td>
</tr>
<tr>
<td>alg.move</td>
<td>25.2.2</td>
<td></td>
</tr>
<tr>
<td>alg.nonmodifying</td>
<td>25.1</td>
<td></td>
</tr>
<tr>
<td>alg.nth.element</td>
<td>25.3.2</td>
<td></td>
</tr>
<tr>
<td>alg.partitions</td>
<td>25.2.13</td>
<td></td>
</tr>
<tr>
<td>alg.permutation.generators</td>
<td>25.3.9</td>
<td></td>
</tr>
<tr>
<td>alg.random.shuffle</td>
<td>25.2.12</td>
<td></td>
</tr>
<tr>
<td>alg.remove</td>
<td>25.2.8</td>
<td></td>
</tr>
<tr>
<td>alg.replace</td>
<td>25.2.5</td>
<td></td>
</tr>
<tr>
<td>alg.reverse</td>
<td>25.2.10</td>
<td></td>
</tr>
<tr>
<td>alg.rotate</td>
<td>25.2.11</td>
<td></td>
</tr>
<tr>
<td>alg.search</td>
<td>25.1.9</td>
<td></td>
</tr>
<tr>
<td>alg.set.operations</td>
<td>25.3.5</td>
<td></td>
</tr>
<tr>
<td>alg.sort</td>
<td>25.3.1</td>
<td></td>
</tr>
</tbody>
</table>
alg.sorting  25.3
alg.swap    25.2.3
alg.transform 25.2.4
alg.unique   25.2.9
algorithms  25
alloc.errors 18.5.2
allocator.globals 20.6.1.2
allocator.members 20.6.1.1
allocator.requirements 20.1.2
alt.headers  17.4.3.2
arithmetic.operations 20.5.6
array 23.2.1
array.cons   23.2.1.1
array.data   23.2.1.4
array.size   23.2.1.3
array.special 23.2.1.2
array.tuple   23.2.1.6
array.zero    23.2.1.5
assertions  19.2
associative  23.3
associative.reqmts 23.1.2
auto.ptr D.9.1
auto.ptr.cons D.9.1.1
auto.ptr.conv D.9.1.3
auto.ptr.members D.9.1.2

basic.fundamental 3.9.1
basic.fundamental 3.9.1
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.class 3.3.6
basic.scope.enum 3.3.7
basic.scope.inheritance 3.3.8
basic.scope.local 3.3.2
basic.scope.namespace 3.3.5
basic.scope.pdecl 3.3.1
basic.scope.proto 3.3.3
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 3.7.2
basic.stc.dynamic 3.7.3
basic.stc.dynamic.allocation 3.7.3.1
basic.stc.dynamic.deallocation 3.7.3.2
basic.stc.inheritance 3.7.4
basic.stc.static 3.7.1
basic.string 21.3
basic.type.qualifier 3.9.3
basic.types 3.9
bidirectional.iterators 24.1.4
binary.search 25.3.3.4
bind 20.5.10
bitset.cons 23.3.5.1
bitset.members 23.3.5.2
bitset.operators 23.3.5.3
byte.strings 17.3.2.1.3.1
C

c.files 27.8.2
c.limits 18.2.2
c.locals 22.3
c.malloc 20.6.8
c.math 26.7
c.strings 21.4

category.collate 22.2.4
category.ctype 22.2.1
category.messages 22.2.7
category.monetary 22.2.6
category.numeric 22.2.2
category.time 22.2.5

cmplx 26.3.10
cfenv 26.2
cfenv.syn 26.2.1
char.traits 21.1
char.traits.require 21.1.1
char.traits.specializations 21.1.3
char.traits.specializations.char 21.1.3.1
char.traits.specializations.char16_t 21.1.3.2
char.traits.specializations.char32_t 21.1.3.3
char.traits.specializations.wchar.t 21.1.3.4
char.traits.typedefs 21.1.2
char16_t.seq 17.3.2.1.3.3
char32_t.seq 17.3.2.1.3.4
character.seq 17.3.2.1.3
class 9
class.abstract 10.4
class.access 11
class.access.base 11.2
class.access.dcl 11.3
class.access.nest 11.8
class.access.spec 11.1
class.access.virt 11.6
class.base.init 12.6.2
class.bit 9.6
class.editor 12.7
class.conv 12.3
class.conv.ctor 12.3.1
class.conv.fct 12.3.2
class.copy 12.8
class.ctor 12.1
class.derived 10
class.dtor 12.4
class.expl.init 12.6.1
class.free 12.5
class.friend 11.4
class.gslice 26.5.6
class.init 12.6
class.local 9.8
class.mem 9.2
class.member.lookup 10.2
class.mfct 9.3
class.mfct.non-static 9.3.1
class.mi 10.1
class.name 9.1
class.nest 9.7
class.nested.type 9.9
class.paths 11.7
class.protected 11.5
class.qual 3.4.3.1
class.slice 26.5.4
class.static 9.4
class.static.data 9.4.2
class.static.mfct 9.4.1
class.temporary 12.2
class.this 9.3.2
class.union 9.5
class.virtual 10.3
classification 22.1.3.1
cmplx.over 26.3.9
cmplxh 26.3.11
comparisons 20.5.7
complex 26.3.2
complex.member.ops 26.3.5
complex.members 26.3.4
complex.numbers 26.3
complex.ops 26.3.6
complex.special 26.3.3
complex.synopsis 26.3.1
complex.transcendentals 26.3.8
complex.value.ops 26.3.7
compliance 17.4.1.3
conforming 17.4.4
cons.slice 26.5.4.1
cons.slice.arr 26.5.5.1
constraints 17.4.3
container.adaptors 23.2.4
container.requirements 23.1
containers 23
except  15
except.access  15.6
except.ctor  15.2
except.handle  15.3
except.spec  15.4
except.special  15.5
except.terinate  15.5.1
except.throw  15.1
except.uncaught  15.5.3
except.unexpected  15.5.2
exception  18.7.1
exception.terinate  18.7.3
exception.unexpected  18.7.2
expr  5
expr.add  5.7
expr.alignof  5.3.6
expr.ass  5.17
expr.bit.and  5.11
expr.call  5.2.2
expr.cast  5.4
expr.comma  5.18
expr.cond  5.16
expr.const  5.19
expr.const.cast  5.2.11
expr.delete  5.3.5
expr.dynamic.cast  5.2.7
expr.eq  5.10
expr.log.and  5.14
expr.log.or  5.15
expr.mptr.oper  5.5
expr.mul  5.6
expr.new  5.3.4
expr.or  5.13
expr.post  5.2
expr.post.incr  5.2.6
expr.pre.incr  5.3.2
expr.prim  5.1
expr.pseudo  5.2.4
expr.ref  5.2.5
expr.reinterpret.cast  5.2.10
expr.rel  5.9
expr.shift  5.8
expr.sizeof  5.3.3
expr.static.cast  5.2.9
expr.sub  5.2.1
expr.type.conv  5.2.3
expr.typeid  5.2.8
expr.unary  5.3
expr.unary.op  5.3.1
expr.xor  5.12
ext.manip  27.6.4
extendid  E
extern.names  17.4.3.1.3
extern.types  17.4.3.1.4

F
facet.ctype.char.dtor  22.2.1.3.1
facet.ctype.char.members  22.2.1.3.2
facet.ctype.char.statics  22.2.1.3.3
facet.ctype.char.virtuals  22.2.1.3.4
facet.ctype.special  22.2.1.3
facet.num.get.members  22.2.2.1.1
facet.num.get.virtuals  22.2.2.1.2
facet.num.put.members  22.2.2.2.1
facet.num.put.virtuals  22.2.2.2.2
facet.numpunct  22.2.3
facet.numpunct.members  22.2.3.1.1
facet.numpunct.virtuals  22.2.3.1.2
facets.examples  22.2.8
fenv  26.2.2
file.streams  27.8
filebuf  27.8.1.1
filebuf.assign  27.8.1.3
filebuf.cons  27.8.1.2
filebuf.members  27.8.1.4
filebuf.virtuals  27.8.1.5
floatfield.manip  27.4.5.4
fmtflags.manip  27.4.5.1
fmtflags.state  27.4.2.2
forward  20.2.2
forward.iterators  24.1.3
fpos  27.4.3
fpos.members  27.4.3.1
fpos.operations  27.4.3.2
front.insert.iter.cons  24.4.2.4.1
front.insert.iter.op*  24.4.2.4.3
front.insert.iter.op++  24.4.2.4.4
front.insert.iter.op=  24.4.2.4.2
front.insert.iter.ops  24.4.2.4
front.insert.iterator  24.4.2.3
front inserter  24.4.2.4.5

Draft
Cross references

fstream 27.8.1.14
fstream.assign 27.8.1.16
fstream.cons 27.8.1.15
fstream.members 27.8.1.17
fstreams 27.8.1
func.bind 20.5.10.1
func.bind.bind 20.5.10.1.3
func.bind.isbind 20.5.10.1.1
func.bind.isplace 20.5.10.1.2
func.bind.place 20.5.10.1.4
func.def 20.5.1
func.memfn 20.5.13
func.require 20.5.2
func.ret 20.5.4
func.wrap 20.5.14
func.wrap.badcall 20.5.14.1
func.wrap.badcall.const 20.5.14.1.1
func.wrap.func 20.5.14.2
func.wrap.func.alg 20.5.14.2.8
func.wrap.func.cap 20.5.14.2.3
func.wrap.func.con 20.5.14.2.1
func.wrap.func.inv 20.5.14.2.4
func.wrap.func.mod 20.5.14.2.2
func.wrap.func.nullptr 20.5.14.2.7
func.wrap.func.targ 20.5.14.2.5
func.wrap.func.undef 20.5.14.2.6
function.objects 20.5
function.pointer.adaptors 20.5.11
functions.within.classes 17.3.2.2

G

global.functions 17.4.4.3
global.names 17.4.4.3.2
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.smtp A.5
gram.temp A.12
gslice.access 26.5.6.2
gslice.array.assign 26.5.7.2
gslice.array.comp.assign 26.5.7.3
gslice.array.cons 26.5.7.1
gslice.array.fill 26.5.7.4
gslice.cons 26.5.6.1

H

handler.functions 17.4.3.5
headers 17.4.1.2

I

ifstream 27.8.1.6
ifstream.assign 27.8.1.8
ifstream.cons 27.8.1.7
ifstream.members 27.8.1.9
implimits B
includes 25.3.5.1
indirect.array.assign 26.5.9.2
indirect.array.comp.assign 26.5.9.3
indirect.array.cons 26.5.9.1
indirect.array.fill 26.5.9.4
inner.product 26.6.2
input.iterators 24.1.1
input.output 27
input.streams 27.6.1
insert.iter.cons 24.4.2.6.1
insert.iter.op* 24.4.2.6.3
insert.iter.op++ 24.4.2.6.4
insert.iter.op= 24.4.2.6.2
insert.iter.ops 24.4.2.6.5
insert.iterator 24.4.2.6
inserters 24.4.2.6.5
intro 1
intro.ack 1.10
intro.compliance 1.4
intro.defs 1.3
intro.execution 1.9
intro.memory 1.7
intro.object 1.8

Draft
Cross references

limits 18.2.1
list 23.2.3
list.capacity 23.2.3.2
list.cons 23.2.3.1
list.modifiers 23.2.3.3
list.ops 23.2.3.4
list.special 23.2.3.5
locale 22.1.1
locale.categories 22.2
locale.category 22.1.1.1.1
locale.codecvt 22.2.1.4
locale.codecvt.byname 22.2.1.5
locale.codecvt.members 22.2.1.4.1
locale.codecvt.virtuals 22.2.1.4.2
locale.collate 22.2.4.1
locale.collate.byname 22.2.4.2
locale.collate.members 22.2.4.1.1
locale.collate.virtuals 22.2.4.1.2
locale.cons 22.1.1.2
locale.convenience 22.1.3
locale ctype 22.2.1.1
locale ctype.byname 22.2.1.2
locale.ctype.members 22.2.1.1.1
locale.ctype.virtuals 22.2.1.1.2
locale.facet 22.1.1.1.2
locale.global.templates 22.1.2
locale.id 22.1.1.1.3
locale.members 22.1.1.3
locale.messages 22.2.7.1
locale.messages.byname 22.2.7.2
locale.messages.members 22.2.7.1.1
locale.messages.virtuals 22.2.7.1.2
locale.money.get 22.2.6.1
locale.money.get.members 22.2.6.1.1
locale.money.get.virtuals 22.2.6.1.2
locale.money.put 22.2.6.2
locale.money.put.members 22.2.6.2.1
locale.money.put.virtuals 22.2.6.2.2
locale.moneypunct 22.2.6.3
locale.moneypunct.byname 22.2.6.4
locale.moneypunct.members 22.2.6.3.1
locale.moneypunct.virtuals 22.2.6.3.2
locale.nm.put 22.2.2.2
locale.num.get 22.2.2.1
locale.numpunct 22.2.3.1
locale.numpunct.byname 22.2.3.2
locale.operators 22.1.1.4
locale.statics 22.1.1.5
locale.time.get 22.2.5.1
locale.time.get.byname 22.2.5.2
locale.time.get.members 22.2.5.1.1
locale.time.get.virtuals 22.2.5.1.2
locale.time.put 22.2.5.3
locale.time.put.byname 22.2.5.4
locale.time.put.members 22.2.5.3.1
locale.time.put.virtuals 22.2.5.3.2
locale.types 22.1.1.1
locales 22.1
localization 22
logic.error 19.1.1
logical.operations 20.5.8
lower.bound 25.3.3.1

