Working Draft, Standard for Programming Language C++

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

1 General
   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 Multi-threaded executions and data races ....................... 11
   1.11 Acknowledgments ............................................... 13

2 Lexical conventions ................................................ 15
   2.1 Phases of translation ............................................ 15
   2.2 Character sets .................................................. 16
   2.3 Trigraph sequences .............................................. 17
   2.4 Preprocessing tokens ............................................ 18
   2.5 Alternative tokens .............................................. 18
   2.6 Tokens .......................................................... 19
   2.7 Comments ....................................................... 19
   2.8 Header names .................................................... 19
   2.9 Preprocessing numbers ......................................... 20
   2.10 Identifiers ..................................................... 20
   2.11 Keywords ....................................................... 21
   2.12 Operators and punctuators ..................................... 21
   2.13 Literals ........................................................ 22

3 Basic concepts ....................................................... 31
   3.1 Declarations and definitions .................................... 31
   3.2 One definition rule .............................................. 33
   3.3 Declarative regions and scopes ................................ 35
   3.4 Name lookup ..................................................... 41
   3.5 Program and linkage ............................................ 55
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6</td>
<td>Start and termination</td>
<td>58</td>
</tr>
<tr>
<td>3.7</td>
<td>Storage duration</td>
<td>60</td>
</tr>
<tr>
<td>3.8</td>
<td>Object Lifetime</td>
<td>63</td>
</tr>
<tr>
<td>3.9</td>
<td>Types</td>
<td>67</td>
</tr>
<tr>
<td>3.10</td>
<td>Lvalues and rvalues</td>
<td>72</td>
</tr>
<tr>
<td>3.11</td>
<td>Alignment</td>
<td>74</td>
</tr>
<tr>
<td>4</td>
<td>Standard conversions</td>
<td>75</td>
</tr>
<tr>
<td>4.1</td>
<td>Lvalue-to-rvalue conversion</td>
<td>76</td>
</tr>
<tr>
<td>4.2</td>
<td>Array-to-pointer conversion</td>
<td>76</td>
</tr>
<tr>
<td>4.3</td>
<td>Function-to-pointer conversion</td>
<td>76</td>
</tr>
<tr>
<td>4.4</td>
<td>Qualification conversions</td>
<td>76</td>
</tr>
<tr>
<td>4.5</td>
<td>Integral promotions</td>
<td>78</td>
</tr>
<tr>
<td>4.6</td>
<td>Floating point promotion</td>
<td>78</td>
</tr>
<tr>
<td>4.7</td>
<td>Integral conversions</td>
<td>78</td>
</tr>
<tr>
<td>4.8</td>
<td>Floating point conversions</td>
<td>78</td>
</tr>
<tr>
<td>4.9</td>
<td>Floating integral conversions</td>
<td>79</td>
</tr>
<tr>
<td>4.10</td>
<td>Pointer conversions</td>
<td>79</td>
</tr>
<tr>
<td>4.11</td>
<td>Pointer to member conversions</td>
<td>79</td>
</tr>
<tr>
<td>4.12</td>
<td>Boolean conversions</td>
<td>80</td>
</tr>
<tr>
<td>4.13</td>
<td>Integer conversion rank</td>
<td>80</td>
</tr>
<tr>
<td>5</td>
<td>Expressions</td>
<td>81</td>
</tr>
<tr>
<td>5.1</td>
<td>Primary expressions</td>
<td>83</td>
</tr>
<tr>
<td>5.2</td>
<td>Postfix expressions</td>
<td>87</td>
</tr>
<tr>
<td>5.3</td>
<td>Unary expressions</td>
<td>98</td>
</tr>
<tr>
<td>5.4</td>
<td>Explicit type conversion (cast notation)</td>
<td>106</td>
</tr>
<tr>
<td>5.5</td>
<td>Pointer-to-member operators</td>
<td>108</td>
</tr>
<tr>
<td>5.6</td>
<td>Multiplicative operators</td>
<td>108</td>
</tr>
<tr>
<td>5.7</td>
<td>Additive operators</td>
<td>109</td>
</tr>
<tr>
<td>5.8</td>
<td>Shift operators</td>
<td>110</td>
</tr>
<tr>
<td>5.9</td>
<td>Relational operators</td>
<td>111</td>
</tr>
<tr>
<td>5.10</td>
<td>Equality operators</td>
<td>112</td>
</tr>
<tr>
<td>5.11</td>
<td>Bitwise AND operator</td>
<td>113</td>
</tr>
<tr>
<td>5.12</td>
<td>Bitwise exclusive OR operator</td>
<td>113</td>
</tr>
<tr>
<td>5.13</td>
<td>Bitwise inclusive OR operator</td>
<td>113</td>
</tr>
<tr>
<td>5.14</td>
<td>Logical AND operator</td>
<td>113</td>
</tr>
<tr>
<td>5.15</td>
<td>Logical OR operator</td>
<td>113</td>
</tr>
<tr>
<td>5.16</td>
<td>Conditional operator</td>
<td>114</td>
</tr>
<tr>
<td>5.17</td>
<td>Assignment and compound assignment operators</td>
<td>115</td>
</tr>
<tr>
<td>5.18</td>
<td>Comma operator</td>
<td>116</td>
</tr>
<tr>
<td>5.19</td>
<td>Constant expressions</td>
<td>116</td>
</tr>
<tr>
<td>6</td>
<td>Statements</td>
<td>119</td>
</tr>
<tr>
<td>6.1</td>
<td>Labeled statement</td>
<td>119</td>
</tr>
<tr>
<td>6.2</td>
<td>Expression statement</td>
<td>119</td>
</tr>
</tbody>
</table>
6.3 Compound statement or block .................................................. 119
6.4 Selection statements .............................................................. 120
6.5 Iteration statements ............................................................... 122
6.6 Jump statements ................................................................. 124
6.7 Declaration statement .......................................................... 125
6.8 Ambiguity resolution .............................................................. 126

7 Declarations ................................................................. 129
7.1 Specifiers ........................................................................... 130
7.2 Enumeration declarations ..................................................... 145
7.3 Namespaces ....................................................................... 148
7.4 The `asm` declaration ........................................................... 162
7.5 Linkage specifications ........................................................... 162

8 Declarators ................................................................. 167
8.1 Type names ........................................................................ 168
8.2 Ambiguity resolution ............................................................. 169
8.3 Meaning of declarators .......................................................... 171
8.4 Function definitions .............................................................. 184
8.5 Initializers .......................................................................... 186

9 Classes ................................................................. 197
9.1 Class names ....................................................................... 199
9.2 Class members ................................................................. 201
9.3 Member functions .............................................................. 203
9.4 Static members ................................................................. 206
9.5 Unions ............................................................................... 208
9.6 Bit-fields ............................................................................ 210
9.7 Nested class declarations .................................................... 211
9.8 Local class declarations ....................................................... 212
9.9 Nested type names .............................................................. 213

10 Derived classes ........................................................ 215
10.1 Multiple base classes .......................................................... 216
10.2 Member name lookup ......................................................... 218
10.3 Virtual functions ............................................................... 222
10.4 Abstract classes ............................................................... 226

11 Member access control ................................................ 229
11.1 Access specifiers ............................................................... 231
11.2 Accessibility of base classes and base class members ................. 232
11.3 Access declarations ............................................................ 234
11.4 Friends ............................................................................ 235
11.5 Protected member access .................................................... 239
11.6 Access to virtual functions .................................................. 240
11.7 Multiple access ................................................................. 241

Draft
16.7 Null directive ................................................. 400
16.8 Predefined macro names ................................. 400
16.9 Pragma operator ............................................. 401

17 Library introduction ........................................ 403
  17.1 Definitions ................................................ 403
  17.2 Additional definitions ................................. 406
  17.3 Method of description (Informative) .............. 406
  17.4 Library-wide requirements ......................... 414

18 Language support library .................................. 423
  18.1 Types .................................................. 423
  18.2 Implementation properties ......................... 424
  18.3 Integer types ......................................... 434
  18.4 Start and termination .............................. 436
  18.5 Dynamic memory management .................... 437
  18.6 Type identification ................................ 442
  18.7 Exception handling ................................ 445
  18.8 Other runtime support ................................ 450

19 Diagnostics library ......................................... 453
  19.1 Exception classes .................................... 453
  19.2 Assertions ............................................ 458
  19.3 Error numbers ........................................ 458
  19.4 System error support ............................... 459

20 General utilities library .................................. 473
  20.1 Requirements .......................................... 473
  20.2 Utility components .................................. 478
  20.3 Tuples ................................................ 485
  20.4 Metaprogramming and type traits ................ 494
  20.5 Function objects .................................... 506
  20.6 Memory ................................................ 531
  20.7 Date and time ......................................... 572

21 Strings library ............................................. 575
  21.1 Character traits ..................................... 575
  21.2 String classes ........................................ 581
  21.3 Class template \texttt{basic\_string} ................... 585
  21.4 Numeric Conversions ............................... 613
  21.5 Null-terminated sequence utilities .......... 615

22 Localization library ....................................... 619
  22.1 Locales ............................................... 619
  22.2 Standard \texttt{locale} categories .................. 634
  22.3 Standard code conversion facets ............. 678
  22.4 C Library Locales ................................... 679
## 23 Containers library

- **23.1 Container requirements** ......................................................... 681
- **23.2 Sequence containers** ............................................................. 701
- **23.3 Associative containers** ........................................................... 741
- **23.4 Unordered associative containers** ............................................ 768

## 24 Iterators library

- **24.1 Iterator requirements** ............................................................ 783
- **24.2 Header `<iterator>` synopsis** .................................................. 788
- **24.3 Iterator primitives** ................................................................. 791
- **24.4 Predefined iterators** ............................................................... 795
- **24.5 Stream iterators** ................................................................. 811

## 25 Algorithms library

- **25.1 Non-modifying sequence operations** ....................................... 830
- **25.2 Mutating sequence operations** ............................................... 834
- **25.3 Sorting and related operations** .............................................. 842
- **25.4 C Library algorithms** ............................................................ 857

## 26 Numerics library

- **26.1 Numeric type requirements** .................................................. 859
- **26.2 The floating-point environment** ............................................. 860
- **26.3 Complex numbers** ................................................................. 861
- **26.4 Random number generation** ................................................... 871
- **26.5 Numeric arrays** ................................................................. 917
- **26.6 Generalized numeric operations** ........................................... 938
- **26.7 C Library** ........................................................................ 941

## 27 Input/output library

- **27.1 Iostreams requirements** ....................................................... 947
- **27.2 Forward declarations** .......................................................... 948
- **27.3 Standard iostream objects** .................................................... 950
- **27.4 Iostreams base classes** ......................................................... 952
- **27.5 Stream buffers** ................................................................. 971
- **27.6 Formatting and manipulators** ............................................... 982
- **27.7 String-based streams** ....................................................... 1012
- **27.8 File-based streams** ............................................................. 1026

## 28 Regular expressions library

- **28.1 Definitions** ................................................................. 1043
- **28.2 Requirements** ................................................................. 1044
- **28.3 Regular expressions summary** ............................................ 1046
- **28.4 Header `<regex>` synopsis** .................................................. 1046
- **28.5 Namespace std::regex::constants** ...................................... 1052
- **28.6 Class regex_error** ............................................................ 1056
- **28.7 Class template regex_traits** ............................................. 1057
28.8 Class template basic_regex ........................................... 1059
28.9 Class template sub_match ............................................. 1064
28.10 Class template match_results ...................................... 1070
28.11 Regular expression algorithms .................................... 1075
28.12 Regular expression Iterators ...................................... 1079
28.13 Modified ECMAScript regular expression grammar ............ 1085

29 Atomic operations library ............................................. 1089
  29.1 Order and Consistency ............................................. 1091
  29.2 Lock-free Property ................................................ 1093
  29.3 Flag Type and Operations ....................................... 1093
  29.4 Other Types and Operations ..................................... 1095

30 Thread support library ............................................... 1105
  30.1 Requirements ..................................................... 1105
  30.2 Threads ........................................................... 1106
  30.3 Mutual exclusion .................................................. 1112
  30.4 Condition variables .............................................. 1126

A Grammar summary .................................................... 1133
  A.1 Keywords .......................................................... 1133
  A.2 Lexical conventions .............................................. 1133
  A.3 Basic concepts .................................................... 1138
  A.4 Expressions ........................................................ 1138
  A.5 Statements ......................................................... 1142
  A.6 Declarations ........................................................ 1143
  A.7 Declarators .......................................................... 1146
  A.8 Classes ............................................................... 1148
  A.9 Derived classes ..................................................... 1149
  A.10 Special member functions ..................................... 1149
  A.11 Overloading ........................................................ 1149
  A.12 Templates .......................................................... 1150
  A.13 Exception handling ............................................... 1151
  A.14 Preprocessing directives ...................................... 1151

B Implementation quantities ........................................... 1153

C Compatibility .......................................................... 1155
  C.1 C++ and ISO C ..................................................... 1155
  C.2 Standard C library ................................................. 1165

D Compatibility features .............................................. 1171
  D.1 Increment operator with bool operand .......................... 1171
  D.2 static keyword ...................................................... 1171
  D.3 Access declarations ............................................... 1171
  D.4 Implicit conversion from const strings .......................... 1171
  D.5 Standard C library headers ......................................... 1171