M

macro.names 17.4.3.1.1
make.heap 25.3.6.3
map 23.3.1
map.access 23.3.1.2
map.cons 23.3.1.1
map.modifiers 23.3.1.3
map.ops 23.3.1.4
map.special 23.3.1.5
mask.array.assign 26.5.8.2
mask.array.comp.assign 26.5.8.3
mask.array.cons 26.5.8.1
mask.array.fill 26.5.8.4
member.functions 17.4.4.4
member.pointer.adaptors 20.5.12
memory 20.6
meta 20.4
meta.help 20.4.3
meta.rel 20.4.5
meta.rqmts 20.4.1
meta.trans 20.4.6
meta.trans.arr 20.4.6.4
meta.trans.cv 20.4.6.1
meta.trans.other 20.4.7
meta.trans.ptr 20.4.6.5
meta.trans.ref 20.4.6.2
meta.trans.sign 20.4.6.3
meta.type.synop 20.4.2
over.ass 13.5.3
over.best.ics 13.3.3.1
over.binary 13.5.2
over.built 13.6
over.call 13.5.4
over.call.func 13.3.1.1.1
over.call.object 13.3.1.1.2
over.dcl 13.2
over.ics.ellipsis 13.3.3.1.3
over.ics.rank 13.3.3.2
over.ics.ref 13.3.3.1.4
over.ics.scs 13.3.3.1.1
over.ics.user 13.3.3.1.2
over.inc 13.5.7
over.load 13.1
over.match 13.3
over.match.best 13.3.3
over.match.call 13.3.1.1
over.match.conv 13.3.1.5
over.match.copy 13.3.1.4
over.matchctor 13.3.1.3
over.match.funcs 13.3.1
over.match.oper 13.3.1.2
over.match.viable 13.3.1.6
over.oper 13.5
over.over 13.4
over.ref 13.5.6
over.sub 13.5.5
overflow.error 19.1.8

P
pairs 20.2.3
partial.sort 25.3.1.3
partial.sort.copy 25.3.1.4
partial.sum 26.6.3
pop.heap 25.3.6.2
predef.iterators 24.4
priority.queue 23.2.4.2
priqueue.cons 23.2.4.2.1
priqueue.members 23.2.4.2.2
priqueue.special 23.2.4.2.3
propagation 18.7.5
protection.within.classes 17.4.4.6

Q
queue 23.2.4.1
queue.defn 23.2.4.1.1
queue.ops 23.2.4.1.2
queue.opspecial 23.2.4.1.3

R
rand 26.4
rand.adapt 26.4.4
rand.adapt.disc 26.4.4.1
rand.adapt.ibits 26.4.4.2
rand.adapt.shuf 26.4.4.3
rand.adapt.xor 26.4.4.4
rand.device 26.4.6
rand.dist 26.4.8
rand.dist.bern 26.4.8.2
rand.dist.bern.bernoulli 26.4.8.2.1
rand.dist.bern.bin 26.4.8.2.2
rand.dist.bern.geo 26.4.8.2.3
rand.dist.bern.negbin 26.4.8.2.4
rand.dist.norm 26.4.8.4
rand.dist.norm.cauchy 26.4.8.4.4
rand.dist.norm.chisq 26.4.8.4.3
rand.dist.norm.f 26.4.8.4.5
rand.dist.norm.lognormal 26.4.8.4.2
rand.dist.norm.normal 26.4.8.4.1
rand.dist.norm.t 26.4.8.4.6
rand.dist.pois 26.4.8.3
rand.dist.pois.exp 26.4.8.3.2
rand.dist.pois.extreme 26.4.8.3.5
rand.dist.pois.gamma 26.4.8.3.3
rand.dist.pois.poisson 26.4.8.3.1
rand.dist.pois.weibull 26.4.8.3.4
rand.dist.samp 26.4.8.5
rand.dist.samp.discrete 26.4.8.5.1
rand.dist.samp.genpdf 26.4.8.5.3
rand.dist.samp.pconst 26.4.8.5.2
rand.dist.uni 26.4.8.1
rand.dist.uni.int 26.4.8.1.1
rand.dist.uni.real 26.4.8.1.2
rand.eng 26.4.3
reverse.iter.op> 24.4.1.3.16
reverse.iter.op>= 24.4.1.3.17
reverse.iter.opdiff 24.4.1.3.19
reverse.iter.opindex 24.4.1.3.12
reverse.iter.opref 24.4.1.3.5
reverse.iter.ops 24.4.1.3
reverse.iter.opsum 24.4.1.3.20
reverse.iter.op-- 24.4.1.3.7
reverse.iterator 24.4.1.1
reverse.iterators 24.4.1
round.style 18.2.1.3
runtime.error 19.1.6
stmt.ambig 6.8
stmt.block 6.3
stmt.break 6.6.1
stmt.cont 6.6.2
stmt.dcl 6.7
stmt.do 6.5.2
stmt.expr 6.2
stmt.for 6.5.3
stmt.goto 6.6.4
stmt.if 6.4.1
stmt.iter 6.5
stmt.jump 6.6
stmt.label 6.1
stmt.return 6.6.3
stmt.select 6.4
stmt.stmt 6
stmt.switch 6.4.2
stmt.while 6.5.1
storage.iterator 20.6.2
stream.buffers 27.5
stream.iterators 24.5
stream.types 27.4.1
streambuf 27.5.2
streambuf.assign 27.5.2.3.1
streambuf.buffer 27.5.2.2.2
streambuf.cons 27.5.2.1
streambuf.get.area 27.5.2.3.2
streambuflocales 27.5.2.2.1
streambuf.members 27.5.2.2
streambuf.protected 27.5.2.3
streambuf.pub.get 27.5.2.2.3
streambufpub.pback 27.5.2.2.4
streambuf.pub.put 27.5.2.2.5
streambuf.pub.area 27.5.2.3.3
streambuf.reqts 27.5.1
streambuf.virt.buffer 27.5.2.4.2
streambuf.virt.get 27.5.2.4.3
streambuf.virtlocales 27.5.2.4.1
streambuf.virt.pback 27.5.2.4.4
streambuf.virt.put 27.5.2.4.5
streambuf.virtuals 27.5.2.4
string.access 21.3.5
string.capacity 21.3.4
string.classes 21.2
string.cons 21.3.2
string.io 21.3.8.9
<table>
<thead>
<tr>
<th>Cross references</th>
<th>1118</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Cross references</th>
<th>1118</th>
</tr>
</thead>
</table>

| string.iterators | 21.3.3 |
| string.modifiers | 21.3.6 |
| string.nonmembers | 21.3.8 |
| string.ops | 21.3.7 |
| string.require | 21.3.1 |
| string.special | 21.3.8 |
| stringstreams | 27.7 |
| string.append | 21.3.6.2 |
| stringassign | 21.3.6.3 |
| stringcompare | 21.3.7.8 |
| stringcopy | 21.3.6.7 |
| stringerase | 21.3.6.5 |
| stringfind | 21.3.7.1 |
| string::find.first.not.of | 21.3.7.5 |
| string::find.first.of | 21.3.7.3 |
| string::find.last.not.of | 21.3.7.6 |
| string::find.last.of | 21.3.7.4 |
| string::insert | 21.3.6.4 |
| string::op!= | 21.3.8.3 |
| string::op+ | 21.3.8.1 |
| string::op+= | 21.3.6.1 |
| string::op< | 21.3.8.4 |
| string::op<= | 21.3.8.6 |
| string::op>| 21.3.8.5 |
| string::op>= | 21.3.8.7 |
| string::operator== | 21.3.8.2 |
| string::replace | 21.3.6.6 |
| string::rfind | 21.3.7.2 |
| string::substr | 21.3.7.7 |
| string::swap | 21.3.6.8 |
| stringbuf | 27.7.1 |
| stringbuf.assign | 27.7.1.2 |
| stringbuf.cons | 27.7.1.1 |
| stringbuf.members | 27.7.1.3 |
| stringbuf.virtuals | 27.7.1.4 |
| strings | 21 |
| stringstream | 27.7.4 |
| stringstream.assign | 27.7.5.1 |
| stringstream.cons | 27.7.5 |
| stringstream.members | 27.7.6 |
| structure | 17.3.1 |
| structure.requirements | 17.3.1.2 |
| structure.see.also | 17.3.1.4 |
| structure.specifications | 17.3.1.3 |
| structure.summary | 17.3.1.1 |
| support.dynamic | 18.5 |
| support.exception | 18.7 |
| support.limits | 18.2 |
| support.rtti | 18.6 |
| support.runtime | 18.8 |
| support.start.term | 18.4 |
| support.types | 18.1 |
| syntax | 1.6 |
| syserr | 19.4 |
| syserr.errcat | 19.4.1 |
| syserr.errcat.nonvirtuals | 19.4.1.3 |
| syserr.errcat.objects | 19.4.1.4 |
| syserr.errcat.overview | 19.4.1.1 |
| syserr.errcat.virtuals | 19.4.1.2 |
| syserr.errcode | 19.4.2 |
| syserr.errcode.constructors | 19.4.2.2 |
| syserr.errcode.modifiers | 19.4.2.3 |
| syserr.errcode.nonmembers | 19.4.2.6 |
| syserr.errcode.observers | 19.4.2.4 |
| syserr.errcode.overview | 19.4.2.1 |
| syserr.errcode.relational | 19.4.2.5 |
| syserr.syserr | 19.4.3 |
| syserr.syserr.members | 19.4.3.2 |
| syserr.syserr.overview | 19.4.3.1 |

<table>
<thead>
<tr>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp</td>
</tr>
<tr>
<td>temp.alias</td>
</tr>
<tr>
<td>temp.arg</td>
</tr>
<tr>
<td>temp.arg.explicit</td>
</tr>
<tr>
<td>temp.arg.nontype</td>
</tr>
<tr>
<td>temp.arg.template</td>
</tr>
<tr>
<td>temp.arg.type</td>
</tr>
<tr>
<td>temp.class</td>
</tr>
<tr>
<td>temp.class.order</td>
</tr>
<tr>
<td>temp.class.spec</td>
</tr>
<tr>
<td>temp.class.spec.match</td>
</tr>
<tr>
<td>temp.class.spec.mfunc</td>
</tr>
<tr>
<td>temp.decls</td>
</tr>
<tr>
<td>temp.deduct</td>
</tr>
<tr>
<td>temp.deduct.call</td>
</tr>
<tr>
<td>temp.deduct.conv</td>
</tr>
<tr>
<td>temp.deduct.funcaddr</td>
</tr>
<tr>
<td>temp.deduct.partial</td>
</tr>
<tr>
<td>temp.deduct.type</td>
</tr>
<tr>
<td>temp.dep</td>
</tr>
</tbody>
</table>

Draft
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.inject 14.5.6.2
temp.local 14.6.1
temp.mem 14.5.2
temp.mem.class 14.5.1.2
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 23.3.5
template.gslice.array 26.5.7
template.indirect.array 26.5.9
template.mask.array 26.5.8
template.slice.array 26.5.5
template.valarray 26.5.2
temporary.buffer 20.6.3
terminate 18.7.3.3
terminate.handler 18.7.3.1	
tuple 20.3
tuple.cnstr 20.3.1.1
tuple.creation 20.3.1.2
tuple.elem 20.3.1.4
tuple.helper 20.3.1.3
tuple.rel 20.3.1.5
tuple.tuple 20.3.1
type.descriptions 17.3.2.1
type.info 18.6.1

U
uncaught 18.7.4
underflow.error 19.1.9
unexpected 18.7.2.4
unexpected.handler 18.7.2.2
uninitialized.copy 20.6.4.1
uninitialized.fill 20.6.4.2
uninitialized.fill.n 20.6.4.3
unique.ptr 20.6.5
unique.ptr.compiletime 20.6.5.4
unique.ptr.compiletime.dtor 20.6.5.4.1
unique.ptr.compiletime.modifiers 20.6.5.4.3
unique.ptr.compiletime.observers 20.6.5.4.2
unique.ptr.dltr 20.6.5.1
unique.ptr.dltr.dlt 20.6.5.1.1
unique.ptr.dltr.dlt1 20.6.5.1.2
unique.ptr.dltr.dlt2 20.6.5.1.3
unique.ptr.runtime 20.6.5.3
unique.ptr.runtime.ctor 20.6.5.3.1
unique.ptr.runtime.modifiers 20.6.5.3.3
unique.ptr.runtime.observers 20.6.5.3.2
unique.ptr.single 20.6.5.2
unique.ptr.single.asgn 20.6.5.2.3
unique.ptr.single.ctor 20.6.5.2.1
unique.ptr.single.dtor 20.6.5.2.2
unique.ptr.single.modifiers 20.6.5.2.5
unique.ptr.single.observers 20.6.5.2.4
unique.ptr.special 20.6.5.5
unord 23.4
unord.hash 20.5.15
unord.map 23.4.1
unord.map.cnstr 23.4.1.1
unord.map.elem 23.4.1.2
unord.map.swap 23.4.1.3
unord.multimap 23.4.2
unord.multimap.cnstr 23.4.2.1
unord.multimap.swap 23.4.2.2
unord.multiset 23.4.4
unord.multiset.cnstr 23.4.4.1
unord.multiset.swap 23.4.4.2
unord.req 23.1.3
unord.req.exception 23.1.3.1
unord.set 23.4.3

Draft
Index

!, see logical negation operator
!=, see inequality operator
(), see function call operator
   function declarator, 168
*, see indirect operator, see multiplication operator
   pointer declarator, 163
+, see unary plus operator, see addition operator
++, see increment operator
,, see comma operator
-, see unary minus operator, see subtraction operator
->, see class member access operator
->*, see pointer to member operator
--., see decrement operator
,, see class member access operator
., see ellipsis
/, see division operator
   field declaration, 202
   label specifier, 111
::, see scope resolution operator
::*
   pointer to member declarator, 166
<, see less than operator
   template and, 297, 298
<<, see left shift operator
<=, see less than or equal to operator
=, see assignment operator
==, see equality operator
>, see greater than operator
>=, see greater than or equal operator
>>, see right shift operator
?::, see conditional expression operator
[][], see subscripting operator
   array declarator, 166
#define, 388
#else, 386
#endif, 386
#error, 393
#if, 385, 414
ifdef, 386
ifndef, 386
#include, 386, 409
#line, 393
#pragma, 393
#undef, 390, 410
%, see modulus operator
&., see address-of operator, see bitwise AND operator
   reference declarator, 164
&&, see logical AND operator
^, see bitwise exclusive OR operator
## operator, 389
# operator, 389
basic_ios::failure argument
   implementation-defined, 928
const object
   undefined change to, 131
exception::what message
   implementation-defined, 439
friend::what message
   implementation-defined, 439
friend function
   nested class, 204
delete, 94, 97
new, 94
operator bool
   basic_ios, 927
\, see backslash
   __cplusplus, 394
   __DATE__, 394
   __FILE__, 394
   __LINE__, 394
   __STDC_HOSTED__, 394
   __STDC_ISO_10646__, 394
INDEX 1122

implementation-defined, 394

-STDC_VERSION-, 394

implementation-defined, 394

-STDC-, 394

implementation-defined, 394

-TIME-, 394

{}

block statement, 111
class declaration, 189
class definition, 189
enum declaration, 137
initializer list, 181