Draft
D.6 Old iostreams members ................................................................. 1172
D.7 char* streams ........................................................................ 1174
D.8 Binders ..................................................................................... 1184
D.9 auto_ptr ................................................................................... 1185

E Universal-character-names .......................................................... 1189

F Cross references ........................................................................... 1191
# 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>17</td>
</tr>
<tr>
<td>2</td>
<td>alternative tokens</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>keywords</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>alternative representations</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>Types of Integer Constants</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>escape sequences</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>string literal concatenations</td>
<td>28</td>
</tr>
<tr>
<td>8</td>
<td>relations on const and volatile</td>
<td>72</td>
</tr>
<tr>
<td>9</td>
<td>simple-type-specifiers and the types they specify</td>
<td>141</td>
</tr>
<tr>
<td>10</td>
<td>relationship between operator and function call notation</td>
<td>282</td>
</tr>
<tr>
<td>11</td>
<td>conversions</td>
<td>290</td>
</tr>
<tr>
<td>12</td>
<td>Library Categories</td>
<td>407</td>
</tr>
<tr>
<td>13</td>
<td>C++ Library Headers</td>
<td>414</td>
</tr>
<tr>
<td>14</td>
<td>C++ Headers for C Library Facilities</td>
<td>415</td>
</tr>
<tr>
<td>15</td>
<td>C++ Headers for Freestanding Implementations</td>
<td>415</td>
</tr>
<tr>
<td>16</td>
<td>Language support library summary</td>
<td>423</td>
</tr>
<tr>
<td>17</td>
<td>Header <code>&lt;cassert&gt;</code> synopsis</td>
<td>458</td>
</tr>
<tr>
<td>18</td>
<td>Header <code>&lt;cfloat&gt;</code> synopsis</td>
<td>434</td>
</tr>
<tr>
<td>19</td>
<td>Header <code>&lt;cstdlib&gt;</code> synopsis</td>
<td>436</td>
</tr>
<tr>
<td>20</td>
<td>Header <code>&lt;csetjmp&gt;</code> synopsis</td>
<td>451</td>
</tr>
<tr>
<td>21</td>
<td>Header <code>&lt;ctime&gt;</code> synopsis</td>
<td>451</td>
</tr>
<tr>
<td>22</td>
<td>Header <code>&lt;csignal&gt;</code> synopsis</td>
<td>451</td>
</tr>
<tr>
<td>23</td>
<td>Header <code>&lt;cstdarg&gt;</code> synopsis</td>
<td>451</td>
</tr>
<tr>
<td>24</td>
<td>Header <code>&lt;cstddef&gt;</code> synopsis</td>
<td>453</td>
</tr>
<tr>
<td>25</td>
<td>Header <code>&lt;cstdint&gt;</code> synopsis</td>
<td>451</td>
</tr>
<tr>
<td>26</td>
<td>Header <code>&lt;cstdio&gt;</code> synopsis</td>
<td>451</td>
</tr>
<tr>
<td>27</td>
<td>Diagnostics library summary</td>
<td>453</td>
</tr>
<tr>
<td>28</td>
<td>Header <code>&lt;cassert&gt;</code> synopsis</td>
<td>458</td>
</tr>
<tr>
<td>Table</td>
<td>Title</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>459</td>
</tr>
<tr>
<td>30</td>
<td>General utilities library summary</td>
<td>473</td>
</tr>
<tr>
<td>31</td>
<td>EqualityComparable requirements</td>
<td>474</td>
</tr>
<tr>
<td>32</td>
<td>LessThanComparable requirements</td>
<td>474</td>
</tr>
<tr>
<td>33</td>
<td>MoveConstructible requirements</td>
<td>474</td>
</tr>
<tr>
<td>34</td>
<td>CopyConstructible requirements</td>
<td>474</td>
</tr>
<tr>
<td>35</td>
<td>MoveAssignable requirements</td>
<td>474</td>
</tr>
<tr>
<td>36</td>
<td>CopyAssignable requirements</td>
<td>475</td>
</tr>
<tr>
<td>37</td>
<td>Swappable requirements</td>
<td>475</td>
</tr>
<tr>
<td>38</td>
<td>Destructible requirements</td>
<td>475</td>
</tr>
<tr>
<td>39</td>
<td>Descriptive variable definitions</td>
<td>475</td>
</tr>
<tr>
<td>40</td>
<td>Allocator requirements</td>
<td>477</td>
</tr>
<tr>
<td>41</td>
<td>Primary Type Category Predicates</td>
<td>497</td>
</tr>
<tr>
<td>42</td>
<td>Composite Type Category Predicates</td>
<td>497</td>
</tr>
<tr>
<td>43</td>
<td>Type Property Predicates</td>
<td>498</td>
</tr>
<tr>
<td>44</td>
<td>Type Property Queries</td>
<td>501</td>
</tr>
<tr>
<td>45</td>
<td>Type Relationship Predicates</td>
<td>501</td>
</tr>
<tr>
<td>46</td>
<td>Const-volatile modifications</td>
<td>502</td>
</tr>
<tr>
<td>47</td>
<td>Reference modifications</td>
<td>503</td>
</tr>
<tr>
<td>48</td>
<td>Sign modifications</td>
<td>503</td>
</tr>
<tr>
<td>49</td>
<td>Array modifications</td>
<td>504</td>
</tr>
<tr>
<td>50</td>
<td>Pointer modifications</td>
<td>505</td>
</tr>
<tr>
<td>51</td>
<td>Other transformations</td>
<td>505</td>
</tr>
<tr>
<td>52</td>
<td>Header <code>&lt;cstdlib&gt;</code> synopsis</td>
<td>572</td>
</tr>
<tr>
<td>53</td>
<td>Header <code>&lt;cstring&gt;</code> synopsis</td>
<td>572</td>
</tr>
<tr>
<td>54</td>
<td>Header <code>&lt;ctime&gt;</code> synopsis</td>
<td>572</td>
</tr>
<tr>
<td>55</td>
<td>Strings library summary</td>
<td>575</td>
</tr>
<tr>
<td>56</td>
<td>Traits requirements</td>
<td>576</td>
</tr>
<tr>
<td>57</td>
<td><code>basic_string(const Allocator&amp;)</code> effects</td>
<td>591</td>
</tr>
<tr>
<td>58</td>
<td><code>basic_string(const basic_string&amp;)</code> effects</td>
<td>591</td>
</tr>
<tr>
<td>59</td>
<td><code>basic_string(const basic_string&amp;, size_type, const Allocator&amp;)</code></td>
<td>591</td>
</tr>
<tr>
<td>60</td>
<td><code>basic_string(const charT*, size_type, const Allocator&amp;)</code></td>
<td>592</td>
</tr>
<tr>
<td>61</td>
<td><code>basic_string(const charT*, const Allocator&amp;)</code></td>
<td>592</td>
</tr>
<tr>
<td>62</td>
<td><code>basic_string(size_t, charT, const Allocator&amp;)</code></td>
<td>592</td>
</tr>
<tr>
<td>63</td>
<td><code>basic_string(const basic_string&amp;, const Allocator&amp;) and basic_string</code></td>
<td>593</td>
</tr>
<tr>
<td>64</td>
<td>operator=(const basic_string&lt;charT, traits, Allocator&amp;) effects</td>
<td>593</td>
</tr>
<tr>
<td>65</td>
<td>operator=(const basic_string&lt;charT, traits, Allocator&amp;) effects</td>
<td>594</td>
</tr>
<tr>
<td>66</td>
<td><code>compare()</code> results</td>
<td>607</td>
</tr>
<tr>
<td>67</td>
<td>Header <code>&lt;cctype&gt;</code> synopsis</td>
<td>617</td>
</tr>
<tr>
<td>68</td>
<td>Header <code>&lt;cwctype&gt;</code> synopsis</td>
<td>617</td>
</tr>
<tr>
<td>69</td>
<td>Header <code>&lt;cwchar&gt;</code> synopsis</td>
<td>617</td>
</tr>
<tr>
<td>70</td>
<td>Header <code>&lt;cstdlib&gt;</code> synopsis</td>
<td>618</td>
</tr>
<tr>
<td>71</td>
<td>Header <code>&lt;cstdliblib&gt;</code> synopsis</td>
<td>618</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>72</td>
<td>Header <code>&lt;cuchar&gt;</code> synopsis</td>
<td>618</td>
</tr>
<tr>
<td>73</td>
<td>Localization library summary</td>
<td>619</td>
</tr>
<tr>
<td>74</td>
<td>Locale Category Facets</td>
<td>623</td>
</tr>
<tr>
<td>75</td>
<td>Required Specializations</td>
<td>624</td>
</tr>
<tr>
<td>76</td>
<td><code>do_in/do_out</code> result values</td>
<td>644</td>
</tr>
<tr>
<td>77</td>
<td><code>do_unshift</code> result values</td>
<td>645</td>
</tr>
<tr>
<td>78</td>
<td>Integer Conversions</td>
<td>649</td>
</tr>
<tr>
<td>79</td>
<td>Length Modifier</td>
<td>649</td>
</tr>
<tr>
<td>80</td>
<td>Integer Conversions</td>
<td>653</td>
</tr>
<tr>
<td>81</td>
<td>Floating-point Conversions</td>
<td>654</td>
</tr>
<tr>
<td>82</td>
<td>Length Modifier</td>
<td>654</td>
</tr>
<tr>
<td>83</td>
<td>Numeric conversions</td>
<td>654</td>
</tr>
<tr>
<td>84</td>
<td>Fill padding</td>
<td>655</td>
</tr>
<tr>
<td>85</td>
<td><code>do_get_date</code> effects</td>
<td>663</td>
</tr>
<tr>
<td>86</td>
<td>Header <code>&lt;locale&gt;</code> synopsis</td>
<td>679</td>
</tr>
<tr>
<td>87</td>
<td>Containers library summary</td>
<td>681</td>
</tr>
<tr>
<td>88</td>
<td><code>ConstructibleAsElement&lt;A, T, Args&gt;</code> requirements</td>
<td>682</td>
</tr>
<tr>
<td>89</td>
<td>Container requirements</td>
<td>682</td>
</tr>
<tr>
<td>90</td>
<td>Reversible container requirements</td>
<td>685</td>
</tr>
<tr>
<td>91</td>
<td>Allocator-aware container requirements</td>
<td>686</td>
</tr>
<tr>
<td>92</td>
<td>Sequence container requirements (in addition to container)</td>
<td>687</td>
</tr>
<tr>
<td>93</td>
<td>Optional sequence container operations</td>
<td>690</td>
</tr>
<tr>
<td>94</td>
<td>Associative container requirements (in addition to container)</td>
<td>691</td>
</tr>
<tr>
<td>95</td>
<td>Container requirements that are not required for unordered associative containers</td>
<td>695</td>
</tr>
<tr>
<td>96</td>
<td>Unordered associative container requirements (in addition to container)</td>
<td>696</td>
</tr>
<tr>
<td>97</td>
<td>Iterators library summary</td>
<td>783</td>
</tr>
<tr>
<td>98</td>
<td>Relations among iterator categories</td>
<td>783</td>
</tr>
<tr>
<td>99</td>
<td>Input iterator requirements</td>
<td>785</td>
</tr>
<tr>
<td>100</td>
<td>Output iterator requirements</td>
<td>785</td>
</tr>
<tr>
<td>101</td>
<td>Forward iterator requirements</td>
<td>786</td>
</tr>
<tr>
<td>102</td>
<td>Bidirectional iterator requirements (in addition to forward iterator)</td>
<td>787</td>
</tr>
<tr>
<td>103</td>
<td>Random access iterator requirements (in addition to bidirectional iterator)</td>
<td>788</td>
</tr>
<tr>
<td>104</td>
<td>Algorithms library summary</td>
<td>819</td>
</tr>
<tr>
<td>105</td>
<td>Header <code>&lt;cstdlib&gt;</code> synopsis</td>
<td>857</td>
</tr>
<tr>
<td>106</td>
<td>Numerics library summary</td>
<td>859</td>
</tr>
<tr>
<td>107</td>
<td>Uniform random number generator requirements</td>
<td>872</td>
</tr>
<tr>
<td>108</td>
<td>Random number engine requirements</td>
<td>873</td>
</tr>
<tr>
<td>109</td>
<td>Random number engine adaptor requirements</td>
<td>875</td>
</tr>
<tr>
<td>110</td>
<td>Random number distribution requirements</td>
<td>876</td>
</tr>
<tr>
<td>111</td>
<td>Header <code>&lt;cmath&gt;</code> synopsis</td>
<td>942</td>
</tr>
<tr>
<td>112</td>
<td>Header <code>&lt;cstdlib&gt;</code> synopsis</td>
<td>943</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>113</td>
<td>Input/output library summary</td>
<td>947</td>
</tr>
<tr>
<td>114</td>
<td><code>fmtflags</code> effects</td>
<td>957</td>
</tr>
<tr>
<td>115</td>
<td><code>fmtflags</code> constants</td>
<td>957</td>
</tr>
<tr>
<td>116</td>
<td><code>iostate</code> effects</td>
<td>957</td>
</tr>
<tr>
<td>117</td>
<td><code>openmode</code> effects</td>
<td>958</td>
</tr>
<tr>
<td>118</td>
<td><code>seekdir</code> effects</td>
<td>958</td>
</tr>
<tr>
<td>119</td>
<td>Position type requirements</td>
<td>963</td>
</tr>
<tr>
<td>120</td>
<td><code>basic_ios::init()</code> effects</td>
<td>965</td>
</tr>
<tr>
<td>121</td>
<td><code>seekoff</code> positioning</td>
<td>1018</td>
</tr>
<tr>
<td>122</td>
<td><code>newoff</code> values</td>
<td>1018</td>
</tr>
<tr>
<td>123</td>
<td>File open modes</td>
<td>1030</td>
</tr>
<tr>
<td>124</td>
<td><code>seekoff</code> effects</td>
<td>1033</td>
</tr>
<tr>
<td>125</td>
<td>Header <code>&lt;cstdio&gt;</code> synopsis</td>
<td>1042</td>
</tr>
<tr>
<td>126</td>
<td>Header <code>&lt;cinttypes&gt;</code> synopsis</td>
<td>1042</td>
</tr>
<tr>
<td>127</td>
<td>regular expression traits class requirements</td>
<td>1044</td>
</tr>
<tr>
<td>128</td>
<td><code>syntax_option_type</code> effects</td>
<td>1054</td>
</tr>
<tr>
<td>129</td>
<td><code>regex_constants::match_flag_type</code> effects when obtaining a match against a</td>
<td>1055</td>
</tr>
<tr>
<td></td>
<td>character container sequence [first,last).</td>
<td></td>
</tr>
<tr>
<td>130</td>
<td><code>error_type</code> values in the C locale</td>
<td>1056</td>
</tr>
<tr>
<td>131</td>
<td><code>match_results</code> assignment operator effects</td>
<td>1072</td>
</tr>
<tr>
<td>132</td>
<td>Effects of <code>regex_match</code> algorithm</td>
<td>1075</td>
</tr>
<tr>
<td>133</td>
<td>Effects of <code>regex_search</code> algorithm</td>
<td>1077</td>
</tr>
<tr>
<td>134</td>
<td>Atomics library summary</td>
<td>1089</td>
</tr>
<tr>
<td>135</td>
<td><code>memory_order</code> effects</td>
<td>1091</td>
</tr>
<tr>
<td>136</td>
<td>Atomics for builtin types</td>
<td>1097</td>
</tr>
<tr>
<td>137</td>
<td>Atomics for standard typedef types</td>
<td>1098</td>
</tr>
<tr>
<td>138</td>
<td>Atomic arithmetic computations</td>
<td>1103</td>
</tr>
<tr>
<td>139</td>
<td>Thread support library summary</td>
<td>1105</td>
</tr>
<tr>
<td>140</td>
<td>Standard Macros</td>
<td>1166</td>
</tr>
<tr>
<td>141</td>
<td>Standard Values</td>
<td>1166</td>
</tr>
<tr>
<td>142</td>
<td>Standard Types</td>
<td>1166</td>
</tr>
<tr>
<td>143</td>
<td>Standard Structs</td>
<td>1166</td>
</tr>
<tr>
<td>144</td>
<td>Standard Functions</td>
<td>1167</td>
</tr>
<tr>
<td>145</td>
<td>C Headers</td>
<td>1171</td>
</tr>
<tr>
<td>146</td>
<td><code>strstreambuf(streamsize)</code> effects</td>
<td>1175</td>
</tr>
<tr>
<td>147</td>
<td><code>strstreambuf(void* (*)(size_t), void (*)(void*))</code> effects</td>
<td>1176</td>
</tr>
<tr>
<td>148</td>
<td><code>strstreambuf(charT*, streamsize, charT*)</code> effects</td>
<td>1176</td>
</tr>
<tr>
<td>149</td>
<td><code>seekoff</code> positioning</td>
<td>1179</td>
</tr>
<tr>
<td>150</td>
<td><code>newoff</code> values</td>
<td>1179</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.

C++ is a general purpose programming language based on the C programming language as described in ISO/IEC 9899:1990 Programming languages — C (1.2). In addition to the facilities provided by C, C++ provides additional data types, classes, templates, exceptions, namespaces, inline functions, operator overloading, function name overloading, references, free store management operators, and additional library facilities.

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
1.3 Definitions

1. 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.\(^1\)


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

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

5. 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 \(^1\) 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.
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) and the ref-qualifier (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, for which this International Standard 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.”

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.

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.

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 “opt”, so

\{ expression\_opt \}

3) This documentation also defines implementation-defined behavior; see 1.9.
Names for syntactic categories have generally been chosen according to the following rules:

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

### 1.7 The C++ memory model

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

- **Note:** the representation of types is described in 3.9. — end note —

A memory location is either an object of scalar type or a maximal sequence of adjacent bit-fields all having non-zero width. **Note:** Various features of the language, such as references and virtual functions, might involve additional memory locations that are not accessible to programs but are managed by the implementation. — end note —

Two threads of execution can update and access separate memory locations without interfering with each other.

- **Note:** Thus a bit-field and an adjacent non-bit-field are in separate memory locations, and therefore can be concurrently updated by two threads of execution without interference. The same applies to two bit-fields, if one is declared inside a nested struct declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field declaration. It is not safe to concurrently update two bit-fields in the same struct if all fields between them are also bit-fields, no matter what the sizes of those intervening bit-fields happen to be. — end note —

**Example:** A structure declared as

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

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

### 1.8 The C++ object model

The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An object is a region of

Draft
7 General 1.9 Program execution

storage. [Note: A function is not an object, regardless of whether or not it occupies storage in the way that objects do. — end note] An object is created by a definition (3.1), by a new-expression (5.3.4) or by the implementation (12.2) when needed. The properties of an object are determined when the object is created. An object can have a name (clause 3). An object has a storage duration (3.7) which influences its lifetime (3.8). An object has a type (3.9). The term object type refers to the type with which the object is created. Some objects are polymorphic (10.3); the implementation generates information associated with each such object that makes it possible to determine that object’s type during program execution. For other objects, the interpretation of the values found therein is determined by the type of the expressions (clause 5) used to access them.

2 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.

3 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 \(x\).

4 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.

5 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 trivial or standard-layout type (3.9) shall occupy contiguous bytes of storage.

6 [Note: C++ provides a variety of built-in types and several ways of composing new types from existing types (3.9). — end note]

1.9 Program execution [intro.execution]

1 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.\(^4\)

2 Certain aspects and operations of the abstract machine are described in this International Standard as implementation-defined (for example, \texttt{sizeof(int)}). These constitute the parameters of the abstract machine. Each implementation shall include documentation describing its characteristics and behavior in these respects.\(^5\) Such documentation shall define the instance of the abstract machine that corresponds to that implementation (referred to as the “corresponding instance” below).

3 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

\(^4\) 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.

\(^5\) This documentation also includes conditionally-supported constructs and locale-specific behavior. See 1.4.
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.

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

5 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).

6 The observable behavior of the abstract machine is its sequence of reads and writes to volatile data and calls to library I/O functions.\(^6\)

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 which are neither

— of type volatile\_std::\_sig\_atomic\_t, nor

— lock-free atomic objects (29.2)

are unspecified, and the value of any object not of type volatile\_std::\_sig\_atomic\_t in either of these two categories that is modified by the handler becomes undefined.

8 An instance of each object with automatic storage duration (3.7.2) 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).

9 The least requirements on a conforming implementation are:

— 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.\(^7\) For example, in the following fragment

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

\(^6\) 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.

\(^7\) Overloaded operators are never assumed to be associative or commutative.
the expression statement behaves exactly the same as

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

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

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

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

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

or

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

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

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:

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

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

void f() {
  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]

Draft
1.9 Program execution

[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.

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.

[Example:

```
i = v[i++];           // the behavior is undefined
i = 7, i++, i++;      // i becomes 9

i = ++i + 1;          // the behavior is undefined
i = i + 1;            // the value of i is incremented
```

—end example]

When calling a function (whether or not the function is inline), every value computation and side effect associated with any argument expression, or with the postfix expression designating the called function, is sequenced before execution of every expression or statement in the body of the called function. [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. Several contexts in C++ cause evaluation of a function call, even though no corresponding function call syntax appears in the translation unit.

8) As specified in 12.2, after a full-expression is evaluated, a sequence of zero or more invocations of destructor functions for temporary objects takes place, usually in reverse order of the construction of each temporary object.

9) In other words, function executions do not interleave with each other.

Draft
1 Under a hosted implementation, a C++ program can have more than one thread of execution (a.k.a. thread) running concurrently. The execution of each thread proceeds as defined by the remainder of this standard. The execution of the entire program consists of an execution of all of its threads. [Note: Usually the execution can be viewed as an interleaving of all its threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving, as described below. — end note] Under a freestanding implementation, it is implementation-defined whether a program can have more than one thread of execution.

2 The value of an object visible to a thread $T$ at a particular point might be the initial value of the object, a value assigned to the object by $T$, or a value assigned to the object by another thread, according to the rules below. [Note: In some cases, there may instead be undefined behavior. Much of this section is motivated by the desire to support atomic operations with explicit and detailed visibility constraints. However, it also implicitly supports a simpler view for more restricted programs. — end note]

3 Two expression evaluations conflict if one of them modifies a memory location and the other one accesses or modifies the same memory location.

4 The library defines a number of atomic operations (clause 29) and operations on locks (clause 30) that are specially identified as synchronization operations. These operations play a special role in making assignments in one thread visible to another. A synchronization operation is either an acquire operation or a release operation, or both, on one or more memory locations; the semantics of these are described below. In addition, there are relaxed atomic operations, which are not synchronization operations, and atomic read-modify-write operations, which have special characteristics, also described below. [Note: For example, a call that acquires a lock will perform an acquire operation on the locations comprising the lock. Correspondingly, a call that releases the same lock will perform a release operation on those same locations. Informally, performing a release operation on $A$ forces prior side effects on other memory locations to become visible to other threads that later perform an acquire operation on $A$. We do not include “relaxed” atomic operations as synchronization operations although, like synchronization operations, they cannot contribute to data races. — end note]

5 All modifications to a particular atomic object $M$ occur in some particular total order, called the modification order of $M$. If $A$ and $B$ are modifications of an atomic object $M$, and $A$ happens before $B$, then $A$ shall precede $B$ in the modification order of $M$, which is defined below. [Note: This states that the modification orders must respect happens before. — end note] [Note: There is a separate order for each scalar object. There is no requirement that these can be combined into a single total order for all objects. In general this will be impossible since different threads may observe modifications to different variables in inconsistent orders. — end note]

6 A release sequence on an atomic object $M$ is a maximal contiguous sub-sequence of side effects in the modification order of $M$, where the first operation is a release, and every subsequent operation
   — is performed by the same thread that performed the release, or
   — is a non-relaxed atomic read-modify-write operation.

7 An evaluation $A$ that performs a release operation on an object $M$ synchronizes with an evaluation $B$ that performs
an acquire operation on $M$ and reads a value written by any side effect in the release sequence headed by $A$. [Note: Except in the specified cases, reading a later value does not necessarily ensure visibility as described below. Such a requirement would sometimes interfere with efficient implementation. — end note] [Note: The specifications of the synchronization operations define when one reads the value written by another. For atomic variables, the definition is clear. All operations on a given lock occur in a single total order. Each lock acquisition “reads the value written” by the last lock release. — end note]

8 An evaluation $A$ happens before an evaluation $B$ if:
   — $A$ is sequenced before $B$, or
   — $A$ synchronizes with $B$, or
   — for some evaluation $X$, $A$ happens before $X$ and $X$ happens before $B$.

9 A visible side effect $A$ on an object $M$ with respect to a value computation $B$ of $M$ satisfies the conditions:
   — $A$ happens before $B$, and
   — there is no other side effect $X$ to $M$ such that $A$ happens before $X$ and $X$ happens before $B$.

The value of a non-atomic scalar object $M$, as determined by evaluation $B$, shall be the value stored by the visible side effect $A$. [Note: If there is ambiguity about which side effect to a non-atomic object is visible, then there is a data race, and the behavior is undefined. — end note] [Note: This states that operations on ordinary variables are not visibly reordered. This is not actually detectable without data races, but it is necessary to ensure that data races, as defined here, and with suitable restrictions on the use of atomics, correspond to data races in a simple interleaved (sequentially consistent) execution. — end note]

10 The visible sequence of side effects on an atomic object $M$, with respect to a value computation $B$ of $M$, is a maximal contiguous sub-sequence of side effects in the modification order of $M$, where the first side effect is visible with respect to $B$, and for every subsequent side effect, it is not the case that $B$ happens before it. The value of an atomic object $M$, as determined by evaluation $B$, shall be the value stored by some operation in the visible sequence of $M$ with respect to $B$. Furthermore, if a value computation $A$ of an atomic object $M$ happens before a value computation $B$ of $M$, and the value computed by $A$ corresponds to the value stored by side effect $X$, then the value computed by $B$ shall either equal the value computed by $A$, or be the value stored by side effect $Y$, where $Y$ follows $X$ in the modification order of $M$. [Note: This effectively disallows compiler reordering of atomic operations to a single object, even if both operations are “relaxed” loads. By doing so, we effectively make the “cache coherence” guarantee provided by most hardware available to C++ atomic operations. — end note] [Note: The visible sequence depends on the happens before relation, which depends on the values observed by loads of atomics, which we are restricting here. The intended reading is that there must exist an association of atomic loads with modifications they observe that, together with suitably chosen modification orders and the happens before relation derived as described above, satisfy the resulting constraints as imposed here. — end note]

11 The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior. [Note: It can be shown that programs that correctly use simple locks to prevent all data races, and use no other synchronization operations, behave as though the executions of their constituent threads were simply interleaved, with each observed value of an object being the last value assigned in that interleaving. This is normally referred to as “sequential consistency”. However, this applies only to race-free programs, and race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result must perform an undefined operation. — end note]
Note: Compiler transformations that introduce assignments to a potentially shared memory location that would not be modified by the abstract machine are generally precluded by this standard, since such an assignment might overwrite another assignment by a different thread in cases in which an abstract machine execution would not have encountered a data race. This includes implementations of data member assignment that overwrite adjacent members in separate memory locations. We also generally preclude reordering of atomic loads in cases in which the atomics in question may alias, since this may violate the "visible sequence" rules. — end note

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

1.11 Acknowledgments


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

All rights in these originals are reserved.
Chapter 2  Lexical conventions

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]

[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

The precedence among the syntax rules of translation is specified by the following phases.\(^\text{10}\)

1. Physical source file characters are mapped, in an implementation-defined manner, to the basic source character set (introducing new-line characters for end-of-line indicators) if necessary. The set of physical source file characters accepted is implementation-defined. 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.\(^\text{11}\) 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.\(^\text{11}\)

\(^{10}\) Implementations must behave as if these separate phases occur, although in practice different phases might be folded together.