~, see one’s complement operator, see destructor

~Init

io_base::Init, 919

~auto_ptr

auto_ptr, 1101

~basic_filebuf

basic_filebuf, 988

~basic_iostream

basic_iostream, 957

~basic_ostream

basic_ostream, 961

~ctype<char>

cctype<char>, 609

~exception

exception, 439

~locale

locale, 597

~sentry

basic_istream, 947

basic_ostream, 962

~sstream

sstream, 1097

~sstreambuf

sstreambuf, 1091

~valarray

valarray, 882

_., see character, underscore

|, 104

0, see also zero, null

null character, 23

string terminator, 23

-1, 504

a()

cauhy_distribution<>, 870

dextreme_value_distribution<>, 866

uniform_int_distribution<>, 856

uniform_real_distribution<>, 857

weibull_distribution<>, 865

abort, 54, 116, 409, 429, 435, 441

abs, 888, 902, 903

complex, 828

abstract-declarator, 160

access

union default member, 189

adjusting base class member, 227

base class, 224

base class member, 207

class member, 82

member name, 221

overloading and, 266

virtual function, 232

access-specifier, 207

access control, 221

anonymous union, 201

member function and, 235

overloading resolution and, 211

access specifier, 223, 224

accumulate, 899

acos, 888, 903

complex, 829

acosh, 903

complex, 829

addition operator, 101

additive-expression, 101

address, 65, 103

ddress of member function

unspecified, 414

adjacent_difference, 901

adjacent_find, 793

advance, 755

aggregate, 181

algorithm

stable, 400

<algorithm>, 781

alias, 144

alignment

extended, 68

fundamental, 68

Draft
alignment requirement
  implementation-defined, 68
allocation
  alignment storage, 94
  implementation defined bit-field, 202
  unspecified, 195
allocator, 730, 733, 736, 739, 1034
allocator, 516
alpha()
  gamma_distribution<>, 864
always_noconv
  codecvt, 613
ambiguity
  base class member, 210
  class conversion, 123
  declaration type, 123
  declaration versus cast, 161
  declaration versus expression, 118
  function declaration, 179
  member access, 210
  parentheses and, 93
ambiguity detection
  overloaded function, 267
Amendment 1, 411
any
  bitset, 727
append
  basic_string, 568
apply
  valarray, 886
arg, 830
  complex, 828
argc, 52
argument, 2, 413, 414, 448
  access checking and default, 222
  binding of default, 172
  evaluation of default, 172, 173
  example of default, 171, 172
  overloaded operator and default, 286
  reference, 80
  scope of default, 173
  template, 299
  type checking of default, 172
arguments
  implementation-defined order of evaluation of function, 173
argument and name hiding
  default, 173
argument and virtual function
  default, 174
argument list
  empty, 168
  variable, 168
argument passing, 80
  reference and, 185
argument substitution, 388
argument type
  unknown, 168
argv, 52
arithmetic
  pointer, 101
  unsigned, 64
array, 169
  bound, 167
  const, 66
  delete, 97
  multidimensional, 167
  new, 94
  overloading and pointer versus, 264
  sizeof, 92
  storage of, 168
<array>, 669
array, 673, 675
  as aggregate, 673
  begin, 673
  contiguous storage, 673
  data, 675
  end, 673
  get, 675
  initialization, 673, 674
  max_size, 673
  size, 673, 675
  swap, 675
  tuple interface to, 675
  zero sized, 675
array size
  default, 167
arrow operator, see class member access operator
asin, 888, 903
  complex, 829
asinh, 903
  complex, 829
asm
  implementation-defined, 154
assembler, 154
<assert.h>, 409
assign
  basic_string, 568
deque, 679
list, 684
vector, 698
basic_regex, 1023
swap
  function, 512
assignment
  and lvalue, 107
  conversion by, 107
  reference, 185
assignment-expression, 107
assignment-operator, 107
assignment operator
  copy, 258
  overloaded, 287
associative containers
  unordered, see unordered associative containers
at
  basic_string, 567
  map, 709
atan, 888, 903
  complex, 829
atan2, 888, 903
atanh, 903
  complex, 829
atexit, 54, 409, 430
auto
  destruction of, 116, 117
auto_ptr, 535, 1099
  auto_ptr, 1100
awk, 1015
b()
  cauchy_distribution<>, 870
  extreme_value_distribution<>, 866
  uniform_int_distribution<>, 856
  uniform_real_distribution<>, 857
  weibull_distribution<>, 865
back
  basic_string, 567
back_insert_iterator, 762
  back_insert_iterator, 762
back_inserter, 763
backslash character, 21
bad
  basic_ios, 928
  bad_alloc, 95, 431, 435
  bad_alloc, 435
  bad_alloc::what
    implementation-defined, 435
bad_cast, 84, 437
  bad_cast, 437
  bad_cast::what
    implementation-defined, 437
bad_exception, 380, 439
  bad_exception, 440
  bad_exception::what
    implementation-defined, 440
bad_function_call, 508
  bad_function_call, 508
bad_typeid, 85, 437
  bad_typeid, 438
  bad_typeid::what
    implementation-defined, 438
bad_weak_ptr, 531
  bad_weak_ptr, 531
  what, 531
base-specifier, 207
base-specifier-list, 207
base class, 207, 208
  direct, 207
  indirect, 207
  private, 224
  protected, 224
  public, 224
base class virtual, see virtual base class
basic_filebuf, 908, 986
  basic_filebuf, 987
basic_filebuf<char>, 985
basic_filebuf<wchar_t>, 985
basic_fstream, 908, 998
  basic_fstream, 999
basic_ifstream, 908, 993
  basic_ifstream, 994
basic_ifstream<char>, 985
basic_ifstream<wchar_t>, 985
basic_ios, 908, 924
basic_ios, 925
basic_ios<char>, 913
basic_ios<wchar_t>, 913
basic_istream, 956
    basic_istream, 957
basic_istream, 908, 943
    basic_istream, 946
basic_istream<char>, 942
basic_istream<wchar_t>, 942
basic_istringstream, 908, 978
    basic_istringstream, 979
basic_istringstream<char>, 971
basic_istringstream<wchar_t>, 971
basic_ofstream, 908, 995
    basic_ofstream, 996
basic_ofstream<char>, 985
basic_ofstream<wchar_t>, 985
basic_ostream, 908, 1030
    basic_ostream, 960
basic_ostream<char>, 943
basic_ostream<wchar_t>, 943
basic_ostreambuf_iterator, 908
basic_ostringstream, 908, 983
    basic_ostringstream, 984
basic_regex, 1006, 1019, 1045
    assign, 1023
    basic_regex, 1021, 1022
    constants, 1021
    operator=, 1022
    swap, 1024
basic_streambuf, 908, 932
    basic_streambuf, 934
basic_streambuf<char>, 931
basic_streambuf<wchar_t>, 931
basic_string, 557, 579, 971
    basic_string, 557, 579, 971
basic_stringbuf, 908, 972
    basic_stringbuf, 973
basic_stringbuf<char>, 971
basic_stringbuf<wchar_t>, 971
basic_stringstream, 908, 983
    basic_stringstream, 984
    type_info, 436
begin
    basic_string, 565
    array, 673
    match_results, 1033
    unordered associative containers, 668
behavior
    conditionally-supported, 2, 5
    default, 398, 403
    implementation-defined, 3, 636
    locale-specific, 3
    required, 399, 403
    undefined, 4
    unspecified, 4
Ben, 266
Bernoulli distributions, 857–861
bernoulli_distribution, 857
    constructor, 858
    discrete probability function, 857
    p(), 858
beta()
    gamma_distribution<>, 864
bidirectional_iterator_tag, 753, 754
binary function, 497, 498, 508
binary_function, 496
binary_narrow, 496
binary_search, 808
binary operator
    interpretation of, 287
    overloaded, 287
bind, 503–504
bind1st, 1098
bind2nd, 1099
binder1st, 1098
binder2nd, 1098
binding
    reference, 185
binomial_distribution<>, 858
    constructor, 859
    discrete probability function, 858
    p(), 859
    t(), 859
bit-field, 202
    address of, 202
    alignment of, 202
    implementation-defined sign of, 202
Draft
implementation defined alignment of, 202
type of, 202
unnamed, 202
zero width of, 202
<bitset>, 722
bitset, 722
  bitset, 724
block
  initialization in, 117
block scope; see local scope, 31
block structure, 117
body
  function, 175
bool()
    basic_istream operator, 947
    basic_ostream operator, 962
boolalpha, 928
Boolean, 202
Boolean literal, 24
boolean-literal, 24
Boolean type, 64
bound arguments, 504
bound, of array, 167
bucket
  unordered associative containers, 668
bucket_count
    unordered associative containers, 668
bucket_size
    unordered associative containers, 668
buckets, 663
built-in type; see fundamental type, 63
byte, 92
C
  linkage to, 155
c-char, 20
c-char-sequence, 20
c_str
    basic_string, 574
cabs
    complex, 828
cacos
    complex, 829
cacosh
    complex, 829
call, see also function call, member function call, overloaded function call, virtual function call
  operator function, 286
  pseudo destructor, 81
call signature, 495
call wrapper, 495, 496
    forwarding, 496
    simple, 496
call wrapper type, 495
callable object, 495, 510
callable type, 495
calloc, 545, 1083
capacity
    basic_string, 566
    vector, 698
carry
    subtract_with_carry_engine<, 844
casin
    complex, 829
casinh
    complex, 829
<cassert>, 409
cast
  base class, 87
  const, 89
  derived class, 87
  dynamic, 83, 437
  integer to pointer, 88
  lvalue, 86, 87
  pointer to function, 88
  pointer to integer, 88
  pointer to member, 87, 88
  reference, 86, 88
  reinterpret, 87
  reinterpret_cast
    lvalue, 87
    reference, 88
    static, 86
  static_cast
    lvalue, 86
    reference, 86
    undefined pointer to function, 88
cast-expression, 98
casting, 81, 98
catan
    complex, 829
catanh
complex, 829
catch, 371
category
locale, 592
caucho_distribution<>., 869
a(), 870
b(), 870
constructor, 870
probability density function, 869
cbrt, 903
<ccomplex>, 831
cerr, 911
<cerrno>, 411
<cfenv>, 820
char
implementation-defined sign of, 63
char-like object, 547
char-like type, 547
char16_t, 20, 406
char16_t character, 20
char32_t, 20, 406
char32_t character, 20
char_class_type
regex_traits, 1017
Regular Expression Traits, 1004
character, 398
decimal-point, 405
multibyte, 3
set
basic execution, 6
basic source, 12
signed, 63
underscore, 411
in identifier, 17
character-literal, 20
character string, 22
checking
point of error, 325
syntax, 325
chi_squared_distribution<>., 868
constructor, 869
n(), 869
probability density function, 868
cin, 911
<cs0646>, 1082
class, 65, 189
abstract, 218
base, 411, 415
cast to incomplete, 99
constructor and abstract, 219
definition, 27
derived, 415
linkage of, 49
linkage specification, 156
pointer to abstract, 219
polymorphic, 214
scope of enumerator, 140
standard-layout, 190
template, 723
trivial, 189
unnamed, 127
class-key, 189
class-name, 189
class-specifier, 189
classes
narrow-oriented iostream, 399
wide-oriented iostream, 400
classic
locale, 598
classic_table
classic_table
ctype<char>, 611
class base, see base class
class derived, see derived class
class local, see local class
class member, see also member
class name, 160
eaborated, 135, 192
point of declaration, 193
scope of, 191
typedef, 127, 193
class nested, see nested class
class object
assignment to, 107
const, 66
member, 194
sizeof, 92
class object copy, see also copy constructor
class object initialization, see also constructor
clear
class object copy, see also copy constructor
class object initialization, see also constructor
clear
basic_ios, 928
basic_string, 566
unordered associative containers, 667

<climits>, 1090
<locale>, 1082
<locale>, 405
clog, 911
close
  basic_filebuf, 989, 1000
  basic_ifstream, 995
  basic_ofstream, 998
  messages, 644
codecvt, 611
codecvt_byname, 616
collate, 628
collate_byname, 630
collating element, 1003
combine
  locale, 597
comment, 14
  /* */, 15
  //, 15
compare
  basic_string, 578
  collate, 629
  sub_match, 1025
comparison
  pointer, 103
  pointer to function, 103
  undefined pointer, 101, 103
  unspecified pointer, 103
  void* pointer, 103
compilation
  separate, 11
compiler control line, see preprocessing directive
completely defined, 194
<complex>, 821
complex, 823
  complex, 825
  <complex.h>, 831
component, 398
compound-statement, 112
concatenation
  string, 23
    undefined string literal, 23
condition, 112
  condition s
    rules for, 112
conditional-expression
  throw-expression in, 105
conj, 830
  complex, 828
consistency
  linkage, 124
  linkage specification, 156
  type declaration, 51
const, 65
  constructor and, 199, 236
  destructor and, 199, 242
  linkage of, 49, 124
  overloading and, 265
const_local_iterator, 664
  unordered associative containers, 664
const_mem_fun1_ref_t, 507
const_mem_fun1_t, 506
const_mem_fun_ref_t, 507
const_mem_fun_t, 506
const_pointer_cast
  shared_ptr, 540
constant, 18, 77
  enumeration, 138
  null pointer, 73
constant-expression, 109
constant-initializer, 193
constructor, 235
  address of, 236
  array of class objects and, 248
  conversion by, 240
  copy, 236, 238, 256, 407
  exception handling, 374
  inheritance of, 236
  non-trivial, 236
  random number engine requirement, 833
  type of, 236
  union, 201
    unspecified argument to, 97
constructor call
  explicit, 237
constructor conversion by, see also user-defined conversion
constructor default, see default constructor
context
  non-deduced, 362
control line, see preprocessing directive