\(^{11}\) A partial preprocessing token would arise from a source file ending in the first portion of a multi-character token that requires a terminating sequence of characters, such as a header-name that is missing the closing * or >. A partial comment would arise from a source file ending with an unclosed /* comment.
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 and universal-character-name in a character literal or a string literal, as well as each escape sequence in a character literal or a non-raw string literal, 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.¹²

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 un-instantiated 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 [lex.charset]

The basic source character set consists of 96 characters: the space character, the control characters representing horizontal tab, vertical tab, form feed, and new-line, plus the following 91 graphical characters:¹³

```
 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
```

¹² An implementation need not convert all non-corresponding source characters to the same execution character.

¹³ The glyphs for the members of the basic source character set are intended to identify characters from the subset of ISO/IEC 10646 which corresponds to the ASCII character set. However, because the mapping from source file characters to the source character set (described in translation phase 1) is specified as implementation-defined, an implementation is required to document how the basic source characters are represented in source files.
The universal-character-name construct provides a way to name other characters.

hex-quad:
  hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

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

The character designated by the universal-character-name \UNNNNNNNN is that character whose character short name in ISO/IEC 10646 is NNNNNNNN; the character designated by the universal-character-name \uNNNN is that character whose character short name in ISO/IEC 10646 is 0000NNNN. If the hexadecimal value for a universal-character-name corresponds to a surrogate code point (in the range 0xD800–0xDFFF, inclusive), the program is ill-formed. Additionally, if the hexadecimal value for a universal-character-name outside a character or string literal corresponds to a control character (in either of the ranges 0x00–0x1F or 0x7F–0x9F, both inclusive) or to a character in the basic source character set, the program is ill-formed.

3 The basic execution character set and the basic execution wide-character set shall each contain all the members of the basic source character set, plus control characters representing alert, backspace, and carriage return, plus a null character (respectively, null wide character), whose 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>??(</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:

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

becomes

Draft
2.4 Preprocessing tokens

```
#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

```
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
```

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.\(^{14}\)
```

\(^{14}\)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”.

Draft
In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling.\(^{15}\) The set of alternative tokens is defined in Table 2.

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

### 2.6 Tokens

There are five kinds of tokens: identifiers, keywords, literals,\(^{16}\) 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

\(\text{header-name:}\)

\(< \text{h-char-sequence} >\)

\"\text{q-char-sequence}"\)

\(\text{h-char-sequence:}\)

\text{h-char}

\text{h-char-sequence} \text{h-char}

\(\text{h-char:}\)

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

\(^{15}\) Thus the “stringized” values (16.3.2) of \{ and <: will be different, maintaining the source spelling, but the tokens can otherwise be freely interchanged.

\(^{16}\) Literals include strings and character and numeric literals.
2.9 Preprocessing numbers

q-char-sequence:
  q-char
  q-char-sequence q-char

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

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

2 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.\(^{(17)}\)

2.9 Preprocessing numbers [lex/ppnumber]

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

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

2 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 [lex/name]

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

1 An identifier is an arbitrarily long sequence of letters and digits. Each universal-character-name in an identifier shall designate a character whose encoding in ISO 10646 falls into one of the ranges specified in Annex E. Upper- and

\(^{(17)}\) Thus, sequences of characters that resemble escape sequences cause undefined behavior.
lower-case letters are different. All characters are significant.\(^\text{(18)}\)

2 In addition, some identifiers are reserved for use by C++ implementations and standard libraries (17.4.3.2.2) and shall not be used otherwise; no diagnostic is required.

### 2.11 Keywords

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

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

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:

\(^{18}\)On systems in which linkers cannot accept extended characters, an encoding of the universal-character-name may be used in forming valid external identifiers. For example, some otherwise unused character or sequence of characters may be used to encode the ‘\u’ in a universal-character-name. Extended characters may produce a long external identifier, but C++ does not place a translation limit on significant characters for external identifiers. In C++, upper- and lower-case letters are considered different for all identifiers, including external identifiers.
2.13 Literals

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

### 2.13.1 Integer literals

An *integer-literal* is a sequence of **decimal-literal** optionally followed by an **integer-suffix**.

**decimal-literal**: character consisting only of **nonzero-digit***

**octal-literal**: character consisting only of **octal-digit***

**hexadecimal-literal**: character consisting only of **hexadecimal-digit***

---

19) 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 5: Types of Integer Constants

<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 int</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>u or U</td>
<td>unsigned int</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>unsigned long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>l or L</td>
<td>long int</td>
<td>long int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long int</td>
</tr>
<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>long long int</td>
<td>long long int</td>
</tr>
<tr>
<td>Both u or U and ll or LL</td>
<td>unsigned long long int</td>
<td>unsigned long long int</td>
</tr>
</tbody>
</table>

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

---

20) 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

escape-sequence
universal-character-name

c-char:

any member of the source character set except

the single-quote ' , backslash \ , or new-line character
escapeshape-sequence
universal-character-name

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

simple-escape-sequence:
one of
\' \" \? \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'x', is a character literal of type char16_t. The value of a
char16_t literal containing a single c-char is equal to its ISO 10646 code point value, provided that the code point
is representable with a single 16-bit code unit. (That is, provided it is a basic multi-lingual plane code point.) If the
value is not representable within 16 bits, the program is ill-formed. A char16_t literal containing multiple c-chars
is ill-formed. A character literal that begins with the letter U, such as U'x', is a character literal of type char32_t.
The value of a char32_t literal containing a single c-char is equal to its ISO 10646 code point value. A char32_t literal containing multiple c-chars is ill-formed. 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-chars is implementation-defined.

Certain nongraphic 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 6: escape sequences

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-line NL(LF)</td>
<td>\n</td>
</tr>
<tr>
<td>horizontal tab</td>
<td>HT \t</td>
</tr>
<tr>
<td>vertical tab</td>
<td>VT \v</td>
</tr>
<tr>
<td>backspace</td>
<td>BS \b</td>
</tr>
<tr>
<td>carriage return</td>
<td>CR \r</td>
</tr>
<tr>
<td>form feed</td>
<td>FF \f</td>
</tr>
<tr>
<td>alert</td>
<td>BEL \a</td>
</tr>
<tr>
<td>backslash</td>
<td>\ \f</td>
</tr>
<tr>
<td>question mark</td>
<td>? ?</td>
</tr>
<tr>
<td>single quote</td>
<td>’ \’</td>
</tr>
<tr>
<td>double quote</td>
<td>” &quot;</td>
</tr>
<tr>
<td>octal number</td>
<td>ooo \ooo</td>
</tr>
<tr>
<td>hex number</td>
<td>hhh \xhhh</td>
</tr>
</tbody>
</table>

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 ]

21) They are intended for character sets where a character does not fit into a single byte.
2.13.3 Floating literals

floating-literal:
  fractional-constant exponent-part<opt> floating-suffix<opt>
  digit-sequence exponent-part floating-suffix<opt>

fractional-constant:
  digit-sequence<opt> . digit-sequence
digit-sequence

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

sign: one of
  + -

digit-sequence:
  digit
digit-sequence digit

floating-suffix: one of
  f l F L

A floating literal consists of an integer part, a decimal point, a fraction part, an \( \mathbf{e} \) or \( \mathbf{E} \), 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 \( \mathbf{e} \) (or \( \mathbf{E} \)) and the exponent (not both) can be omitted. The integer part, the optional decimal point and the optional fraction part form the \textit{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 \texttt{double} unless explicitly specified by a suffix. The suffixes \texttt{f} and \texttt{F} specify \texttt{float}, the suffixes \texttt{l} and \texttt{L} specify \texttt{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

string-literal:
  " s-char-sequence<opt> "
  u8" s-char-sequence<opt> "
  u" s-char-sequence<opt> "
  U" s-char-sequence<opt> "
  L" s-char-sequence<opt> "
  R raw-string
  u8R raw-string
  uR raw-string
  UR raw-string
  LR raw-string

s-char-sequence:
  s-char
digit-sequence
  s-char s-char
s-char:

any member of the source character set except
- the double-quote "," backslash \, or new-line character

escape-sequence
universal-character-name

raw-string:

" d-char-sequence_opt [ r-char-sequence_opt ] d-char-sequence_opt "

r-char-sequence:

r-char

r-char-sequence r-char

r-char:

any member of the source character set, except
- (1), a backslash \ followed by a u or U, or
- (2), a right square bracket ] followed by the initial d-char-sequence
  (which may be empty) followed by a double quote ".
universal-character-name

d-char-sequence:

d-char

d-char-sequence d-char

d-char:

any member of the basic source character set except:
- space, the left square bracket [, the right square bracket ],
- and the control characters representing horizontal tab,
  vertical tab, form feed, and newline.

1 A string literal is a sequence of characters (as defined in 2.13.2) surrounded by double quotes, optionally prefixed by R, u8, u8R, u, uR, U, UR, L, or LR, as in "...", R"...", u8"...", u8R"**[...]**", u"...", uR"*~ [...]*~ ", U"...", UR"zzz[...]zzz", L"...", or LR"[...]", respectively.

2 A string literal that has an R in the prefix is a raw string literal. The terminating d-char-sequence of a raw-string is the same sequence of characters as the initial d-char-sequence. A d-char-sequence shall consist of at most 16 characters.

3 [Note: A source-file new-line in a raw string literal results in a new-line in the resulting execution string-literal, unless preceded by a backslash. Assuming no whitespace at the beginning of lines in the following example, the assert will succeed:

```c
const char *p = R"[a\b
c"");
assert(strcmp(p, "ab\nc") == 0);
```

—end note ]

4 A string literal that does not begin with u8, u, U, or L is an ordinary string literal, and is initialized with the given characters.

5 A string literal that begins with u8, such as u8"asdf", is a UTF-8 string literal and is initialized with the given characters as encoded in UTF-8.

Draft
Ordinary string literals and UTF-8 string literals are also referred to as narrow string literals. A narrow string literal has type “array of \(n\) const char”, where \(n\) is the size of the string as defined below, and has static storage duration (3.7).

A string literal that begins with \(u\), such as \(u"\text{asdf}"\), 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 \(U"\text{asdf}"\), 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 \(L"\text{asdf}"\), 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 belowit 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. If a UTF-8 string literal token is adjacent to a wide string literal token, the program is ill-formed. Any other concatenations are conditionally supported with implementation-defined behavior. [Note: This concatenation is an interpretation, not a conversion. —end note] [Example: Here are some examples of valid concatenations:

<table>
<thead>
<tr>
<th>source</th>
<th>means</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{u}&quot;a&quot;</td>
<td>\text{u}&quot;b&quot;</td>
</tr>
<tr>
<td>\text{u}&quot;a&quot;</td>
<td>\text{u}&quot;b&quot;</td>
</tr>
<tr>
<td>\text{u}&quot;a&quot;</td>
<td>\text{u}&quot;b&quot;</td>
</tr>
</tbody>
</table>

Characters in concatenated strings are kept distinct. [Example: \"\text{x}\text{A}\" "\text{B}" contains the two characters ‘\text{xA}’ and ‘\text{B}’ after concatenation (and not the single hexadecimal character ‘\text{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 in non-raw string literals 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, universal-character-names, and other characters, plus one for the terminating U’\0’ or L’\0’. The size of a char16_t string literal is the total number of escape sequences, universal-character-names, and other characters, plus one for each character requiring a surrogate pair, plus one for the terminating u’\0’. [Note: The size of a char16_t string literal is the number of code units, not the number of characters. —end note] Within char32_t and char16_t literals, any universal-character-names shall be within the range 0x0 to 0x10FFFF. The size of a narrow string literal is the total number of escape sequences and other characters, plus at least one for the multibyte encoding of each universal-character-name, plus one for the terminating ’\0’.

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.

2.13.6 Pointer literals

pointer-literal:
  nullptr

The pointer literal is the keyword nullptr. It is an rvalue of type std::nullptr_t.
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\(^{223}\) (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-declaration (7.3.3), or a using-directive (7.3.4).

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

```cpp
int a; // defines a
eextern 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
};
extint X::y = 1; // defines X::y
eenum { up, down }; // defines up and down
namespace N { int d; } // defines N and N::d
namespace N1 = N; // defines N1
X anX; // defines anX
```

whereas these are just declarations:

```cpp
extern int a; // declares a
eextern const int c; // declares c
tint f(int); // declares f
struct S; // declares S
ttypedef int Int; // declares Int	eextern X anotherX; // declares anotherX
tusing N::d; // declares N::d
```

— end example ]

3 [ 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

```cpp
#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

\(^{223}\text{Appearing inside the braced-enclosed declaration-seq in a linkage-specification does not affect whether a declaration is a definition.}\)
3.2 One definition rule

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

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 a 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). 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.

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.

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:

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

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

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 \textit{new-expression} (5.3.4), or
- an \texttt{lvalue-to-rvalue} conversion is applied to an \texttt{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 \texttt{null pointer constant}, and has type other than \texttt{void *}, is converted to the \texttt{type pointer} to \( T \) or reference to \( T \) using an implicit conversion (clause 4), a \texttt{dynamic_cast} (5.2.7) or a \texttt{static_cast} (5.2.9), or
- a class member access operator is applied to an expression of type \( T \) (5.2.5), or
- the \texttt{typeid} operator (5.2.8) or the \texttt{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 \texttt{lvalue} of type \( T \) is assigned to (5.17), or
- the type \( T \) is the subject of an \texttt{alignof} expression (5.3.6) or an \texttt{alignas} specifier (7.1.7).

— end note ]

There can be more than one definition of a class type (clause 9), enumeration type (7.2), inline function with external linkage (7.1.2), class template (clause 14), non-static function template (14.5.6), static data member of a class template (14.5.1.3), member function of a class template (14.5.1.1), or template specialization for which some template parameters are not specified (14.7, 14.5.5) in a program provided that each definition appears in a different translation unit, and provided the definitions satisfy the following requirements. Given such an entity named \( D \) defined in more than one translation unit, then

- each definition of \( D \) shall consist of the same sequence of tokens; and
- in each definition of \( D \), corresponding names, looked up according to 3.4, shall refer to an entity defined within the definition of \( D \), or shall refer to the same entity, after overload resolution (13.3) and after matching of partial template specialization (14.8.3), except that a name can refer to a \texttt{const} object with internal or no linkage if the object has the same literal type in all definitions of \( D \), and the object is initialized with a constant expression (5.19), 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).24)

— 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:

```cpp
// translation unit 1:
struct X {
    X(int);
    X(int, int);
};
X::X(int = 0) { } //X(int) called by D()
class D: public X { }; D d2;

// translation unit 2:
struct X {
    X(int);
    X(int, int);
};
X::X(int = 0, int = 0) { } //X(int, int) called by D();
class D: public X { }; //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 require-
ments from the list above shall apply to names from the template’s enclosing scope used in the template defini-
tion (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 [basic.scope]

1 Every name is introduced in some portion of program text called a declarative region, which is the largest part of
the program in which that name is valid, that is, in which that name may be used as an unqualified name to refer to
the same entity. In general, each particular name is valid only within some possibly 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
dclaration of the same name. In that case, the potential scope of the declaration in the inner (contained) declarative
region is excluded from the scope of the declaration in the outer (containing) declarative region.

2 [Example: in

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

24) 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 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-directives (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-specifiers (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`.

```
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,

```
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.
2 The potential scope of a function parameter name (including one appearing in a lambda-parameter-declaration-clause) or of a function-local predefined variable in a function definition (8.4) begins at its point of declaration. If the function has a function-try-block the potential scope of a parameter or of a function-local predefined variable ends at the end of the last associated handler, otherwise it ends at the end of the outermost block of the function definition. A parameter name shall not be redeclared in the outermost block of the function definition nor in the outermost block of any handler associated with a function-try-block.

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

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

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

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

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

namespace N {
    int g(char a) { // overloads N::g(int)
        return l+a; // l is from unnamed namespace
    }
```
3.3 Declarative regions and scopes

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

int j() {    // OK: definition of N::j()
  return g(i); // calls N::g(int)
}

int q();     // error: different return type

— end example —

2 A namespace member can also be referred to after the :: scope resolution operator (5.1) applied to the name of its namespace or the name of a namespace which nominates the member’s namespace in a using-directive; see 3.4.3.2.

3 The outermost declarative region of a translation unit is also a namespace, called the global namespace. A name declared in the global namespace has global namespace scope (also called global scope). The potential scope of such a name begins at its point of declaration (3.3.1) 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 [basic.scope.class]

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 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 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:

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

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

Draft
3.3 Declarative regions and scopes

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

typedef int I;
class D {
    typedef I I; // error, even though no reordering involved
};

— end example }

2 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

1 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

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

2 A class name (9.1) or enumeration name (7.2) can be hidden by the name of 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.

3 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.

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

```c
typedef int f;
namespace N {
    struct A {
        friend void f(A &); // f is the typedef, not the friend
        operator int(); // function: equivalent to int(a)
        void g(A a) {
            int i = f(a);  // f is the typedef, not the friend
                            // function: equivalent to int(a)
        }
    };
};
```
Because the expression is not a function call, the argument-dependent name lookup (3.4.2) does not apply and the friend function f is not found. — end note]
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\(^ {25}\) 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. \([\text{Example:}\]

\begin{verbatim}
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{verbatim}

— end example \]

A name used in the definition of a class \(X\) outside of a member function body or nested class definition\(^ {26}\) 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),\(^ {27}\) 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.

\([\text{Example:}\]

\begin{verbatim}
namespace M {
    class B { }
}
\end{verbatim}

\(^{25}\)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.

\(^{26}\) 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.

\(^{27}\) 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 ]

8 A name used in the definition of a member function (9.3) of class X following the function’s declarator-id 28) 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) 29) 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 { }
    namespace M {
      namespace N {
        class X : public B {
          void f();
        };
      }
    }

28) 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.

29) 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.
3.4 Name lookup

```cpp
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:

```cpp
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:

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

—end example]

A name used in the handler for a function-try-block (clause 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-declarations 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]
— 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 \( X \).

If an associated namespace is an inline namespace (7.3.1), its enclosing namespace is also included in the set. If an associated namespace directly contains inline namespaces, those inline namespaces are also included in the set. In addition, if the argument is the name or address of a set of overloaded functions and/or function templates, its associated classes and namespaces are the union of those associated with each of the members of the set: 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.

3 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:

    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-directives 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 enumerator can be referred to after the `::` scope resolution operator (5.1) applied to a nested-name-specifier that nominates its class, 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: C::X C::arr[C::number];
```

— end example]

A name prefixed by the unary scope operator `::` (5.1) is looked up in global scope, in the translation unit where it is used. The name shall be declared in global namespace scope or shall be a name whose declaration is visible in global scope because of a using-directive (3.4.3.2). The use of `::` allows a global name to be referred to even if its identifier has been hidden (3.3.8).
3.4 Name lookup

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

If a pseudo-destructor-name (5.2.4) contains a nested-name-specifier, the type-names are looked up as types in the scope designated by the nested-name-specifier. Similarly, in a qualified-id of the form:

```cpp
::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 [class.qual]

If the nested-name-specifier of a qualified-id nominates a class, the name specified after the nested-name-specifier is looked up in the scope of the class (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).

In a lookup in which the constructor is an acceptable lookup result, if and the nested-name-specifier nominates a class C:
— if the name specified after the nested-name-specifier, when looked up in C, is the injected-class-name of C (clause 9), or
— if the name specified after the nested-name-specifier is the same as the identifier or the simple-template-id’s
template-name in the last component of the nested-name-specifier,

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.—end note] Such a constructor name shall be used only in the declarator-id of a declaration that names a constructor
or in a using-declaration. [Example:

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

B::A ba; // object of type A
A::A a; // error, A::A is not a type name
struct A::A a2; // object of type A
```

—end example]

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

### 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-directives in X and its used
namespaces, except that using-directives that nominate non-inline namespaces (7.3.1) 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:

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

namespace Z {
    void h(double);
}
```
namespace A {
    using namespace Y;
    void f(int);
    void g(int);
    int i;
}

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

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

void h() {
    AB::g();  // g is declared directly in AB,
               // therefore S is { AB::g() } and AB::g() is chosen
    AB::f(1);  // f is not declared directly in AB so the rules are
               // applied recursively to A and B;
               // namespace Y is not searched and Y::f(float)
               // is not considered;
               // S is { A::f(int), B::f(char) } and overload
               // resolution chooses A::f(int)
    AB::f('c');  // as above but resolution chooses B::f(char)
    AB::x++;    // x is not declared directly in AB, and
                // is not declared in A or B, so the rules are
                // applied recursively to Y and Z,
                // S is { } so the program is ill-formed
    AB::i++;    // i is not declared directly in AB so the rules are
                // applied recursively to A and B,
                // S is { A::i, B::i } so the use is ambiguous
                // and the program is ill-formed
    AB::h(16.8);  // h is not declared directly in AB and
                  // not declared directly in A or B so the rules are
                  // applied recursively to Y and Z,
                  // S is { Y::h(int), Z::h(double) } and overload
                  // resolution chooses Z::h(double)
}

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

namespace A {

Draft
4 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;  
}
```
3.4 Name lookup

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 {A::a}
    B::b++;  // OK: b declared directly in B, S is {B::b}
}

— end example ]

5 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:

    namespace A {
        struct x { };  // OK, A::x (of type int)
        int x;
        int y;
    }

    namespace B {
        struct y { };  // ambiguous, A::y or B::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 ]

6 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:

    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:

Draft
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

1. 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).

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

   ```
   class-key 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.

3. 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
   ```
3.4 Name lookup

at global scope.

```c
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
};
```

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

— end example }

3.4.5 Class member access  

1 In a class member access expression (5.2.5), if the . or -> token is immediately followed by an identifier followed by a <, the identifier must be looked up to determine whether the < is the beginning of a template argument list (14.2) or a less-than operator. The identifier is first looked up in the class of the object expression. If the identifier is not found, it is then looked up in the context of the entire postfix-expression and shall name a class 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 shall 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

   ```c
   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

```plaintext
::class-name-or-namespace-name::...
```

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

6 If the `nested-name-specifier` contains a `simple-template-id` (14.2), the names in its `template-arguments` are looked up in the context in which the entire `postfix-expression` occurs.

7 If the `id-expression` is a `conversion-function-id`, its `conversion-type-id` 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

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

### 3.5 Program and linkage

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

```plaintext
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.

5 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.

6 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]

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

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

    void middle() {
        q();          // error: q not yet declared
    }
}
```
void q() { /* ... */ }  // definition of X::q
}
void q() { /* ... */ }  // some other, unrelated q
— end example ]

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)\textsuperscript{30}; 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:

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.

\textsuperscript{30} 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.
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]

### 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 (NTMBSs) (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 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,

```c
inline double fd() { return 1.0; }
extern double d1;
double d2 = d1; // unspecified:
    // may be statically initialized to 0.0 or
    // dynamically initialized to 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.31) [Example:

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

31) An object defined in namespace scope having initialization with side-effects must be initialized even if it is not used (3.7.1).
3.7 Storage duration

---

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

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 `operator new` (5.3.4).

The storage class specifiers `static` and `auto` are related to storage duration as described below.

3.7.1 Static storage duration

All objects which neither have dynamic storage duration nor are local have 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 `static` can be used to declare a local variable with static storage duration. [Note: 6.7 describes the initialization of local static variables; 3.6.3 describes the destruction of local static variables. — end note]

4 The keyword `static` applied to a class data member in a class definition gives the data member static storage duration.

3.7.2 Automatic storage duration

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

2 [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

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

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 (17.4.3.5). The following allocation and deallocation functions (18.5) are implicitly declared in global scope in each translation unit of a program.
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();

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 [basic.stc.dynamic.allocation]

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

2 The allocation function attempts to allocate the requested amount of storage. If it is successful, it shall return the address of the start of a block of storage whose length in bytes shall be at least as large as the requested size. There are no constraints on the contents of the allocated storage on return from the allocation function. The order, contiguity, and initial value of storage allocated by successive calls to an allocation function are unspecified. The pointer returned shall be suitably aligned so that it can be converted to a pointer of any complete object type with a fundamental alignment requirement (3.11) and then used to access the object or array in the storage allocated (until the storage is explicitly deallocated by a call to a corresponding deallocation function). Even if the size of the space requested is zero, the request can fail. If the request succeeds, the value returned shall be a non-null pointer value (4.10) p0 different from any previously returned value p1, unless that value p1 was subsequently passed to an operator delete. The effect of dereferencing a pointer returned as a request for zero size is undefined.32)

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

32) The intent is to have operator new() implementable by calling std::malloc() or std::calloc(), so the rules are substantially the same. C++ differs from C in requiring a zero request to return a non-null pointer.
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 functions are declared in a namespace scope other than global scope or declared static in global scope.

Each deallocation function shall return void and its first parameter shall be void*. A deallocation function 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, the second of which has type std::size_t (18.1), then this function is a usual deallocation function. Similarly, if a class T has a member deallocation function named operator delete[] with exactly one parameter, then that function is a usual (non-placement) deallocation function. If class T does not declare such an operator delete[] but does declare a member deallocation function named operator delete[] with exactly two parameters, the second of which has type std::size_t, then this function is a usual deallocation function. A deallocation function can be an instance of a function template. Neither the first parameter nor the return type shall depend on a template parameter. [ Note: that is, a deallocation function template shall have a first parameter of type void* and a return type of void (as specified above). — end note ] A deallocation function template shall have two or more function parameters. A template instance is never a usual deallocation function, regardless of its signature.