Draft
convention, 403
conversion
  argument, 168
  array pointer, 70
  array-to-pointer, 70
  Boolean, 74
class, 239
derived-to-base, 278
floating point, 72
floating-integral, 73
function-to-pointer, 70
implementation-defined floating point, 72
implementation defined pointer integer, 88
implicit, 69, 239
implicit user-defined, 239
inheritance of user-defined, 242
integer, 72
integer rank, 74
lvalue-to-rvalue, 70, 1073
overload resolution and, 275
overload resolution and pointer, 285
pointer, 73
pointer to function, 70
pointer to member, 73
  void*, 74
return type, 117
reverse_iterator, 758
signed unsigned integer, 72
standard, 69
static user-defined, 242
type of, 241
user-defined, 239–241
virtual user-defined, 242
conversion operator, see conversion function
conversion rank, 279
conversion-function-id, 241
conversions
qualification, 70
usual arithmetic, 76
conversion explicit type, see casting
conversion function, see also user-defined conversion
copy
  class object, 256
copy, 795
  basic_string, 573
copy_backward, 796
copyfmt
  basic_ios, 926
copysign, 903
copy assignment operator
  implicitly-declared, 259
copy constructor
  implicitly-declared, 257
cos, 888, 903
  complex, 829
cosh, 888, 903
  complex, 829
count, 793
  bitset, 727
  unordered associative containers, 667
count_if, 793
cout, 911
cref
  reference_wrapper, 499, 500
<cstdio>, 411
cshift
  valarray, 886
<cstdlib>, 169
<cstdlib>, 411
<cstring>, 92, 101, 1082, 1083
<ctgmath>, 428
<ctgmath>, 990
<ctgmath>, 52, 54, 409, 1082, 1086
<cstring>, 405, 1082
<cstring>, 405, 1090, 1096
<cstring>, 901
<ctime>, 1082
<ctime>, 591
cTOR-INITIALIZER, 248
ctype, 605
cctype<char>
  cctype<char>, 610
cctype_byname, 608
<uchar>, 406, 411
cv-qualified, 65
cv-qualifier, 160
<cwchar>, 406, 411, 1082
<cwchar>, 411
DAG
  multiple inheritance, 210
non-virtual base class, 210
virtual base class, 210
data
  basic_string, 574
  vector, 699
  array, 675
data member, see member
  static, 199
date_order
  time_get, 631
deallocation, see delete
dec, 930
dec, 964
decimal-literal, 18
decimal_point
  numpunct, 627
deci-specifier, 123
declaration, 25, 121
  extern reference, 185
typedef as type, 126
  access, 227
  array, 167
  asm, 154
  bit-field, 202
  class member, 193
  class name, 26
  constant pointer, 163
default argument, 171
definition versus, 25
  ellipsis in function, 80, 168
  enumerator point of, 31
  extern, 26
  forward, 125
  forward class, 192
  function, 25, 168
  member, 193
  multiple, 51
  name, 25
  overloaded, 263
  overloaded name and friend, 229
  parameter, 168
  parentheses in, 161, 163
  pointer, 163
  reference, 165
  register, 124
  static member, 26
  storage class, 123
type, 162
typedef, 26
declaration, 121
declaration-statement, 117
declaration hidding, see name hiding
declaration matching
  overloaded function, 266
declarator, 122, 159
  meaning of, 162
  multidimensional array, 167
declarator, 159
declarator-id, 160
decrement operator
  overloaded, 288
default
  access control, 221
default-initialization, 178
defaultfloat, 931
default argument
  overload resolution and, 275
default constructor, 236
default initializers
  overloading and, 265
definition, 25, 397
  static member, 200
  alternate, 412
  class, 189, 193
  class name as type, 191
  constructor, 175
declaration as, 122
  empty class, 189
  function, 175
  local class, 204
  member function, 195
  namespace, 141
  nested class, 203
  pure virtual function, 218
  scope of class, 191
  virtual function, 216
delete, 55, 97, 245
  operator, 545
destructor and, 97, 243
  operator, 412, 432, 433
  overloading and, 57
type of, 246

Draft
undefined, 97
delete-expression, 97
densities()
    piecewise_constant_distribution<>, 875
deprecated features, 83, 91
<deque>, 670
deque, 676
dereferencing, see also indirection
derivation, see inheritance
derived class, 207
    most, 6
    overloading and, 266
derived object
    most, 6
destructor, 242, 407
    default, 243
    exception handling, 374
    non-trivial, 243
    program termination and, 243
    pure virtual, 243
    union, 201
    virtual, 243
destructor call
    explicit, 243, 244
    implicit, 243
digit, 16
digit-sequence, 22
digraph, 15
direct-abstract-declarator, 160
direct-declarator, 159
direct-new-declarator, 93
directed acyclic graph, see DAG
directive
derror, 393
    null, 394
    pragma, 393
    preprocessing, 383
discard()
    random number engine requirement, 834
discard_block_engine<>, 845
    constructor, 846
    generation algorithm, 845
    state, 845
    template parameters, 846
    textual representation, 846
    transition algorithm, 845
discrete probability function, 836
    bernoulli_distribution<>, 857
    binomial_distribution<>, 858
    discrete_distribution<>, 872
    geometric_distribution<>, 859
    negative_binomial_distribution<>, 860
    poisson_distribution<>, 861
    uniform_int_distribution<>, 855
discrete_distribution<>, 872
    constructor, 873
    discrete probability function, 872
    discrete_distribution<>, 873
    probabilities(), 873
    weights, 873
distance, 755
distribution, see random number distribution
div, 902
    divides, 500
division
    implementation defined, 100
do_always_noconv
codecvt, 615
do_close
    messages, 645
do_compare
    collate, 629
do_curr_symbol
    moneypunct, 642
do_date_order
    time_get, 633
do_decimal_point
    moneypunct, 642
    num_get, 627
    numpunct, 627
do_encoding
    codecvt, 615
do_falsename
    num_get, 628
    numpunct, 628
do_frac_digits
    moneypunct, 642
    num_get, 618
do_get
    messages, 644
    money_get, 637
    num_get, 618
do_get_date
    time_get, 633
    do_get_monthname
Draft
INDEX

- time_get, 633
- do_get_time
time_get, 633
- do_get_weekday
time_get, 633
- do_get_year
time_get, 634
- do_grouping
  moneypunct, 642
  numpunct, 627
do_hash
collate, 629
do_in
  codecvt, 613
do_is
  ctype, 606
do_length
  codecvt, 615
do_max_length
  codecvt, 615
do_narrow, 611
cctype, 608
do_neg_format
  moneypunct, 643
do_negative_sign
  moneypunct, 642
do_open
  messages, 644
do_out
  codecvt, 613
do_pos_format
  moneypunct, 643
do_positive_sign
  moneypunct, 642
do_put
  money_put, 639
  num_put, 622
time_put, 636
do_scan_is
  ctype, 607
do_scan_not
  ctype, 607
do_thousands_sep
  moneypunct, 642
  numpunct, 627
do_tolower
  ctype, 607
do_toupper
  ctype, 607
do_transform
  collate, 629
do_truename
  numpunct, 628
do_unshift
  codecvt, 614
do_widen, 611
cctype, 607
domain_error, 446
domain_error, 446
dominance
  virtual base class, 212
dot operator, see class member access operator
dynamic binding, see virtual function
dynamic_pointer_cast
  shared_ptr, 539

eback
  basic_streambuf, 937
ECMAScript, 1015, 1045
egptr
  basic_streambuf, 938
egrep, 1015
elaborated-type-specifier, 135
elaborated type specifier, see elaborated class name
eision
  copy constructor, 260
eipsis
  conversion sequence, 81, 280
  overload resolution and, 275
else, 112
empty, 754
  basic_string, 567
  match_results, 1032
enable_shared_from_this, 543
  ~enable_shared_from_this, 544
  enable_shared_from_this, 544
  operator=, 544
  shared_from_this, 544
coding
  multibyte, 23
coding
  codecvt, 613

Draft
end
  basic_string, 565
  array, 673
  match_results, 1033
  unordered associative containers, 668
end-of-file, 728
endl, 967
endl, 964
ends, 968
engine, see random number engine
engine adaptor, see random number engine adaptor
engines with predefined parameters
  knuth_b, 851
  minstd_rand, 850
  minstd_rand0, 850
  mt19937, 851
  ranlux24, 851
  ranlux24_base, 851
  ranlux48, 851
  ranlux48_base, 851
entity, 25
entropy()
  random_device, 852
enum, 65
  overloading and, 264
  type of, 137, 138
  underlying type, 138
enumeration, 137, 138
  linkage of, 49
  scoped, 138
  unscoped, 138
enumeration scope, 34
enumeration type
  conversion to, 87
  static_cast
    conversion to, 87
enumerator
  definition, 27
  value of, 138
enumerator, 138
enum name
  typedef, 127
environment
  program, 52
eof
  basic_ios, 928
epptr
  basic_streambuf, 938
eq
  char_traits, 574–577
equal, 794
  istreambuf_iterator, 777
equal_range, 808
  unordered associative containers, 668
equal_to, 500
equality-expression, 103
equivalence
  template type, 306
  type, 126, 191
equivalent parameter declarations, 264
  overloading and, 264
erase
  basic_string, 571
deque, 680
  list, 685
  vector, 700
  unordered associative containers, 667
erf, 903
erfc, 903
error_type, 1014, 1016
escape-sequence, 20
escape character, see backslash
escape sequence
  undefined, 21
evaluation
  order of argument, 81
  unspecified order of, 9, 53
  unspecified order of argument, 81
  unspecified order of function call, 81
example
  *const, 163
  static member, 200
  array, 167
  class definition, 194
  const, 163
  constant pointer, 163
  constructor, 237
  constructor and initialization, 247
  declaration, 26, 170
  declarator, 160
  definition, 26
  delete, 246
derived class, 207
destructor and delete, 246
ellipsis, 168
enumeration, 139
explicit destructor call, 244
explicit qualification, 211
friend, 192
friend function, 228
function declaration, 169
function definition, 175
linkage consistency, 124
local class, 204
member function, 197, 228
member name access, 227
nested type name, 205
nested class, 203
nested class definition, 203, 233
nested class forward declaration, 204
pointer to member, 166
pure virtual function, 218
scope of delete, 246
scope resolution operator, 211
subscripting, 167
typedef, 126
type name, 160
unnamed parameter, 175
variable parameter list, 168
virtual function, 215, 216
exception
allowing an, 378
arithmetic, 75
bad_function_call, 508
bad_weak_ptr, 531
handling, 371
object, 373
undefined arithmetic, 75
<exception>, 438
exception
exception, 439
exception-declaration, 371
exception-specification, 377
exceptions
basic_ios, 928
exit, 52, 54, 116, 409, 430, 435
exp, 888, 903
complex, 829
exp2, 903
expired
weak_ptr, 542
explanation
subscripting, 167
explicit-specialization, 344
explicit type conversion, see casting
expm1, 903
exponent-part, 22
exponential_distribution<>, 862
constructor, 863
lambda(), 863
probability density function, 862
export, 293
expression, 75
constant, 109
order of evaluation of, 8
parenthesized, 77
pointer to member constant, 91
postfix, 79
primary, 77
reference, 75
rvalue reference, 75
unary, 90
expression, 107
expression-list, 79
expression-statement, 111
extended alignment, 68
extended integer type, 63
extended signed integer type, 63
extended unsigned integer type, 63
extern, 123
linkage of, 124
extern "C", 410, 411
extern "C++", 410, 411
extreme_value_distribution<>, 865
a(), 866
b(), 866
constructor, 866
probability density function, 865
fabs
complex, 828
facet
locale, 595
fail
basic_ios, 928
failed
    ostreambuf_iterator, 779
failure
    ios_base::failure, 916
falsename
    numpunct, 627
fclose, 990
fclose, 990
fdim, 903
FE_ALL_EXCEPT, 821
FE_DFL_ENV, 821
FE_DIVBYZERO, 821
FE_DOWNWARD, 821
FE_INEXACT, 821
FE_INVALID, 821
FE_OVERFLOW, 821
FE_TONEAREST, 821
FE_TOWARDZERO, 821
FE_UNDERFLOW, 821
FE_UPWARD, 821
fclearexcept, 821
fegetenv, 821
fegetexceptflag, 821
fegetround, 821
feholdexcept, 821
<fenv.h>, 821
fenv_t, 821
teraise_except, 821
tesetenv, 821
tesetexceptflag, 821
tesetround, 821
testexcept, 821
feupdateenv, 821
fexcept_t, 821
file, 11
    source, 11, 409, 411
filebuf, 908, 985
    implementation-defined, 993
fill, 799
    basic_ios, 926
gslice_array, 895
indirect_array, 898
mask_array, 896
slice_array, 891
fill_n, 799
find, 792
    basic_string, 574
    unordered associative containers, 667
find_end, 792
find_first_not_of
    basic_string, 576
find_first_of, 793
    basic_string, 575
find_if, 792
find_last_not_of
    basic_string, 577
find_last_of
    basic_string, 576
finite state machine, 1003
fisher_f_distribution<>, 870
    constructor, 871
        m(), 871
        n(), 871
    probability density function, 870
fixed, 931
flags
    ios_base, 604, 919
flip
    bitset, 726
    float_round_style, 424
    floating-literal, 22
    floating-suffix, 22
    floating point type, 64
        implementation-defined, 64
floor, 903
flush, 919, 947, 962, 968
    basic_ostream, 967
fma, 903
fmax, 903
fmin, 903
fmtflags
    ios_base, 916
        ios, 969
fopen, 989
fopen, 989
for
    scope of declaration in, 116
    for_each, 791
    formal argument, see parameter
format
    match_results, 1033, 1034