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.33)

3.7.4 Duration of subobjects  

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

3.8 Object Lifetime

The lifetime of an object is a runtime property of the object. 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

33) On some implementations, it causes a system-generated runtime fault.
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.

[Note: the lifetime of an array object or of an object of trivial 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]

The properties ascribed to objects throughout this International Standard apply for a given object only during its lifetime. [Note: in particular, before the lifetime of an object starts and after its lifetime ends there are significant restrictions on the use of the object, as described below, in 12.6.2 and in 12.7. Also, the behavior of an object under construction and destruction might not be the same as the behavior of an object whose lifetime has started and not ended. 12.6.2 and 12.7 describe the behavior of objects during the construction and destruction phases. — end note]

A program may end the lifetime of any object by reusing the storage which the object occupies or by explicitly calling the destructor for an object of a class type with a non-trivial destructor. For an object of a class type with a non-trivial destructor, the program is not required to call the destructor explicitly before the storage which the object occupies is reused or released; however, if there is no explicit call to the destructor or if a delete-expression (5.3.5) is not used to release the storage, the destructor shall not be implicitly called and any program that depends on the side effects produced by the destructor has undefined behavior.

Before the lifetime of an object has started but after the storage which the object will occupy has been allocated 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 \( \text{void}^* \), 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-trivial 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 \( \text{void}^* \), or to \( \text{void}^* \) and subsequently to \( \text{char}^* \), or \( \text{unsigned char}^* \)).
- 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();
};
```]

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

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

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

— end example —

6 Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any 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-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 static_cast (5.2.9) except when the conversion is ultimately to cv char& or cv unsigned char&, or
— the lvalue is used as the operand of a dynamic_cast (5.2.7) or as the operand of typeid.

7 If, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, a new object is created at the storage location which the original object occupied, a pointer that pointed to the original object, a reference that referred to the original object, or the name of the original object will automatically refer to the new object and, once the lifetime of the new object has started, can be used to manipulate the new object, if:

— 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 T and the new object is a most derived object of type T (that is, they are not base class subobjects). [Example:
3.8 Object Lifetime

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

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

C c1;
C c2;
c1 = c2;       // well-defined
  // 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();
};
```

[35]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.

Draft
const B b;

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

— end example ]

3.9 Types [basic.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 ]

2 For any object (other than a base-class subobject) of 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 ]

3 For any trivial type \( T \), if two pointers to \( T \) point to distinct \( T \) objects \( \text{obj1} \) and \( \text{obj2} \), where neither \( \text{obj1} \) nor \( \text{obj2} \) is a base-class subobject, if the value of \( \text{obj1} \) is copied into \( \text{obj2} \), using the \texttt{std::memcpy} library function, \( \text{obj2} \) shall subsequently hold the same value as \( \text{obj1} \). [ Example:

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

4 The object representation of an object of type \( T \) is the sequence of \( N \) unsigned char objects taken up by the object of type \( T \), where \( N \) equals \texttt{sizeof(T)}. The value representation of an object is the set of bits that hold the value of type \( T \). For 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.\(^{37}\)

\(^{36}\) By using, for example, the library functions (17.4.1.2) \texttt{std::memcpy} or \texttt{std::memmove}.

\(^{37}\) The intent is that the memory model of C++ is compatible with that of ISO/IEC 9899 Programming Language C.
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 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:

```cpp
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
    arrpp++; // 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—

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

Arithmetic types (3.9.1), enumeration types, pointer types, pointer to member types (3.9.2), and std::nullptr_t, and cv-qualified versions of these types (3.9.3) are collectively called scalar types. Scalar types, POD classes (clause 9), arrays of such types and cv-qualified versions of these types (3.9.3) are collectively called POD types. Scalar types, trivial class types (clause 9), arrays of such types and cv-qualified versions of these types (3.9.3) are collectively called POD types.

Draft
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.

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 standard-layout structs and standard-layout unions are described in 9.2. — end note ]

### 3.9.1 Fundamental types

Objects declared as characters (char) shall be large enough to store any member of the implementation’s basic character set. If a character from this set is stored in a character object, the integral value of that character object is equal to the value of the single character literal form of that character. It is implementation-defined whether a char object can hold negative values. Characters can be explicitly declared unsigned or signed. Plain char, signed char, and unsigned char are three distinct types. 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.

There are five standard signed integer types: “signed char”, “short int”, “int”, “long int”, and “long long int”. In this list, each type provides at least as much storage as those preceding it in the list. There may also be implementation-defined extended signed integer types. The standard and extended signed integer types are collectively called signed integer types. Plain ints have the natural size suggested by the architecture of the execution environment; the other signed integer types are provided to meet special needs.

For each of the standard signed integer types, there exists a corresponding (but different) standard unsigned integer type: “unsigned char”, “unsigned short int”, “unsigned int”, “unsigned long int”, and “unsigned long long int”, each of which occupies the same amount of storage and has the same alignment requirements (3.11) as the corresponding signed integer type; that is, each signed integer type has the same object representation as its corresponding unsigned integer type. Likewise, for each of the extended signed integer types there exists a corresponding

---

39) that is, large enough to contain any value in the range of INT_MIN and INT_MAX, as defined in the header <climits>.

40) See 7.1.6.2 regarding the correspondence between types and the sequences of type-specifiers that designate them.
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
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.\footnote{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.}

5 Type wchar_t is a distinct type whose values can represent distinct codes for all members of the largest extended
character set specified among the supported locales (22.1.1). Type wchar_t shall have the same size, signedness,
and alignment requirements (3.11) as one of the other integral types, called its underlying type. Types char16_t
and char32_t denote distinct types with the same size, signedness, and alignment as uint_least16_t and uint_least32_t, respectively, in <stdint.h>, called the underlying types.

6 Values of type bool are either true or false.\footnote{Using a 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 true nor false.} [Note: there are no signed, unsigned, short, or long bool types or values. — end note] Values of type bool participate in integral promotions (4.5).

7 Types bool, char, char16_t, char32_t, wchar_t, and the signed and unsigned integer types are collectively called
integral types.\footnote{Therefore, enumerations (7.2) are not integral; however, enumerations can be promoted to integral types as specified in 4.5.} A synonym for integral type is integer type. The representations of integral types shall define values by use of a pure binary numeration system.\footnote{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.)} [Example: this International Standard permits 2’s complement, 1’s
complement and signed magnitude representations for integral types. — end example]

8 There are three floating point types: float, double, and long double. The type double provides at least as much
precision as float, and the type long double provides at least as much precision as double. The set of values of the
type float is a subset of the set of values of the type double; the set of values of the type double is a subset of the set
of values of the type long double. The value representation of floating-point types is implementation-defined. Integral
and floating types are collectively called arithmetic types. Specializations of the standard template std::numeric_limits (18.2) shall specify the maximum and minimum values of each arithmetic type for an implementation.

9 The void type has an empty set of values. The void type is an incomplete type that cannot be completed. It is used as the
return type for functions that do not return a value. Any expression can be explicitly converted to type cv void (5.4). An
expression of type void shall be used only as an expression statement (6.2), as an operand of a comma expression (5.18),
as a second or third operand of ?: (5.16), as the operand of typeid, or as the expression in a return statement (6.6.3)
for a function with the return type void.

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

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

3.9.2 Compound types

Compound types can be constructed in the following ways:

— arrays of objects of a given type, 8.3.4;
— functions, which have parameters of given types and return void or references or objects of a given type, 8.3.5;
— 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 45) 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 ]

4 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*.

45) Static class members are objects or functions, and pointers to them are ordinary pointers to objects or functions.

Draft
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.\(^{47}\)

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

\(^{46}\) The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

\(^{47}\) 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.

---

3.9.3 CV-qualifiers [basic.type.qualifier]

1 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 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).\(^{46}\)

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.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
no cv-qualifier & const \\
\hline
no cv-qualifier & volatile \\
\hline
no cv-qualifier & const volatile \\
\hline
\hline
const & const volatile \\
\hline
volatile & const volatile \\
\hline
\end{tabular}
\caption{relations on const and volatile}
\end{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.
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 each built-in operator in clause 5 indicates whether it expects lvalue operands and whether it yields an lvalue. — end note ]

5 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.

6 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)).

7 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.

8 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.

9 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.

10 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 ]

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

12 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.

13 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).

14 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.

15 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 undefined48)

— 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,

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

Draft
3.11 Alignment [basic.align]

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. 49)

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 (20.6.13) 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]

49) 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). Certain language constructs require that an expression be converted to a Boolean value. An expression e appearing in such a context is said to be contextually converted to bool and is well-formed if and only if the declaration bool t(e); is well-formed, for some invented temporary variable t (8.5). The effect of either 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]
4.1 Lvalue-to-rvalue conversion

[ conv.lval ]

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

[ conv.array ]

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.

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 \( \text{char} \)”, “pointer to \( \text{char16_t} \)”, “pointer to \( \text{char32_t} \)”, or “pointer to \( \text{wchar_t} \)” 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

[ conv.func ]

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

[ conv.qual ]

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 “\( cv1 T \)”.

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 “\( cv1 T \)”.

[50] 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.

[51] This conversion never applies to non-static member functions because an lvalue that refers to a non-static member function cannot be obtained.
3 [Note: Function types (including those used in pointer to member function types) are never cv-qualified (8.3.5). — end note]

4 A conversion can add cv-qualifiers at levels other than the first in multi-level pointers, subject to the following rules:52)

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 $\cdots$ $cv_{1,n-1}$ pointer to $cv_{1,n} T$

and

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

where each $cv_{i,j}$ is const, volatile, const volatile, or nothing. The n-tuple of cv-qualifiers after the first in a pointer type, e.g., $cv_{1,1}$, $cv_{1,2}$, $\cdots$, $cv_{1,n}$ 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 were allowed), a program could inadvertently modify a const object (as it is done on line //2). For example,

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

— end note]

5 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 $\cdots$ $cv_{n-1} P_{n-1}$ 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.

6 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 $\cdots$ $cv_{1,n-1} P_{n-1}$ to $cv_{1,n} T$

and

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

7 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.

---

52 These rules ensure that const-safety is preserved by the conversion.
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 value 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 \( b_{\text{min}} \) to \( b_{\text{max}} \) as described in 7.2: int, unsigned int, long int, unsigned long int, long long int, or unsigned long long int. If none of the types in that list can represent all the values of the enumeration, 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

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

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

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

exactly represented in the destination type, the result of the conversion is that exact representation. If the source value
is between two adjacent destination values, the result of the conversion is an implementation-defined choice of either of
those values. Otherwise, the behavior is undefined.

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

4.9 Floating-integral conversions

1 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]

2 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

1 A null pointer constant is an integral constant expression (5.19) rvalue of integer type that evaluates to zero or an rvalue
of type std::nullptr_t. A null pointer constant can be converted to a pointer type; the result is the null pointer value
of that type and is distinguishable from every other value of 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).

2 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.

3 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

1 A null pointer constant (4.10) can be converted to a pointer to member type; the result is the null member pointer
value of that type and is distinguishable from any pointer to member not created from a null pointer constant. Two null
pointer member values of the same type shall compare equal. The conversion of a null pointer constant to a pointer to
member of cv-qualified type is a single conversion, and not the sequence of a pointer to member conversion followed
by a qualification conversion (4.4).

2 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

Draft
4.12 Boolean conversions

with a D object. The result is the same as if the pointer to member of B were dereferenced with the B subobject of D. The null member pointer value is converted to the null member pointer value of the destination type.\(^\text{53}\)

### 4.12 Boolean conversions

[conv.bool]

1. 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

[conv.rank]

1. 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.\(^{54}\) 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 \(\text{bool}\). — 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]

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

\(^{54}\)The precedence of operators is not directly specified, but it can be derived from the syntax.
\begin{verbatim}
A a;
&& ar = a;

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

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

8 In some contexts, \textit{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). \textit{— 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. \textit{— 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:

- If either operand is of type \texttt{long double}, the other shall be converted to \texttt{long double}.
- Otherwise, if either operand is \texttt{double}, the other shall be converted to \texttt{double}.
- Otherwise, if either operand is \texttt{float}, the other shall be converted to \texttt{float}.
- Otherwise, the integral promotions (4.5) shall be performed on both operands. \textit{55)} Then the following rules shall be applied to the promoted operands:
  - If both operands have the same type, no further conversion is needed.
  - Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank shall be converted to the type of the operand with greater rank.
  - Otherwise, if the operand that has unsigned integer type has rank greater than or equal to the rank of the type of the other operand, the operand with signed integer type shall be converted to the type of the operand with unsigned integer type.
  - Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type shall be converted to the type of the operand with signed integer type.
  - Otherwise, both operands shall be converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

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. \textit{56)}

\textit{55) 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.}

\textit{56) The cast and assignment operators must still perform their specific conversions as described in 5.4, 5.2.9 and 5.17.}
5.1 Primary expressions

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

- primary-expression:
  - literal
  - this
  - \( \langle \text{expression} \rangle \)
  - id-expression
  - lambda-expression

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

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

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

2 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). 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.

3 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]

4 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.

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

6 An identifier is an id-expression provided it has been suitably declared (clause 7). [Note: for operator-function-ids, see 13.5; for conversion-function-ids, see 12.3.2; for template-ids, see 14.2. A class-name prefixed by \( \sim \) 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 84

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 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-names refer to the same class, this notation names the constructor (12.1). Where class-name :: ~ class-name is used, the two class-names 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:

```c
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]

Draft
5.1.1 Lambda expressions

lambda-expression:
\[ \lambda \text{-introducer} \ \lambda\text{-parameter-declaration}_{\text{opt}} \ \text{compound-statement} \]

lambda-introducer:
\[ [ \lambda\text{-capture}_{\text{opt}} ] \]

lambda-capture:
\[ \text{capture-default} \\
\text{capture-list} \]

capture-default:
\[ \& \]  \=  

capture-list:
\[ \text{capture} \\
\text{capture-list} , \text{capture} \]

capture:
\[ \text{identifier} \\
\& \text{identifier} \]
\[ \text{this} \]

lambda-parameter-declaration:
\[ ( \lambda\text{-parameter-declaration-list}_{\text{opt}} ) \ \text{exception-specification}_{\text{opt}} \ \lambda\text{-return-type-clause}_{\text{opt}} \]

lambda-parameter-declaration-list:
\[ \lambda\text{-parameter} \\
\lambda\text{-parameter} , \lambda\text{-parameter-declaration-list} \]

lambda-parameter:
\[ \text{declspecifier-seq declarator} \]

lambda-return-type-clause:
\[ \rightarrow \text{type-id} \]

1 The evaluation of a lambda-expression results in a closure object, which is an rvalue. Invoking the closure object executes the statements specified in the lambda-expression’s compound-statement. Each lambda expression has a unique type. Except as specified below, the type of the closure object is unspecified. [Note: A closure object behaves as a function object (20.5) whose function call operator, constructors, and data members are defined by the lambda-expression and its context. — end note]

2 A name in the lambda-capture shall be in scope in the context of the lambda expression, and shall be this or shall refer to a local variable or reference with automatic storage duration. [Note: A member of an anonymous union is not a variable. — end note] The same name shall not appear more than once in a lambda-capture. In a lambda-introducer of the form [ capture-default \ , capture-list ], if the capture-default is \&, the capture-list shall not contain a capture having the prefix \&, otherwise each capture in the capture-list other than this shall have the prefix \&.

3 An effective capture set is defined as follows:
   — For a lambda-introducer of the form [ ], the effective capture set is empty.
For a lambda-introducer of the form \([\textit{capture-list}]\), the effective capture set consists of the captures in the capture-list.

For a lambda-introducer of the form \([\textit{capture-default}]\) or \([\textit{capture-default}, \textit{capture-list}]\), the effective capture set consists of

1. the captures in the capture-list, if any, and
2. for each name \(v\) that appears in the lambda expression and denotes a local variable or reference with automatic storage duration in the context where the lambda expression appears and that does not appear in the capture-list or as a parameter name in the lambda-parameter-declaration-list, \&\(v\) if the capture-default is \&\(k\) and \(v\) otherwise, and
3. this if the lambda expression contains a member access expression referring to this (implicitly or explicitly).

The compound-statement of a lambda expression shall use (3.2) an automatic variable or reference from the context where the lambda expression appears only if the name of the variable or reference is a member of the effective capture set, and shall reference this (implicitly or explicitly) only if this is a member of the effective capture set. The compound-statement of a lambda expression shall not refer to a member of an anonymous union with automatic storage duration.

A lambda-expression defines a function and the compound-statement of a lambda-expression has an associated function scope (3.3).

The type of the closure object is a class with a unique name, call it \(F\), considered to be defined at the point where the lambda expression occurs.

Each name \(N\) in the effective capture set is looked up in the context where the lambda expression appears to determine its object type; in the case of a reference, the object type is the type to which the reference refers. For each element in the effective capture set, \(F\) has a private non-static data member as follows:

1. if the element is this, the data member has some unique name, call it \(t\), and is of the type of this (9.3.2);
2. if the element is of the form \&\(N\), the data member has the name \(N\) and type “reference to object type of \(N\)”;
3. otherwise, the element is of the form \(N\), and the data member has the name \(N\) and type “cv-unqualified object type of \(N\)”.

The declaration order of the data members is unspecified.

\(F\) has a public const function call operator (13.5.4) with the following properties:

1. The parameter-declaration-clause is the lambda-parameter-declaration-list.
2. The return type is the type denoted by the type-id in the lambda-return-type-clause; for a lambda expression that does not contain a lambda-return-type-clause the return type is void, unless the compound-statement is of the form \{ return expression; \}, in which case the return type is the type of expression.
3. The exception-specification is the lambda expression’s exception-specification, if any.
4. The compound-statement is obtained from the lambda expression’s compound-statement as follows: If the lambda expression is within a non-static member function of some class \(X\), transform id-expressions to class member
Expressions

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 )

  const_cast < type-id > ( expression )

  typeid ( expression )

  typeid ( type-id )

eexpression-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]
5.2 Postfix expressions

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. 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 function (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 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 to 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]

[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]

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.

When a function is called, each parameter (8.3.5) shall be initialized (8.5, 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.

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

58) A static member function (9.4) is an ordinary function.
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, constexpr cast is required to be used to cast away the constness in order to modify the argument’s value. Where a parameter is of constexpr 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-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.

8 [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 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(x1, x2, ...) is equivalent in effect to the declaration T t(x1, x2, ...); 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

The use of a pseudo-destructor-name after a dot or arrow \( ightarrow \) 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.

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

```haskell
::opt nested-name-specifieropt type-name :: ~ type-name
```

shall designate the same scalar type.

### 5.2.5 Class member access

A postfix expression followed by a dot or an arrow \( \rightarrow \), optionally followed by the keyword template (14.8.1), and then followed by an id-expression, is a postfix expression. The postfix expression before the dot or arrow is evaluated; \(^{59}\) 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 \( \rightarrow \) operators. — end note]

If \( E1 \) has the type “pointer to class \( X \),” then the expression \( E1\rightarrow E2 \) is converted to the equivalent form \( *(E1) \). \( E2 \); the remainder of 5.2.5 will address only the first option (dot) \(^{60}\). Abbreviating object-expression.id-expression as \( E1.E2 \), the type and lvalue properties of this expression are determined as follows. In the remainder of 5.2.5, \( cq \) represents either \( const \) or the absence of \( const \); \( vq \) represents either \( volatile \) or the absence of \( volatile \). \( cv \) 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

---

\(^{59}\) 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.

\(^{60}\) Note that if \( E1 \) has the type “pointer to class \( X \),” then \( *(E1) \) is an lvalue.
type of \( E_1.E_2 \) is “\( vq12 T \)”. If \( E_2 \) is not declared to be a \texttt{mutable} member, then the type of \( E_1.E_2 \) is “\( cq12 vq12 T \)”. 

— If \( E_2 \) is a (possibly overloaded) member function, function overload resolution (13.3) is used to determine whether \( E_1.E_2 \) refers to a static or a non-static member function.

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

— Otherwise, if \( E_1.E_2 \) refers to a non-static member function, and the type of \( E_2 \) is “function of parameter-type-list \( cv \) returning \( T \)”, then \( E_1.E_2 \) 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). \[\text{Note: any redundant set of parentheses surrounding the expression is ignored (5.1). — end note}\] The type of \( E_1.E_2 \) is “function of parameter-type-list \( cv \) returning \( T \)”. 

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

— If \( E_2 \) is a member enumerator, and the type of \( E_2 \) is \( T \), the expression \( E_1.E_2 \) is not an lvalue. The type of \( E_1.E_2 \) is \( T \).

5 If \( E_2 \) is a non-static data member or a non-static member function, the program is ill-formed if the class of which \( E_2 \) is directly a member is an ambiguous base (10.2) of the naming class (11.2) of \( E_2 \).

5.2.6 Increment and decrement \[\text{[expr.post.incr]}\] 

The value of a postfix ++ expression is the value of its operand. \[\text{Note: the value obtained is a copy of the original value — end note}\] The operand shall be a modifiable lvalue. The type of the operand shall be an arithmetic type or a pointer to a complete object type. The value of the operand object is modified by adding 1 to it, unless the object is of type \texttt{bool}, in which case it is set to \texttt{true}. \[\text{Note: this use is deprecated, see Annex D. — end note}\] 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. \[\text{Note: Therefore, a function call shall not intervene between the lvalue-to-rvalue conversion and the side effect associated with any single postfix ++ operator. — end note}\] 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 \texttt{bool}. \[\text{Note: For prefix increment and decrement, see 5.3.2. — end note}\]

5.2.7 Dynamic cast \[\text{[expr.dynamic.cast]}\] 

The result of the expression \texttt{dynamic_cast\lt\langle T\rangle(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 \texttt{dynamic_cast}. The \texttt{dynamic_cast} operator shall not cast away constness (5.2.11).

2 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 \( T \).
3 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).

4 If the value of \( v \) is a null pointer value in the pointer case, the result is the null pointer value of type \( R \).

5 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 \( v \). 61) 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 { }; 
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 \texttt{std::bad_cast} (18.6.2).

[ Example:

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

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

62) The recommended name for such a class is extended_type_info.
63) If p is an expression of pointer type, then *p, (**p), *(*(p)), *((**p)), ***((p)), and so on all meet this requirement.
5.2 Postfix expressions

Expressions 94

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.

[ Note: 12.7 describes the behavior of typeid applied to an object under construction or destruction. — end note ]

5.2.9 Static cast [expr.static.cast]

1 The result of the expression static_cast<T>(v) is the result of converting the expression v to type T. If T is an lvalue reference type, the result is an lvalue; otherwise, the result is an rvalue. Types shall not be defined in a static_cast. The static_cast operator shall not cast away constness (5.2.11).

2 An lvalue of type “cv1 B,” where B is a class type, can be cast to type “reference to cv2 D,” where D is a class derived (clause 10) from B, if a valid standard conversion from “pointer to D” to “pointer to B” exists (4.10), cv2 is the same cv-qualification as, or greater cv-qualification than, cv1, and B is neither a virtual base class of D nor a base class of a virtual base class of D. The result has type “cv2 D.” It is an lvalue if the type cast to is an lvalue reference; otherwise, it is an rvalue. An rvalue of type “cv1 B” may be cast to type “rvalue reference to cv2 D” with the same constraints as for an lvalue of type “cv1 B.” The result is an rvalue. If the object of type “cv1 B” is actually a subobject of an object of type D, the result refers to the enclosing object of type D. Otherwise, the result of the cast is undefined. [ Example:

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

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

— end example ]

3 Otherwise, an expression e can be explicitly converted to a type T using a static_cast of the form static_cast<T>(e) if the declaration T t(e); is well-formed, for some invented temporary variable t (8.5). The effect of such an explicit conversion is the same as performing the declaration and initialization and then using the temporary variable as the result of the conversion. The 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 Otherwise, the static_cast shall perform one of the conversions listed below. No other conversion shall be performed explicitly using a static_cast.

5 Any expression can be explicitly converted to type cv 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 lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are not applied to the expression.