Draft
format specifier, 1003
format_default, 1013, 1014
format_first_only, 1013, 1014, 1038
format_no_copy, 1013, 1014, 1038
format_sed, 1013, 1014
forward, 468
forward_iterator_tag, 753, 754
forwarding call wrapper, 496
fpclassify, 905
fpos, 913, 922, 923
fractional-constant, 22
free, 545
freeze
  ostrstream, 1096
  strstreambuf, 1091
  strstream, 1097
free store, see also new, delete
frexp, 903
friend
  virtual and, 216
  access specifier and, 230
  class access and, 228
  inheritance and, 230
  local class and, 231
  template and, 313
friend function
  access and, 228
  inline, 230
  linkage of, 229
  member function and, 228
front
  basic_string, 567
front_insert_iterator, 763
  front_insert_iterator, 764
front_inserter, 764
fseek, 989
<fstream>, 985
fstream, 908
full-expression, 8
function, see also friend function, member function, in-line function, virtual function, 169
  allocation, 56, 94
  comparison, 398
  conversion, 241
  deallocation, 56, 97, 245
  definition, 27
global, 411, 414
handler, 398
linkage specification overloaded, 156
modifier, 399
observer, 399
operator, 285
plain old, 444
pointer to member, 100
replacement, 399
reserved, 400
viable, 267
virtual member, 411, 414
function, 508
  ∼function, 512
  assign, 512
  bool conversion, 512
  function, 510
  invocation, 512
  operator(), 512
  operator=, 511
  swap, 512, 513
  target, 512
  target_type, 512
function objects
  binders, 503–504
  mem_fn, 507–508
  reference_wrapper, 497
  return type, 496–497
  wrapper, 508–513
function-body, 175
function-definition, 175
function-specifier, 125
function-try-block, 371
<iostream>, 491
functions
  candidate, 336
  function argument, see argument
  function call, 80
    recursive, 81
    undefined, 88
  function call operator
    overloaded, 288
  function overloaded, see overloading
  function parameter, see parameter
  function prototype, 32
  function return, see return
function return type, see return type
function virtual, see virtual function
fundamental type
destructor and, 245
fundamental type conversion, see conversion, user-defined conversion
gamma_distribution<> , 863
    alpha(), 864
    beta(), 864
    constructor, 864
    probability density function, 863
gbump
    basic_streambuf, 938
gcount
    basic_istream, 951
general_pdf_distribution<> , 875
    constructor, 875
    probability density function, 875
    xmax(), 876
    xmin(), 876
generate, 799
generate_canonical<>(), 854, 855
generate_n, 799
generated destructor, see default destructor
generation algorithm
discard_block_engine<> , 845
    independent_bits_engine<> , 847
    linear_congruential_engine<> , 841
    mersenne_twister_engine<> , 842
    random number engine, 833
    shuffle_order_engine<> , 848
    subtract_with_carry_engine<> , 844
    xor_combine_engine<> , 849
geometric_distribution<> , 859
    constructor, 860
    discrete probability function, 859
    p(), 860
get
    auto_ptr, 1101
    basic_istream, 951
    money_get, 637
    num_get, 617
    array, 675
    pair, 471
    reference_wrapper, 499
    shared_ptr, 537
tuple, 478
get_date
    time_get, 631
g_delete
    shared_ptr, 540
get_money, 970
get_monthname
    time_get, 632
g_get_seeds()
    seed_seq, 854
get Temporary_buffer, 518, 519
get_time
    time_get, 631
get_weekday
    time_get, 632
get_year
    time_get, 632
getline
    basic_istream, 952, 953
    basic_string, 583
getloc
    basic_streambuf, 935
    ios_base, 920
global
    locale, 598
good
    basic_ios, 928
goto
    initialization and, 117
gptr
    basic_streambuf, 938
grammar, 1049
    regular expression, 1045
greater, 501
greater_equal, 501
grep, 1015
    grouping
    numpunct, 627
gslice
    gslice, 893
    class, 892
gslice_array, 893, 894
gslice_array, 894
Draft
handler
  exception, 375, 415
  incomplete type in exception, 375
handler, 371
handler-seq, 371
has_facet
  locale, 599
hash, 514
  collate, 629
  instantiation restrictions, 514
  operator(), 514
hash code, 663
hash function, 663
hash tables, see unordered associative containers
hash_function
  unordered associative containers, 665
hasher
  unordered associative containers, 664
header
  C, 410, 411, 414, 1085
header-name, 15
headers
  C++, 408
hex, 930
hex-quad, 13
hexadecimal-digit, 18
hexadecimal-escape-sequence, 20
hexadecimal-literal, 18
hexfloat, 931
hiding; see name hiding, 34
hypot, 903
id
  qualified, 78
id
  locale, 595
id-expression, 77
id-expression, 77
identifier, 17, 77, 122
identifier, 16
identifier-nondigit, 16
identity, 468
ifstream, 908, 985
ignore
  basic_istream, 953
ilogb, 903
imag, 830
  complex, 828
imbue
  basic_filebuf, 993
  basic_ios, 926
  basic_streambuf, 938
  ios_base, 920
immolation
  self, 346
implementation
  freestanding, 409
  hosted, 409
implementation-defined, 72, 409, 411, 415, 418, 430, 435,
  437–440, 920, 978, 992, 1082
implementation-dependent, 947, 962
implementation-generated, 26
implicitly-declared default constructor, 236, see also default constructor
implicit object argument, 268
implied object parameter, 268
  implicit conversion sequences, 268
in
  codecvt, 613
  in_avail
    basic_streambuf, 936
includes, 810
inclusion
  conditional, 385
  source file, 386
incomplete, 101
increment
  bool, 83, 91
increment operator
  overloaded, 288
independent_bits_engine<>, 846
  generation algorithm, 847
  state, 846
  template parameters, 847
  textual representation, 847
  transition algorithm, 846
indirect_array, 897
  indirect_array, 897
indirection, 90
inheritance, 207, see also multiple inheritance
Init
  ios_base::Init, 919
Draft
init
  basic_ios, 946, 961
init-declarator, 159
init-declarator-list, 159
initialization, 177
  static member, 200
  static object, 178
  array, 181
  array of class objects, 184, 248
  auto, 118
  auto object, 178
  automatic, 117, 118
  base class, 248, 250
  character array, 184
  class member, 179
  class object, 181, 247
  const, 131, 181
  const member, 250
  constant, 52
  constructor and, 247
  copy, 179
  default, 178
  default constructor and, 247
  definition and, 122
  direct, 179
  dynamic, 52
  jump past, 117
  local static, 118
  member, 248
  member object, 250
  order of, 52, 209
  order of base class, 250
  order of member, 250
  order of virtual base class, 250
  overloaded assignment and, 248
  parameter, 80
  reference, 165, 185
  reference member, 250
  run-time, 52
  static object, 52, 178
  struct, 181
  union, 184, 201
  virtual base class, 251, 258
initialization class object, see also constructor
initializer
  base class, 175
  constant, 193
  member, 175
  scope of member, 251
  temporary and declarator, 238
initializer, 177
initializer-clause, 178
initializer-list, 178
injected-class-name, 189
inline, 414
  linkage of, 49
  inline function, 125
inner_product, 900
inplace_merge, 809
input_iterator_tag, 753, 754
emplace
  deque, 680
insert
  basic_string, 569
  deque, 680
  list, 685
  vector, 699
  unordered associative containers, 666, 667
push_back
  deque, 680
push_front
  deque, 680
insert_iterator, 765
  insert_iterator, 765
inserter, 766
instantiation
  explicit, 341
  point of, 335
  template implicit, 338
int
  bool promotion to, 72
int16_t, 428
int32_t, 428
int64_t, 428
int8_t, 428
int_fast16_t, 428
int_fast32_t, 428
int_fast64_t, 428
int_fast8_t, 428
int_least16_t, 428
int_least32_t, 428

int_least64_t, 428
int_least8_t, 428
integer-literal, 18
integer-suffix, 19
integer type, 64
integral type, 64
sizeof, 63
internal, 930
interval boundaries
   piecewise_constant_distribution<> , 873
intervals()
   piecewise_constant_distribution<> , 874
intmax_t, 428
intptr_t, 428
invalid_argument, 447, 724
   invalid_argument, 447
invocation
   macro, 388
   INVOKE, 495, 496
   <iomanip>, 943
   <ios>, 912
   ios, 908, 913
   ios_base, 913
      ios_base, 922
    ios_base::failure, 916
    ios_base::Init, 918
   <iosfwd>, 908
iostate
   ios_base, 917
   <iostream>, 910
is
   ctype<char>, 610
   ctype, 606
is_bind_expression, 503
   value, 503
is_heap, 814
is_open
   basic_filebuf, 989, 1000
   basic_ifstream, 995
   basic_ofstream, 997
is_placeholder, 503
   value, 503
is_sorted, 806
is_sorted_until, 806
isalnum, 599
isalpha, 599
iscntrl, 599
isctype
   regex_traits, 1018
      Regular Expression Traits, 1005, 1046
isdigit, 599
isfinite, 905
isgraph, 599
isgreater, 905
isgreaterequal, 905
isinf, 905
isless, 905
islessequal, 905
islessgreater, 905
islower, 599
isnan, 905
isnormal, 905
<iso646.h>, 1082
isprint, 599
ispunct, 599
isspace, 599
<iostream>, 942
istream, 908, 942
istream_iterator, 771
   operator!=, 773
   operator==, 773
istreambuf_iterator, 775
   istreambuf_iterator, 776
istringstream, 908, 971
istrstream, 1094
   istrstream, 1094
isunordered, 905
isupper, 599
isxdigit, 599
iter_swap, 797
iteration-statement, 114, 116
<iterator>, 748
iword
   ios_base, 921
Jessie, 240
jump-statement, 116
key_eq
   unordered associative containers, 665
key_equal
   unordered associative containers, 664

Draft
key_type
  unordered associative containers, 664
keyword, 1049
knuth_b, 851

label, 117
  case, 111, 113
  default, 111, 113, 114
  scope of, 32, 111
lambda()
  exponential_distribution<>, 863
lattice; see DAG
  subobject, 208
layout
  bit-field, 202
  class object, 195, 209
layout-compatible type, 63
left, 930
left shift
  undefined, 102
left shift operator, 102
length
  char_traits, 563, 565, 567–570, 572, 574–577, 579–581
  codecvt, 613
  sub_match, 1025
valarray, 885
length_error, 447, 557
  length_error, 447
less, 501
less_equal, 501
lexical conventions, 11
lexicographical_compare, 816
lgamma, 903

library
  C++ Standard, 397, 411–413, 415
  Standard C, 397, 405, 408, 410, 1079, 1082, 1085
limits
  implementation, 3
  <limits>, 418
  linear_congruential_engine<>, 841
    constructor, 842
    generation algorithm, 841
    state, 841
    template parameters, 841
    textual representation, 842
    transition algorithm, 841
linkage, 25, 49
  external, 49, 410, 411
  implementation-defined object, 157
  internal, 49
linkage-specification, 155
linkage specification, 154
  extern, 155
  implementation-defined, 155
list
  operator, 17, 285
  <list>, 670
list, 681
literal, 18, 77
  base of integer, 19
  char16_t, 20
  char32_t, 20
  character, 20
  decimal, 19
  double, 22
  float, 22
  floating point, 22
  hexadecimal, 19
  char, 21
  implementation-defined value of multicharacter, 20
  integer, 19
  long, 19
  long double, 22
  multicharacter, 20
  narrow-character, 20
  octal, 19
  type of character, 20
  type of floating point, 22
  type of integer, 19
  unsigned, 19

literal, 18
  llrint, 903
  llround, 903
load_factor
  unordered associative containers, 668
local_iterator, 664
locale, 1003–1005, 1015, 1019, 1024, 1045
  <locale>, 589
locale
  locale, 596
local class
friend, 231
member function in, 196
scope of, 204
local variable
destruction of, 116, 117
lock
weak_ptr, 542
log, 888, 903
complex, 829
log10, 888, 903
complex, 829
log1p, 903
log2, 903
logb, 903
logic_error, 446
logical_and, 501
logical_not, 501
logical_or, 501
lognormal_distribution<>, 867
constructor, 868
m(), 868
probability density function, 867
s(), 868
long
typedef and, 123
long-long-suffix, 19
long-suffix, 19
longjmp, 444
lookup
argument-dependent, 39
member name, 210
name, 25, 35
template name, 323
lookup_classname
regex_traits, 1018
Regular Expression Traits, 1005, 1046, 1047
lookup_collatename
regex_traits, 1018
Regular Expression Traits, 1005, 1046
lower_bound, 807
lowercase, 405
lrint, 903
lround, 903
lvalue, 66, 1073
modifiable, 66
lvalue reference, 65, 165
m()
fisher_f_distribution<>, 871
lognormal_distribution<>, 868
macro
function-like, 387
masking, 414
object-like, 387
main(), 52
implementation-defined linkage of, 52
implementation-defined parameters to, 52
parameters to, 52
return from, 52, 54
make_heap, 814
make_pair, 471
make_tuple, 476
malloc, 545, 1083
<map>, 702
map, 705
operator<, 708
operator==, 708
mask_array, 895
mask_array, 896
match_any, 1013, 1014
match_continuous, 1013, 1014, 1041
match_default, 1013
match_flag_type, 1013, 1046
match_not_bol, 1013, 1014
match_not_bow, 1013, 1014
match_not_eol, 1013, 1014
match_not_eow, 1013, 1014
match_not_null, 1013, 1014, 1041
match_prev_avail, 1013, 1014, 1041
match_results, 1030, 1039, 1042
as Sequence, 1030
begin, 1033
empty, 1032
end, 1033
format, 1033, 1034
match_results, 1032
matched, 1030
max_size, 1032
operator=, 1032
operator[], 1033
prefix, 1033
size, 1032
str, 1033
suffix, 1033
swap, 1034
matched, 1003
max, 815
  random_device, 852
  uniform random number generator requirement, 832
  valarray, 885
  xor_combine_engine<>, 850
max_bucket_count
  unordered associative containers, 668
max_element, 815
max_length
codecvt, 613
max_load_factor
  unordered associative containers, 668
max_size
  basic_string, 566
  array, 673
  match_results, 1032
mean
  normal_distribution<>, 866
  poisson_distribution<>, 861
mean()
  normal_distribution<>, 867
  poisson_distribution<>, 862
  student_t_distribution<>, 872
mem-initializer, 249
mem-initializer-id, 249
mem_fn, 507
mem_fun, 506, 507
mem_fun1_ref_t, 506
mem_fun1_t, 505
mem_fun_ref, 506, 507
mem_fun_ref_t, 506
mem_fun_t, 505
member, see also base class member
class static, 55
enumerator, 140
static, 199
template and static, 309
static, 91
member-declaration, 193
member-declarator, 193
member-specification, 193
member access operator
  overloaded, 288
member function
class, 195
  const, 198
  constructor and, 237
destructor and, 243
friend, 229
inline, 195
local class, 205
nested class, 233
overload resolution and, 268
static, 199, 200
union, 201
volatile, 198
member function call
  undefined, 197
member pointer to; see pointer to member, 65
member use
  static, 199
memchr, 585
<memory>, 514
memory model, 6
memory management, see also new, delete
merge, 809
  list, 687
mersenne_twister_engine<>, 842
  constructor, 843
generation algorithm, 842
state, 842
template parameters, 843
textual representation, 843
transition algorithm, 842
message
diagnostic, 2
messages, 643
messages_byname, 645
min, 815
random_device, 852
  uniform random number generator requirement, 832
  valarray, 885
min_element, 815
minmax, 815
minmax_element, 816
minstd_rand, 850
Draft
INDEX

INDEX  1144

minstd_rand0, 850
minus, 500
mismatch, 794
mod, 903
modf, 903
modulus
  implementation defined, 100
modulus, 500
money_get, 637
money_put, 639
moneypunct, 640
moneypunct_byname, 643
move, 468
movemove, 796
move_backward, 796
move_iterator, 766
  move_iterator, 768
mt19937, 851
multimap, 710
  operator<, 713
  operator==, 713
multiple inheritance, 207, 208
  virtual and, 216
multiplicative-expression, 100
multiplies, 500
multiset, 718
  operator<, 721
  operator==, 721
mutable, 123

n()
  chi_squared_distribution<>, 869
  fisher_f_distribution<>, 871
name, 17, 25, 77
  address of cv-qualified, 91
  dependent, 329, 330, 335
  elaborated enum, 135
  global, 33
  length of, 17
  macro, 388
  overloaded function, 263
  overloaded member, 194
  point of declaration, 30
  predefined macro, 394
  qualified, 41
  reserved, 410
  scope of, 29
  unqualified, 35
name
  locale, 597
  type_info, 436
namespace, 407, 1086
  global, 411
  unnamed, 142
namespaces, 141
  placeholders, 504
  regex_constants, 1012
name class, see class name
name hiding, 30, 34, 77, 78, 117
  class definition, 191
  function, 266
  overloading versus, 266
  user-defined conversion and, 240
name space
  label, 111
nan, 903
narrow
  basic_ios, 926
  ctype<char>, 611
  ctype, 606
NDEBUG, 409
nearbyint, 903
negate, 500
negative_binomial_distribution<>, 860
  constructor, 861
  discrete probability function, 860
  p(), 861
  t(), 861
nested-name-specifier, 78
nested class
  local class, 205
  scope of, 203
<new>, 412, 431
new, 55, 92, 94
  operator, 431, 434, 545
  array of class objects and, 95
  constructor and, 95
  default constructor and, 95
  exception and, 96
  initialization and, 95
  operator, 412, 432–434
  scoping and, 93

Draft
storage allocation, 93
type of, 245
unspecified constructor and, 97
unspecified order of evaluation, 97

new-declarator, 93
new-expression, 93
new-initializer, 93
new-placement, 93
new-type-id, 93
new_handler, 56, 435
next, 755
next_permutation, 817
nextafter, 903
nexttoward, 903
noboolalpha, 929
nonzero-digit, 16

none
bitset, 727
nonzero-digit, 18
norm, 830
complex, 828
normal distributions, 866–872
normal_distribution<>., 866

constructor, 867
mean, 866
mean(), 867
probability density function, 866
standard deviation, 866
stddev(), 867

noshowbase, 929
nouppercase, 929
noshowpoint, 929
noskipws, 929
notl, 502
not2, 502
not_equal_to, 501
notation
syntax, 5
nounitbuf, 930
nouppercase, 930

NTBS, 405, 989, 1094, 1096

static, 405
NTC16S, 406
static, 406
NTC32S, 406
static, 406

NTCTS, 399
nth_element, 806
NTMBS, 405, 406
static, 406
NTWCS, 406, 407
static, 407
NULL, 418
num_get, 616
num_put, 621

number
hex, 21
octal, 21

<numeric>, 898
numeric_limits, 419
numeric_limits, 64
numpunct, 626
numpunct_byname, 628

object, 6, 25, 66
complete, 6
definition, 27
delete, 97
static, 54
destructor and placement of, 244
linkage specification, 157
local static, 55
undefined deleted, 57
unnamed, 237

object-expression, 76
object class, see also class object
object lifetime, 57
object temporary, see temporary
object type, 62
oct, 931

octal-digit, 18
octal-escape-sequence, 20
octal-literal, 18

offsetof, 1083
ofstream, 908, 985

open
basic_filebuf, 989, 1000
basic_ifstream, 995
basic_ofstream, 998
messages, 644
openmode
ios_base, 917
operator, 286, 886, 887
  **=, 107
  +=, 92, 107
  -=, 107
  /=, 107
  operator()==()
    random number engine requirement, 834
  %=, 107
  &=, 107
  ^=, 107
  <<=, 107
  >>=, 107
  |==, 107
  add, 100
  address-of, 90
  assignment, 107, 407
  bitwise, 104
  bitwise AND, 104
  bitwise exclusive OR, 104
  bitwise inclusive OR, 104
  cast, 90, 98, 160
  class member access, 82
  comma, 107
  conditional expression, 105
  copy assignment, 256
  decrement, 83, 90, 92
  division, 100
  equality, 103
  function call, 79, 286
  greater than, 102
  greater than or equal to, 102
  increment, 83, 90, 91
  indirection, 90
  inequality, 103
  less than, 102
  less than or equal to, 102
  logical AND, 105
  logical negation, 90, 91
  logical OR, 105
  modulus, 100
  multiplication, 100
  multiplicative, 100
  one’s complement, 90, 91
  overloaded, 75
  pointer to member, 99
  pragma, 395
  precedence of, 8
  relational, 102
  scope resolution, 77, 78, 94, 196, 207, 218
  side effects and comma, 107
  side effects and logical AND, 105
  side effects and logical OR, 105
  sizeof, 90, 92
  subscripting, 79, 286
  unary, 90
  unary minus, 90, 91
  unary plus, 90, 91
operator
  overloaded, 285
operator, 286
operator delete, 94, 97, see also delete, 245
operator new, see also new, 94
operator!
  basic_ios, 927
  valarray, 884
operator!=, 467
  basic_string, 581
  complex, 827
  istreambuf_iterator, 777
  locale, 597
  reverse_iterator, 760
  type_info, 436
  bitset, 727
  queue, 689
  regex_iterator, 1040
  regex_token_iterator, 1044
  shared_ptr, 538
  stack, 693
  sub_match, 1025–1029
  tuple, 478
  valarray, 887
operator()
  locale, 597
  function, 512
  hash, 514
  reference_wrapper, 499
operator()()
  random number engine requirement, 834
  random_device, 852
  uniform random number generator requirement, 832
operator*
  auto_ptr, 1101
<table>
<thead>
<tr>
<th>Function/Operator</th>
<th>Line(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>back_insert_iterator</td>
<td>763</td>
</tr>
<tr>
<td>complex</td>
<td>827</td>
</tr>
<tr>
<td>front_insert_iterator</td>
<td>764</td>
</tr>
<tr>
<td>insert_iterator</td>
<td>766</td>
</tr>
<tr>
<td>istreambuf_iterator</td>
<td>777</td>
</tr>
<tr>
<td>ostreambuf_iterator</td>
<td>778</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>758</td>
</tr>
<tr>
<td>regex_iterator</td>
<td>1040</td>
</tr>
<tr>
<td>regex_token_iterator</td>
<td>1044</td>
</tr>
<tr>
<td>shared_ptr</td>
<td>537</td>
</tr>
<tr>
<td>valarray</td>
<td>886</td>
</tr>
<tr>
<td>operator**=</td>
<td>complex, 826</td>
</tr>
<tr>
<td>gslice_array</td>
<td>895</td>
</tr>
<tr>
<td>indirect_array</td>
<td>898</td>
</tr>
<tr>
<td>mask_array</td>
<td>896</td>
</tr>
<tr>
<td>slice_array</td>
<td>891</td>
</tr>
<tr>
<td>valarray</td>
<td>884</td>
</tr>
<tr>
<td>operator*+</td>
<td>basic_string, 579</td>
</tr>
<tr>
<td>complex</td>
<td>826</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>759, 761</td>
</tr>
<tr>
<td>valarray</td>
<td>884</td>
</tr>
<tr>
<td>valarray</td>
<td>886</td>
</tr>
<tr>
<td>operator*++</td>
<td>back_insert_iterator, 763</td>
</tr>
<tr>
<td>front_insert_iterator</td>
<td>764</td>
</tr>
<tr>
<td>insert_iterator</td>
<td>766</td>
</tr>
<tr>
<td>istreambuf_iterator</td>
<td>777</td>
</tr>
<tr>
<td>ostreambuf_iterator</td>
<td>779</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>758</td>
</tr>
<tr>
<td>regex_iterator</td>
<td>1041</td>
</tr>
<tr>
<td>regex_token_iterator</td>
<td>1044, 1045</td>
</tr>
<tr>
<td>operator*++=</td>
<td>basic_string, 567</td>
</tr>
<tr>
<td>complex</td>
<td>825, 826</td>
</tr>
<tr>
<td>gslice_array</td>
<td>895</td>
</tr>
<tr>
<td>indirect_array</td>
<td>898</td>
</tr>
<tr>
<td>mask_array</td>
<td>896</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>759</td>
</tr>
<tr>
<td>slice_array</td>
<td>891</td>
</tr>
<tr>
<td>valarray</td>
<td>884</td>
</tr>
</tbody>
</table>

Draft
indirect_array, 898  
mask_array, 896  
slice_array, 891  
bitset, 725  
valarray, 884  
operator<=, 467  
  basic_string, 582  
  reverse_iterator, 761  
  queue, 689  
  stack, 694  
  sub_match, 1025, 1026, 1028–1030  
  tuple, 479  
  valarray, 887  
operator=, 467  
  auto_ptr, 1101  
  back_insert_iterator, 762, 763  
  bad_alloc, 435  
  bad_cast, 437  
  bad_exception, 440  
  bad_typeid, 438  
  basic_string, 564  
  exception, 439  
  front_insert_iterator, 764  
  gslice_array, 894  
  indirect_array, 897  
  insert_iterator, 765  
  mask_array, 896  
  ostreambuf_iterator, 778  
  reverse_iterator, 757  
  slice_array, 891  
  valarray, 882  
  basic_regex, 1022  
  enable_shared_from_this, 544  
  function, 511  
  match_results, 1032  
  shared_ptr, 536  
  tuple, 475, 476  
  weak_ptr, 542  
operator==, 1101  
  basic_string, 580  
  complex, 827  
  istreambuf_iterator, 777  
  locale, 597  
  pair, 470  
  reverse_iterator, 760  
  type_info, 436  
operator==()  
  random number engine requirement, 834  
operator>, 467  
  basic_string, 582  
  reverse_iterator, 760  
  queue, 689  
  stack, 694  
  sub_match, 1026, 1028–1030  
  tuple, 479  
  valarray, 887  
operator>=, 467  
  basic_string, 582  
  reverse_iterator, 760  
  queue, 690  
  stack, 694  
  sub_match, 1026, 1028–1030  
  tuple, 479  
  valarray, 887  
operator>>, 467  
  basic_istream, 949  
  basic_string, 583  
  complex, 827  
  istream, 948  
  bitset, 727, 728  
  valarray, 886  
operator>>=, 467  
  gslice_array, 895  
  indirect_array, 895  
  mask_array, 896  
  slice_array, 891  
  bitset, 725  
  valarray, 884  
operator[], 467  
  basic_string, 567  
  map, 709  
  match_results, 1033  
  reverse_iterator, 760  

Draft
unordered_map, 732
valarray, 883
operator%
valarray, 886
operator%=
gslice_array, 895
indirect_array, 898
mask_array, 896
slice_array, 891
valarray, 884
operator&
bitset, 728
valarray, 886
operator&=
gslice_array, 895
indirect_array, 898
mask_array, 896
slice_array, 891
bitset, 724
valarray, 884
operator&&
valarray, 886, 887
operator~
bitset, 728
valarray, 886
operator~=
gslice_array, 895
indirect_array, 898
mask_array, 896
slice_array, 891
bitset, 724
valarray, 884
operator--
reverse_iterator, 759
operator<<()
random number engine requirement, 834
operator>>()
random number engine requirement, 834
operator~
valarray, 884
bitset, 726
operator|
bitset, 728
valarray, 886
operator|= 
gslice_array, 895
indirect_array, 898
mask_array, 896
slice_array, 891
bitset, 725
valarray, 884
operator left shift, see left shift operator
operator overloading, see also overloaded operator
operator right shift; right shift operator, 102
operator shift, see left shift operator, right shift operator
operator use
scope resolution, 200
optimization of temporary, see elimination of temporary
ordering
function template partial, 321
order of execution
base class constructor, 236
base class destructor, 243
constructor and static objects, 248
destructor, 243
destructor and array, 243
member constructor, 236
member destructor, 243
ostream, 908, 943
ostream_iterator, 773
output_iterator_tag, 777
stringstream, 908, 971
out
codecvt, 613
out_of_range, 447, 557, 724–727
output_iterator_tag, 753, 754
over-aligned type, 68
overflow, 75
undefined, 75
overflow
basic_filebuf, 991
basic_streambuf, 941
basic_stringbuf, 976
ostreambuf, 1091
overflow_error, 449, 724, 726
overflow_error, 449
Draft
overloaded function
  address of, 91, 284
overloaded operator, 285
    inheritance of, 286
overloading, 169, 191, 263, 320
    example of, 263
overloads
  floating point, 830
overload resolution contexts, 267
overrider
  final, 214
p()
  bernoulli_distribution, 858
  binomial_distribution<> , 859
  geometric_distribution<> , 860
  negative_binomial_distribution<>, 861
pair, 469, 474–476
  get, 471
  tuple interface to, 469
parameter, 3
  reference, 165
  scope of, 31
  void, 168
parameter-declaration, 168
parameterized type, see template
parameters
  macro, 388
  random number distribution, 836
parameter list
  variable, 80, 168
parameter type list, 169
partial_sort, 805
partial_sort_copy, 805
partial_sum, 900
partition, 803
pbackfail
  basic_filebuf, 990
  basic_streambuf, 941
  basic_stringbuf, 975
  strstreambuf, 1092
pbase
  basic_streambuf, 938
pbump
  basic_streambuf, 938
pcount
  ostrstream, 1096
  strstreambuf, 1091
  strstream, 1098
peek
  basic_istream, 954
period, 405
phases
  translation, 11
piecewise_constant_distribution<>, 873
  constructor, 874
  densities(), 875
  interval boundaries, 873
  intervals(), 874
  probability density function, 873
  weights, 874
placeholders, 504
placement syntax
  new, 95
plus, 500
pm-expression, 99
POD class, 190
POD struct, 190
POD union, 190
POF, 444
pointer
  zero, 73
void*, 65
pointer_to_binary_function, 505
pointer_to_unary_function, 504
pointer to member, 65, 99
Poisson distributions, 861–866
poisson_distribution<>, 861
  constructor, 862
  discrete probability function, 861
  mean, 861
  mean(), 862
polar, 830
  complex, 828
pop
  priority_queue, 692
pop_back
  basic_string, 571
pop_heap, 813
POSIX
  extended regular expressions, 1015
  regular expressions, 1015
postfix ++ and --
  overloading, 288
postfix ++ and --, 83
pow, 830, 888, 903
  complex, 829
pp-number, 16
pptr
  basic_streambuf, 938
precision
  ios_base, 604, 919
prefix
  L, 21, 23
prefix
  match_results, 1033
prefix ++ and --
  overloading, 288
prefix ++ and dcr, 91
preprocessing, 383
preprocessing-op-or-punc, 18
preprocessing-token, 14
preprocessor
  macro, 383
prev, 755
prev_permutation, 817
primary equivalence class, 1003
priority_queue
  priority_queue, 690
  priority_queue, 691
private, 221
probabilities()
  discrete_distribution<>, 873
probability density function, 836
  cauchy_distribution<>, 869
  chi_squared_distribution<>, 868
  exponential_distribution<>, 862
  extreme_value_distribution<>, 865
  fisher_f_distribution<>, 870
  gamma_distribution<>, 863
  general_pdf_distribution<>, 875
  lognormal_distribution<>, 867
  normal_distribution<>, 866
  piecewise_constant_distribution<>, 873
  student_t_distribution<>, 871
  uniform_real_distribution<>, 856
  weibull_distribution<>, 864
program, 49
  ill-formed, 2
  well-formed, 4
promotion
  floating point, 72
  integral, 72
  protected, 221
protection, see access control, 415
proxy
  istreambuf_iterator, 776
  pseudo-destructor-name, 81
  pseudo-destructor-name, 79
  ptr-operator, 160
  ptr_fun, 505
  ptrdiff_t, 101
  implementation defined type of, 101
pubimbue
  basic_streambuf, 935
public, 221
pubseekoff
  basic_streambuf, 936
pubseekpos
  basic_streambuf, 936
pubsetbuf
  basic_streambuf, 936
pubsync
  basic_streambuf, 936
punctuators, 17
  pure-specifier, 193
pure specifier, 193
push
  priority_queue, 692
  push_heap, 813
put
  basic_ostream, 967
  money_put, 639
  num_put, 622
  time_put, 635
put_money, 971
putback
  basic_istream, 954
pword
  ios_base, 921
qualification
  explicit, 41
  qualified-id, 78
<queue>, 671

Draft
queue, 688

<stdio>, 838–841

random number distribution
bernoulli_distribution<>, 857
binomial_distribution<>, 858
chi_squared_distribution<>, 868
discrete_distribution<>, 872
exponential_distribution<>, 862
extreme_value_distribution<>, 865
fisher_f_distribution<>, 870
gamma_distribution<>, 863
general_pdf_distribution<>, 875
gamma_distribution<>, 859
lognormal_distribution<>, 867
negative_binomial_distribution<>, 860
normal_distribution<>, 866
parameters, 836
piecewise_constant_distribution<>, 873
poisson_distribution<>, 861
probability density function, 836
requirements, 836–838
student_t_distribution<>, 871
uniform_int_distribution<>, 855
uniform_real_distribution<>, 856

random number distributions
Bernoulli, 857–861
normal, 866–872
Poisson, 861–866
sampling, 872–876
uniform, 855–857

random number engine
generation algorithm, 833
linear_congruential_engine<>, 841
mersenne_twister_engine<>, 842
requirements, 832–835
state, 833
subtract_with_carry_engine<>, 844
successor state, 833
transition algorithm, 833
with predefined parameters, 850–851

random number engine adaptor
discard_block_engine<>, 845
independent_bits_engine<>, 846
requirements, 835–836
shuffle_order_engine<>, 848
with predefined parameters, 850–851
xor_combine_engine<>, 849

random number generation, 831–876
random number generator, see uniform random number generator
random_access_iterator_tag, 753, 754
random_device, 851
constructor, 852
entropy(), 852
implementation leeway, 851
max, 852
min, 852
operator()(), 852
random_shuffle, 802
randomize()
    seed_seq, 854
range_error, 448
    range_error, 449
ranlux24, 851
ranlux24_base, 851
ranlux48, 851
ranlux48_base, 851
rbegin
    basic_string, 565
rdbuf
    basic_filebuf, 1000
    basic_ifstream, 995
    basic_ios, 925
    basic_istream, 980
    basic_ofstream, 997
    basic_ostringstream, 982
    basic_stringstream, 985
    istrstream, 1095
    ostrstream, 1096
    strstream, 1097
rdstate
    basic_ios, 928
read
    basic_istream, 954
readonly
    basic_istream, 954
real, 830
    complex, 828
realloc, 545
redefinition
typedef, 126
reentrancy, 415
ref
    reference_wrapper, 499
reference, 65
    assignment to, 107
call by, 80
call const, 186
direct binding of, 186
lvalue, 65
null, 166
rvalue, 65
sizeof, 92
reference-compatible, 185
reference-related, 185
reference_wrapper, 497
    cref, 499, 500
get, 499
operator(), 499
ref, 499
    reference_wrapper, 498
<regex>, 1006
regex, 1006
regex_constants, 1012
    error_type, 1014, 1016
match_flag_type, 1013
syntax_option_type, 1013
regex_error, 1016, 1019, 1046
regex_iterator, 1039
    end-of-sequence, 1040
    increment, 1041
operator! =, 1040
operator*, 1040
operator++, 1041
operator==, 1040
    regex_iterator, 1040
regex_match, 1034–1036
regex_replace, 1038, 1039
regex_search, 1036–1038
regex_token_iterator, 1041
    end-of-sequence, 1042
operator! =, 1044
operator*, 1044
operator++, 1044, 1045
operator==, 1042, 1044
    regex_token_iterator, 1043
regex_traits, 1017
    char_class_type, 1017
    isctype, 1018
    lookup_classname, 1018
    lookup_collatename, 1018
specializations, 1017
transform, 1018
transform_primary, 1018
translate, 1017
translate_nocase, 1017
region
    declarative, 25, 29
register, 123
register_callback
    ios_base, 922
regular expression, 1003–1047
    grammar, 1045
    matched, 1003
    requirements, 1004
Regular Expression Traits, 1045
    char_class_type, 1004
    isctype, 1005, 1046
    lookup_classname, 1005, 1046, 1047
    lookup_collatename, 1005, 1046
    requirements, 1004, 1017
    transform, 1005, 1047
    transform_primary, 1005, 1046, 1047
    translate, 1004, 1047
    translate_nocase, 1004, 1047
rehash
    unordered associative containers, 668
rel_ops, 466
relational-expression, 102
release
    auto_ptr, 1101
remainder, 903
remainder operator, see modulus operator
remove, 799
    list, 686
    remove_copy, 800
    remove_copy_if, 800
    remove_if, 799
remquo, 903
rend
    basic_string, 565
replace, 798
Draft
basic_string, 571
replace_copy, 798
replace_copy_if, 798
replace_if, 798
replacement
  macro, 387
representation
  object, 61
  value, 61
requirements, 401
  Allocator, 463
  Container, 663, 673, 674, 1030
  not required for unordered associated containers, 663
  not supported by unordered associated containers, 669
  container, 651
  CopyAssignable, 463
  CopyConstructible, 462
  Destructible, 463
  EqualityComparable, 462
  iterator, 743
  LessThanComparable, 462
  MoveAssignable, 462
  MoveConstructible, 462
numeric type, 819
random number distribution, 836–838
random number engine, 832–835
random number engine adaptor, 835–836
Regular Expression Traits, 1004, 1017
Sequence, 1030
Swappable, 463
uniform random number generator, 832
Unordered Associative Container, 663
reraise, 374
rescanning and replacement, 390
reserve
  basic_string, 566
  vector, 698
reserved identifier, 17
reset
  auto_ptr, 1101
  bitset, 725
  shared_ptr, 536
  weak_ptr, 542
resetiosflags, 968
resize
  basic_string, 566
deque, 679
list, 684
vector, 698, 699
valarray, 886
resolution
  argument matching, see overload
  function template overload, 368
  overload, 267
  overloaded function call resolution, see also argument
  matching, overload
  resolution overloading, see overload
scoping ambiguity, 211
template name, 323
template overload, 321
restriction, 413–415
static member local class, 201
address of bit-field, 202
anonymous union, 201
bit-field, 202
constructor, 236
copy assignment operator, 259
copy constructor, 258
destructor, 242, 243
extern, 124
local class, 205
overloading, 286
pointer to bit-field, 202
reference, 165
register, 123
static, 124
union, 201, 236
restrictions
  operator overloading, 286
result_of, 496
type, 497
result_type
  entity characterization based on, 831
  uniform random number generator requirement, 832
  xor_combine_engine<>, 850
rethrow, 374
return, 116, 117
  constructor and, 117
  reference and, 185
return statement, see also return

Draft
return type, 169
  overloading and, 263
reverse, 801
  list, 687
reverse_copy, 801
reverse_iterator, 755
  reverse_iterator, 757
rfind
  basic_string, 575
right, 930
right shift
  implementation defined, 102
right shift operator, 102
rint, 903
rotate, 801
rotate_copy, 802
round, 903
rounding, 73
rule
  as-if, 7
  one-definition, 27
runtime_error, 448
  runtime_error, 448
rvalue, 66
  lvalue conversion to, 70
  lvalue conversion to, 1073
rvalue reference, 65, 165
s()
  lognormal_distribution<>, 868
s-char, 22
s-char-sequence, 22
sampling distributions, 872–876
sbumpc
  basic_streambuf, 936
scalar type, 62
scalbn, 903
scan_is
  ctype<char>, 610
  ctype, 606
scan_not
  ctype<char>, 610
  ctype, 606
scientific, 931
scope, 25, 29
anonymous union at namespace, 201
  class, 33
destructor and exit from, 116
enumeration, 34
exception declaration, 32
function, 32
global, 33
global namespace, 33
iteration-statement, 114
local, 31
macro definition, 390
namespace, 32
overloading and, 266
potential, 29
selection-statement, 112
scope resolution operator, 41
search, 794
seed()
  random number engine requirement, 833
seed_seq
  constructor, 853
  get_seeds(), 854
  overview, 853
  randomize(), 854
  size(), 854
seekdir
  ios_base, 918
seekg
  basic_istream, 955
seekoff
  basic_filebuf, 992
  basic_streambuf, 939
  basic_stringbuf, 976
  strstreambuf, 1092
seekp
  basic_ostream, 962
seekpos
  basic_filebuf, 992
  basic_streambuf, 939
  basic_stringbuf, 976
  strstreambuf, 1093
selection-statement, 112
semantics
  class member, 82
sentry
  basic_istream, 946
basic_ostream, 961
sequence
  ambiguous conversion, 278
  implicit conversion, 277
  standard conversion, 69
  statement, 111
sequencing operator, see comma operator
<set>, 704
set, 715
  operator<, 717
  operator==, 717
  bitset, 725
set_difference, 811
set_intersection, 811
set_new_handler, 412, 435
set_symmetric_difference, 812
set_terminate, 412, 441
set_unexpected, 412, 440
set_union, 810
setbase, 969
setbuf
  basic_filebuf, 991
  basic_streambuf, 939
  streambuf, 1094
  strstreambuf, 1094
setf
  ios_base, 919
setfill, 969
setg
  basic_streambuf, 938
setiosflags, 968
setjmp, 411
setlocale, 405
setp
  basic_streambuf, 938
setprecision, 969
setstate
  basic_ios, 928
setw, 970
sgetc
  basic_streambuf, 936
sgetn
  basic_streambuf, 936
shared_from_this
  enable_shared_from_this, 544
  shared_ptr, 532, 544
~shared_ptr, 535
const_pointer_cast, 540
dynamic_pointer_cast, 539
get, 537
get_deleter, 540
operator!=, 538
operator*, 537
operator->, 537
operator<, 538
operator<<, 539
operator==, 536
operator===, 538
reset, 536
shared_ptr, 533
static_pointer_cast, 539
swap, 536, 539
unique, 537
use_count, 537
shift
  valarray, 885
shift-expression, 102
shift operator, see left shift operator, right shift operator
short
typedef and, 123
showbase, 929
showmanyc
  basic_filebuf, 990
  basic_streambuf, 939, 990
showpoint, 929
showpos, 929
shuffle_order_engine<>, 848
  constructor, 849
generation algorithm, 848
state, 848
template parameters, 848
textual representation, 849
transition algorithm, 848
side effect, 9
side effects, 9
sign, 22
signature, 3
signbit, 905
signed
typedef and, 123
signed integer type, 63
simple call wrapper, 496
Draft
simple-escape-sequence, 20

template-id, 298

simple-type-specifier, 133

sin, 888, 903
  complex, 830
sinf, 888, 903
  complex, 830

size
  basic_string, 566
  gslice, 893
  slice, 890
  array, 673, 675
  bitset, 727
  match_results, 1032

size()
  seed_seq, 854

size_t, 92

sizeof
  empty class, 189
  skips, 929
  slice, 889
  slice, 890
  slice_array, 890
  slice_array, 891

smanip
  implementation-defined, 968

smart pointers, 531–544

sngetc
  basic_streambuf, 936

sort, 804
  list, 687

sort_heap, 814

space
  white, 14

specialization
  class template, 299
  class template partial, 315
  template, 337
  template explicit, 344

special member function, see constructor, destructor, inline
  function, user-defined conversion, virtual function

specification
  template argument, 350

specifications
  C++ Standard Library exception, 416
  implementation-defined exception, 416
  Standard C library exception, 415

specifier
  declaration, 122
  explicit, 126
  friend, 415
  friend, 128
  function, 125
  inline, 125
  missing storage class, 124
  static, 123
  storage class, 123
  typedef, 126
  virtual, 126

specifier access, see access specifier

specifier type, see type specifier

splice
  list, 686

sputchc
  basic_streambuf, 936

sputc
  basic_streambuf, 937

sputn
  basic_streambuf, 937

sqrt, 888, 903
  complex, 830

<sstream>, 971

stable algorithm, 400

stable_partition, 803

stable_sort, 805

<stack>, 672

stack, 692

standard
  structure of, 5
  standard deviation
    normal_distribution<>, 866
  standard-layout class, 190
  standard-layout struct, 190
  standard-layout union, 190
  standard integer type, 63
  standard signed integer type, 63
  standard unsigned integer type, 63

start
  program, 52

start
  gslice, 893
slice, 890
startup program, 410, 412
state
discard_block_engine<>\', 845
independent_bits_engine<>\', 846
linear_congruential_engine<>\', 841
mersenne_twister_engine<>\', 842
object, 399
random number engine, 833
shuffle_order_engine<>\', 848
subtract_with_carry_engine<>\', 844
xor_combine_engine<>\', 849
state
fpos, 922
statement, 111
continue in for, 115
break, 116
compound, 111
continue, 116
declaration, 117
declaration in for, 116
declaration in switch, 114
do, 114, 115
empty, 111
expression, 111
for, 114, 115
goto, 111, 116, 117
if, 112, 113
iteration, 114
jump, 116
labeled, 111
null, 111
selection, 112
switch, 112, 113, 116
while, 114
statement, 111
static, 123
destruction of local, 118
linkage of, 49, 124
overloading and, 263
static_assert, 121, 122
static_pointer_cast
shared_ptr, 539
< stddef.h >, 20, 23
stddev()
sizeof, 23
type of, 23
<string>, 553
string-literal, 22
stringbuf, 908, 971
stringstream, 908
string literal, 22
  char16_t, 22, 23
  char32_t, 22, 23
  implementation-defined, 23
  narrow, 22, 23
type of, 23
  undefined change to, 23
  wide, 22, 23
strlen, 1090
strlen, 1090, 1096
strpbrk, 585
strrchr, 585
strstr, 585
strstream, 1096
  strstream, 1097
strstreambuf, 1088
  strstreambuf, 1089
struct
  standard-layout, 190
struct
class versus, 189
structure, 189
structure tag, see class name
student_t_distribution<>., 871
  constructor, 872
  mean(), 872
  probability density function, 871
sub-expression, 1004
sub_match, 1024
  compare, 1025
  length, 1025
  operator!=, 1025–1029
  operator<, 1025, 1026, 1028–1030
  operator<<, 1030
  operator<=, 1025, 1026, 1028–1030
  operator==, 1025–1029
  operator>, 1026, 1028–1030
  operator>>, 1026, 1028–1030
  str, 1025
subobject, 6
subscripting operator
  overloaded, 288
subsequence rule
  overloading, 281
substr
  basic_string, 577
subtract_with_carry_engine<>., 844
  carry, 844
  constructor, 844, 845
  generation algorithm, 844
  state, 844
template parameters, 844
  textual representation, 844
  transition algorithm, 844
subtraction
  implementation defined pointer, 101
subtraction operator, 101
successor state
  random number engine, 833
suffix
  E, 22
  F, 22
  f, 22
  L, 19, 22
  l, 19, 22
  U, 19
  u, 19
suffix
  match_results, 1033
sum
  valarray, 885
summary
  compatibility with ISO C, 1069
  scope rules, 35
summary, syntax, 1049
sungetc
  basic_streambuf, 937
swap, 797
  basic_string, 573, 583
  pair, 470
  vector, 698
  array, 675
  basic_regex, 1024
  function, 512, 513
  match_results, 1034
  shared_ptr, 536, 539

Draft
unordered_map, 733
unordered_multimap, 736
unordered_multiset, 742
unordered_set, 739
weak_ptr, 542, 543
swap_ranges, 797
sync
  basic_filebuf, 992
  basic_istream, 955
  basic_streambuf, 939
sync_with_stdio
  ios_base, 920
synonym, 144
type name as, 126
syntax
  class member, 82
syntax_option_type, 1013
    awk, 1013, 1015
    basic, 1013, 1015
    collate, 1013, 1015, 1047
    ECMAScript, 1013, 1015
    egrep, 1013, 1015
    extended, 1013, 1015
    grep, 1013, 1015
    icase, 1013, 1015
    nosubs, 1013, 1015
    optimize, 1013, 1015
t()
  binomial_distribution<>, 859
  negative_binomial_distribution<>, 861
table
  ctype<char>, 611
tan, 888, 903
  complex, 830
tanh, 888, 903
  complex, 830
target
  function, 512
target object, 495, 496
target_type
  function, 512
tellg
  basic_istream, 955
tellp
  basic_ostream, 962
template, 293
    definition of, 293
    function, 350
    member function, 308
    primary, 315
template-argument, 298
template-argument-list, 298
template-declaration, 293
template-id, 298
template-name, 298
template-parameter, 294
template-parameter-list, 293
termination
  program, 52, 54
terminology
  pointer, 65
test
  bitset, 727
textual representation
  discard_block_engine<>, 846
  independent_bits_engine<>, 847
  shuffle_order_engine<>, 849
  subtract_with_carry_engine<>, 844
  xor_combine_engine<>, 850
tgamma, 903
  <tgmath.h>, 901
this, 77
type of, 198
this pointer, see this
thousands_sep
  numpunct, 627
throw, 371
throw-expression, 371

Draft
throwing
  exception, 372
tie, 476
  basic_ios, 925
time_get, 630
time_getbyname, 634
time_put, 635
time_putbyname, 636
to_string
  bitset, 726
to_ulong
  bitset, 726
token, 18
token, 15
tolower, 599
  ctype<char>, 610
  ctype, 606
toupper, 599
  ctype<char>, 610
  ctype, 606
traits, 400
transform, 797
  collate, 629
  Regular Expression Traits, 1005, 1047
transform_primary
  Regular Expression Traits, 1005, 1046, 1047
transform_primary
  regex_traits, 1018
translation unit, 11, 49
  name and, 25
trigraph, 11
trivial class, 189
trivial class type, 95
trivial type, 95
truename
  numpunct, 627
trunc, 903
truncation, 73
try, 371
try-block, 371
\textless\texttt{tuple}\textgreater, 471
tuple, 471, 473, 675
  and array, 675
  and pair, 469
  get, 478
  make_tuple, 476
  operator\!=, 478
  operator\<, 478
  operator\<=, 479
  operator\==, 475, 476
  operator\>=, 478
  operator\>, 479
  tie, 476
  tuple, 473, 474
tuple_element, 471, 477, 675
tuple_size, 471, 477, 675
type, 25
  arithmetic, 64
  array, 64, 169
  bitmask, 404
  Boolean, 63
  char, 63
  char16_t, 64
  char32_t, 64
  character, 63
  character container, 398
  class and, 189
  compound, 64
  const, 130
  destination, 180
double, 64
dynamic, 2
  enumerated, 65, 403, 404
enumeration underlying, 139
example of incomplete, 62
extended integer, 63
extended signed integer, 63
extended unsigned integer, 63
float, 64
floating point, 63
function, 64, 168, 169
fundamental, 63
sizeof, 63
incomplete, 27, 31, 61, 70, 79–83, 85, 90, 92, 97, 107, 207
int, 63
integral, 63
long, 63
long double, 64
long long, 63
multi-level mixed pointer and pointer to member, 71
multi-level pointer to member, 71
object, 6
over-aligned, 68
POD, 62
pointer, 65
polymorphic, 214
short, 63
signed char, 63
signed integer, 63
standard integer, 63
standard signed integer, 63
standard unsigned integer, 63
static, 3
underlying wchar_t, 64
unsigned, 63
unsigned char, 63
unsigned int, 63
unsigned long, 63
unsigned long long, 63
unsigned short, 63
unsigned integer, 63
void, 64
volatile, 130
wchar_t, 64
type
result_of, 497
type-id, 160
type-id-list, 377
type-name, 133
type-parameter, 294
type-specifier
bool, 133
wchar_t, 133
type-specifier, 130
type_info, 85, 436
type_info::name
implementation-defined, 437
typedef
function, 170
typedef
overloading and, 264
typedef-name, 126
typedef
 typeid, 85
<typename>, 436
typename, 135
types
implementation-defined, 403
implementation-defined exception, 416
type checking
argument, 80
type conversion, explicit, see casting
type generator, see template
type name, 160
nested, 205
scope of nested, 205
type pun, 88
type specifier
char, 133
class, 189
double, 133
enum, 135
float, 133
int, 133
long, 133
short, 133
struct, 189
union, 189
unsigned, 133
void, 133
volatile, 132
uflow
basic_filebuf, 990
basic_streambuf, 941
Draft
uint16_t, 428
uint32_t, 428
uint64_t, 428
uint8_t, 428
uint_fast16_t, 428
uint_fast32_t, 428
uint_fast64_t, 428
uint_fast8_t, 428
uint_least16_t, 428
uint_least32_t, 428
uint_least64_t, 428
uint_least8_t, 428
uintmax_t, 428
uintptr_t, 428

unary function, 497, 498, 507
unary-expression, 90
unary-operator, 90
unary_function, 496, 514
unary_negate, 502

unary operator
interpretation of, 287
overloaded, 287

uncaught_exception, 441
undefined, 72, 399, 400, 410, 411, 413, 567, 882–885, 887, 888, 893, 898, 924
undefined behavior, 776
underflow
basic_filebuf, 990
basic_streambuf, 940
basic_stringbuf, 975
strstreambuf, 1092
underflow_error
underflow_error, 450
unexpected, 441
unexpected(), 380
unexpected_handler, 412, 440
unget
basic_istream, 955
uniform distributions, 855–857
uniform random number generator
requirements, 832
uniform_int_distribution, 855
a(), 856
b(), 856
constructor, 856
discrete probability function, 855
uniform_real_distribution, 856
a(), 857
b(), 857
constructor, 856
probability density function, 856
uninitialized_copy, 519
uninitialized_fill, 519
uninitialized_fill_n, 520
union
standard-layout, 190
union, 65, 201
class versus, 189
anonymous, 201
global anonymous, 201
unique, 800
list, 687
shared_ptr, 537
unique_copy, 800
unit
instantiation, 12
translation, 409–411
unitbuf, 930
universal-character-name, 13
unordered associative containers, 663–742
begin, 668
bucket, 668
bucket_count, 668
bucket_size, 668
clear, 667
complexity, 663
const_local_iterator, 664
count, 667
del, 668
equal_range, 668
equality function, 663
equivalent keys, 663, 733, 739
erase, 667
exception safety, 669
find, 667
hash function, 663
hash_function, 665
hasher, 664
insert, 666, 667
iterator invalidation, 669
iterators, 669
key_eq, 665
key_equal, 664  
key_type, 664  
lack of comparison operators, 663, 669  
load_factor, 668  
local_iterator, 664  
max_bucket_count, 668  
max_load_factor, 668  
rehash, 668  
requirements, 663, 669  
unique keys, 663, 730, 736  
unordered_map, 728, 730  
element access, 732  
operator[], 732  
swap, 733  
unique keys, 730  
unordered_map, 732  
unordered_multimap, 728, 733  
equivalent keys, 733  
swap, 736  
unordered_multimap, 735  
unordered_multiset, 729, 739  
equivalent keys, 739  
swap, 742  
unordered_multiset, 741  
unordered_set, 729, 736  
swap, 739  
unique keys, 736  
unordered_set, 738  
<unordered_map>, 728  
unordered_map, 728, 730  
element access, 732  
operator[], 732  
swap, 733  
unique keys, 730  
unordered_map, 732  
unordered_multimap, 728, 733  
equivalent keys, 733  
swap, 736  
unordered_multimap, 735  
unordered_multiset, 729, 739  
equivalent keys, 739  
swap, 742  
unordered_multiset, 741  
unordered_set, 729, 736  
swap, 739  
unique keys, 736  
unordered_set, 738  
unqualified-id, 77  
unsetf  
ios_base, 919  
unshift  
codecvt, 613  
unsigned  
typedef and, 123  
unsigned-suffix, 19  
unsigned integer type, 63  
unspecified, 431, 432, 436, 805, 976, 1090, 1092  
unspecified behavior, 885  
unwinding  
stack, 375  
upper_bound, 807  
uppercase, 405, 411  
uppercase, 929  
use_count  
shared_ptr, 537  
weak_ptr, 542  
use_facet  
locale, 598  
using-declaration, 145  
using-directive, 151  
<utility>, 466  
va_end, 411  
va_list, 411  
<valarray>, 876  
valarray, 879, 894  
valarray, 881  
value  
call by, 80  
null member pointer, 73  
null pointer, 73  
undefined unrepresentable integral, 73  
value  
is_bind_expression, 503  
is_placeholder, 503  
value-initialization, 178  
variable  
indeterminate uninitialized, 178  
<vector>, 672  
vector, 694  
operator<, 697  
operator==, 697  
vector, 697  
vector<bool>, 700  
virtual base class, 209  
virtual function, 214  
pure, 218, 219  
virtual function call, 218  
constructor and, 254  
destructor and, 254  
undefined pure, 219  
visibility, 35  
void*  
type, 65  
void&, 165  
volatile, 65  
constructor and, 199, 236  
destructor and, 199, 242  
implementation-defined, 132
overloading and, 265

wcserr, 912
wchar_t, 20, 23, 406, 585
    implementation-defined, 64
wcin, 912
wclog, 912
wcout, 912
wcschr, 585
wcsrchr, 585
wcspbrk, 585
wcsstr, 586
weak result type, 496
weak_ptr, 535, 540
    ~weak_ptr, 541
        expired, 542
        lock, 542
        operator<, 542
        operator=, 542
        reset, 542
        swap, 542, 543
        use_count, 542
        weak_ptr, 541
weibull_distribution<>, 864
    a(), 865
    b(), 865
    constructor, 865
    probability density function, 864
    weibull_distribution<>, 864
weights
    discrete_distribution<>, 873
    piecewise_constant_distribution<>, 874
wfilebuf, 908, 985
wifstream, 908
wofstream, 908
wstringstream, 908
write
    basic_ostream, 967
    ws, 956
    ws, 949
    wstreambuf, 908, 931
    wstringbuf, 908, 971
    wstringstream, 908
    X(X&), see copy constructor
xalloc
    ios_base, 921
xmax()
    general_pdf_distribution<>, 876
xmin()
    general_pdf_distribution<>, 876
xor_combine_engine<>, 849
    generation algorithm, 849
    max, 850
    result_type, 850
    state, 849
    template parameters, 850
    textual representation, 850
    transition algorithm, 849
xsgetn
    basic_streambuf, 939
xsputn
    basic_streambuf, 941
zero
    undefined division by, 75, 100
    undefined modulus, 75
    zero-initialization, 178