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

```cpp
struct B { }; 
struct D : private B { }; 
void f() {
    static_cast<D*>(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 cv l 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, cv l, 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 cv l 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 cv l 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, cv l.64) 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 cv l 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, cv l. 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:

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

[64]Function types (including those used in pointer to member function types) are never cv-qualified; see 8.3.5.
5.2 Postfix expressions

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]

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 of integral type is not necessarily converted to a null pointer value. (A null pointer constant of type `std::nullptr_t` cannot appear as the operand of `reinterpret_cast`, nor can any value be converted by `reinterpret_cast` to type `std::nullptr_t`.) — 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

---

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

5.2.11 Const cast [expr.const.cast]

1 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.

2 [ 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.

\[^{67}\text{This is sometimes referred to as a type pun.}\]
5.3 Unary expressions

Expressions with unary operators group right-to-left.

---

$^{(68)}$ `const_cast` is not limited to conversions that cast away a const-qualifier.
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

1 The unary * operator performs indirection: the expression to which it is applied shall be a pointer to an object type, or a pointer to a function type and the result is an lvalue referring to the object or function to which the expression points. If the type of the expression is “pointer to T,” the type of the result is “T.” [Note: 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]

2 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]

3 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]

4 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.

5 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]

Draft
5.3 Unary expressions

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 contextually converted to \texttt{bool} (clause 4); its value is \texttt{true} if the converted operand is \texttt{false} and \texttt{false} otherwise. The type of the result is \texttt{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 \textit{unary-expression} \(~X\), where \(X\) is a \textit{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

The operand of prefix \(++\) is modified by adding 1, or set to \texttt{true} if it is \texttt{bool} (this use is deprecated). The operand shall be a modifiable lvalue. The type of the operand shall be an arithmetic type or a pointer to a completely-defined object type. The result is the updated operand; it is an lvalue, and it is a bit-field if the operand is a bit-field. If \(x\) is not of type \texttt{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]

The operand of prefix \(--\) is modified by subtracting 1. The operand shall not be of type \texttt{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

The \texttt{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 \textit{type-id}. The \texttt{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. \texttt{sizeof(char)}, \texttt{sizeof(signed char)} and \texttt{sizeof(unsigned char)} are 1. The result of \texttt{sizeof} applied to any other fundamental type (3.9.1) is implementation-defined. [Note: in particular, \texttt{sizeof(bool)}, \texttt{sizeof(char16_t)}, \texttt{sizeof(char32_t)}, and \texttt{sizeof(wchar_t)} are implementation-defined. — end note] [Note: See 1.7 for the definition of \textit{byte} and 3.9 for the definition of \textit{object representation}. — end note]

When applied to a reference or a reference type, the result is the size of the referenced type. When applied to a class, the result is the number of bytes in an object of that class including any padding required for placing objects of that type in an array. The size of a most derived class shall be greater than zero (1.8). The result of applying \texttt{sizeof} to a base class subobject is the size of the base class type.\(^{70}\) When applied to an array, the result is the total number of bytes in the array. This implies that the size of an array of \(n\) elements is \(n\) times the size of an element.

The \texttt{sizeof} operator can be applied to a pointer to a function, but shall not be applied directly to a function.

\(^{69}\)\texttt{sizeof(bool)} is not required to be 1.

\(^{70}\)The actual size of a base class subobject may be less than the result of applying \texttt{sizeof} to the subobject, due to virtual base classes and less strict padding requirements on base class subobjects.
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.

Types shall not be defined in a sizeof expression.

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 ]

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

5.3.4 New [expr.new]

The new-expression attempts to create an object of the type-id (8.1) or new-type-id to which it is applied. The type of that object is the allocated type. This type shall be a complete object type, but not an abstract class type or array thereof (1.8, 3.9, 10.4). It is implementation-defined whether over-aligned types are supported (3.11). [Note: because references are not objects, references cannot be created by new-expressions. — end note ] [Note: the type-id may be a cv-qualified type, in which case the object created by the new-expression has a cv-qualified type. — end note ]

new-expression:
```
::opt new new-placement opt new-type-id new-initializer opt
::opt new new-placement opt (type-id) new-initializer opt
```

class new-placement:
```
(expression-list)
```

class new-type-id:
```
type-specifier-seq new-declarator_opt
```

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

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

class 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.

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
5.3 Unary expressions

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-declarators. [Note: this prevents ambiguities between declarator operators & *, [], and their expression counterparts. — end note] [Example:
  new int * i;  // syntax error: parsed as (new int*) i, not as (new int)*i
  The * is the pointer declarator and not the multiplication operator. — end example]

4 [Note: parentheses in a new-type-id of a new-expression can have surprising effects. [Example:
  new int(*[10])();  // error
  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):
  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 non-explicit 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:
  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.
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 (17.4.3.5) and/or class-specific versions (12.5). — end note]

If the new-expression begins with a unary :: operator, the allocation function’s name is looked up in the global scope. Otherwise, if the allocated type is a class type T or array thereof, the allocation function’s name is looked up in the scope of T. If this lookup fails to find the name, or if the allocated type is not a class type, the allocation function’s name is looked up in the global scope.

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 strictest fundamental alignment requirement (3.11) of any object type whose size is no greater than the size of the array being created. [Note: Because allocation functions are assumed to return pointers to storage that is appropriately aligned for objects of any type with fundamental alignment, this constraint on array allocation overhead permits the common idiom of allocating character arrays into which objects of other types will later be placed. — end note]

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-expressions, including those referencing the library function operator new[](std::size_t, void*) and other placement allocation functions. The amount of overhead may vary from one invocation of new to another. — end example]

[Note: unless an allocation function 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

16 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-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-provided default constructor.

      — Otherwise, the object created has indeterminate value. If `T` is a const-qualified type, or a (possibly cv-qualified) 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);

      — 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\(^{71}\) 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:

\(^{71}\)This may include evaluating a `new-initializer` and/or calling a constructor.
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.

22 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 [expr.delete]

1 The delete-expression operator destroys a most derived object (1.8) or array created by a new-expression.

   delete-expression:
       ::opt delete cast-expression
       ::opt delete [ ] cast-expression

The first alternative is for non-array objects, and the second is for arrays. Whenever the delete keyword is immediately followed by empty square brackets, it shall be interpreted as the second alternative. The operand shall have a pointer type, or a class type having a single non-explicit conversion function (12.3.2) to a pointer type. The result has type void.

2 If the operand has a class type, the operand is converted to a pointer type by calling the above-mentioned conversion function, and the converted operand is used in place of the original operand for the remainder of this 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. 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

72) A lambda expression with a lambda-introducer that consists of empty square brackets can follow the delete keyword if the lambda expression is enclosed in parentheses.

73) 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.
Explicit type conversion (cast notation)

Expressions 106

5.4 Explicit type conversion (cast notation)

An `alignof` expression yields the alignment requirement of its operand type. The operand shall be a `type-id` representing a complete object type.

The result is an integral constant of type `std::size_t`.

When `alignof` is applied to a reference type, the result shall be the alignment of the referenced type. When `alignof` is applied to an array type, the result shall be the alignment of the element type.

A type shall not be defined in an `alignof` expression.

5.4 Explicit type conversion (cast notation)

The result of the expression `(T)` `cast-expression` is of type `T`. The result is an lvalue if `T` is an lvalue reference type, otherwise the result is an rvalue. [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; see 3.10. — end note]

An explicit type conversion can be expressed using functional notation (5.2.3), a type conversion operator (`dynamic_cast`, `static_cast`, `reinterpret_cast`, `const_cast`), or the `cast` notation.

`cast-expression:
  unary-expression
  ( type-id ) cast-expression`

---

74) This implies that an object cannot be deleted using a pointer of type `void*` because there are no objects of type `void`.

Draft
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

- a `const_cast` (5.2.11),
- a `static_cast` (5.2.9),
- a `static_cast` followed by a `const_cast`,
- a `reinterpret_cast` (5.2.10), or
- a `reinterpret_cast` followed by a `const_cast`,

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

- 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;
- 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;
- 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:

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

— end example ]

The operand of a cast using the cast notation can be 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

The pointer-to-member operators 

\( \texttt{-} * \) and \( . * \) group left-to-right.

\[ \begin{align*}
\text{pm-expression:} \\
\text{cast-expression} \\
\text{pm-expression} . * \text{cast-expression} \\
\text{pm-expression} \rightarrow* \text{cast-expression}
\end{align*} \]

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.

The binary operator \( \rightarrow* \) 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.

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

The restrictions on \( cv \)-qualification, and the manner in which the \( cv \)-qualifiers of the operands are combined to produce the \( cv \)-qualifiers of the result, are the same as the rules for \( E1.E2 \) given in 5.2.5. [Note: it is not possible to use a pointer to member that refers to a \( mutable \) member to modify a \( const \) class object. For example,

```c
struct S {
    S() : i(0) { }
        mutable int i;
};
void f()
{
    const S cs;
    int S::* pm = &S::i; // pm refers to mutable member S::i
    cs.*pm = 88; // ill-formed: cs is a const object
}
```

— end note ]

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

[Example:

\( (\texttt{ptr_to_obj->*ptr_to_mfct})(10); \)

calls the member function denoted by \( \texttt{ptr_to_mfct} \) for the object pointed to by \( \texttt{ptr_to_obj} \). — end example ]

The result of a \( . * \) expression is an lvalue only if its first operand is an lvalue and its second operand is a pointer to data member. The result of an \( \rightarrow* \) expression is an lvalue only if its second operand is a pointer to data member. If the second operand is the null pointer to member value (4.11), the behavior is undefined.

5.6 Multiplicative operators

The multiplicative operators \( \ast \), \( / \), and \( \% \) group left-to-right.
2 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.

3 The binary * operator indicates multiplication.

4 The binary / operator yields the quotient, and the binary % operator yields the remainder from the division of the first expression by the second. If the second operand of / or % is zero the behavior is undefined; otherwise \((a/b) \times b + a \% b\) 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

1 The additive operators + and - group left-to-right. The usual arithmetic conversions are performed for operands of arithmetic or enumeration type.

```
additive-expression:
  multiplicative-expression
  additive-expression + multiplicative-expression
  additive-expression - multiplicative-expression
```

For addition, either both operands shall have arithmetic or 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

\(^{75}\) 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.
5.8 Shift operators Expressions

The shift operators \( \ll \) and \( \gg \) group left-to-right.

\[
\text{shift-expression:} \\
\quad \text{additive-expression} \\
\quad \text{shift-expression} \quad \ll \quad \text{additive-expression} \\
\quad \text{shift-expression} \quad \gg \quad \text{additive-expression}
\]

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 \( E1 \ll E2 \) is \( E1 \) (interpreted as a bit pattern) left-shifted \( E2 \) bit positions; vacated bits are zero-filled. If \( E1 \) has an unsigned type, the value of the result is \( E1 \) multiplied by the quantity 2 raised to the power \( E2 \), reduced modulo \( \text{ULLONG\_MAX}+1 \) if \( E1 \) has type unsigned long long int, \( \text{ULONG\_MAX}+1 \) if \( E1 \) has type unsigned long int, \( \text{UINT\_MAX}+1 \) otherwise. [Note: the constants \( \text{ULLONG\_MAX} \), \( \text{ULONG\_MAX} \), and \( \text{UINT\_MAX} \) are defined in the header \text{<climits>}. — end note]

The value of \( E1 \gg E2 \) is \( E1 \) right-shifted \( E2 \) bit positions. If \( E1 \) has an unsigned type or if \( E1 \) has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of \( E1 \) divided by the quantity 2 raised to the

---

Another way to approach pointer arithmetic is first to convert the pointer(s) to character pointer(s): In this scheme the integral value of the expression added to or subtracted from the converted pointer is first multiplied by the size of the object originally pointed to, and the resulting pointer is converted back to the original type. For pointer subtraction, the result of the difference between the character pointers is similarly divided by the size of the object originally pointed to.

When viewed in this way, an implementation need only provide one extra byte (which might overlap another object in the program) just after the end of the object in order to satisfy the “one past the last element” requirements.
power E2. If E1 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]

```latex
relational-expression:
  shift-expression
  relational-expression < shift-expression
  relational-expression > shift-expression
  relational-expression <= shift-expression
  relational-expression >= shift-expression
```

The operands shall have arithmetic, enumeration, or pointer type, or type `std::nullptr_t`. The operators `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to) all yield `false` or `true`. The type of the result is `bool`.

The usual arithmetic conversions 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 **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 cv12 void,” where `cv12` 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 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* before comparison
pi <= pci; // Both converted to const int *const * before comparison
}
```

— 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 \( p \leq q \) and \( p \geq q \) 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 \( p < q \), \( p > q \), \( p \leq q \), and \( p \geq q \) 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 have the same access control (clause 11) and provided their class is not a union.
5.10 Equality operators

--- If two pointers point to non-static data members of the same object 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.

If two operands of type std::nullptr_t are compared, the result is true if the operator is <= or >=, and false otherwise.

5.10 Equality operators

equality-expression:

relational-expression
equality-expression == relational-expression
equality-expression != relational-expression

1 The == (equal to) and the != (not equal to) operators 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).

2 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:

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

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

— end example ]

Draft
If two operands of type `std::nullptr_t` are compared, the result is `true` if the operator is `==`, and `false` otherwise.

### 5.11 Bitwise AND operator

```latex
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

```latex
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

```latex
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

```latex
logical-and-expression:
  inclusive-or-expression
logical-and-expression && inclusive-or-expression
```

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

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

```latex
logical-or-expression:
  logical-and-expression
logical-or-expression || logical-and-expression
```

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

2 The result is a `bool`. 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

**conditional-expression:**

- **logical-or-expression**
- **logical-or-expression** ? **expression** : **assignment-expression**

1. Conditional expressions group right-to-left. The first expression is contextually converted to `bool` (clause 4). It is evaluated and if it is `true`, the result of the conditional expression is the value of the second expression, otherwise that of the third expression. Only one of the second and third expressions is evaluated. Every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second or third expression.

2. If either the second or the third operand has type (possibly cv-qualified) `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 `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 `void`; the result is of type `void` and is an rvalue. [Note: this includes the case where both operands are `throw-expressions`. — end note]

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

1 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
      \(=\), \(/=\), \(\%\), \(<\), \(<=\), \(\ge\), \(>=\), \(\&\), \(\&\&\), \(|\)

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

3 If the left operand is not of class type, the expression is implicitly converted (clause 4) to the cv-unqualified type of the left operand.
5.18 Comma operator

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 \( E_1 \ op \ = \ E_2 \) is equivalent to \( E_1 \ = \ E_1 \ op \ E_2 \) except that \( E_1 \) is evaluated only once. In \(+\) and \(\)\(-\), \( E_1 \) shall either have arithmetic type or be a pointer to a possibly cv-qualified completely defined object type. In all other cases, \( E_1 \) shall have arithmetic type.

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) — end example] 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

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]

constant-expression:

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]:

— \( \text{this} \) (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];
— a lambda-expression (5.1.1);
— 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;
— 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).

3 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 initializ-
  ers (7.2), as static member initializers (9.4.2), and as integral or enumeration non-type template arguments (14.3). — end
  note]

4 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 non-explicit conversion function to an integral or enumeration type and that conversion function
  shall be constexpr. [Example:

```cpp
struct A {
    constexpr A(int i) : val(i) { }
    constexpr operator int() { return val; }
    constexpr operator long() { return 43; }
private:
    int val;
```
Draft
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

   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

Selection statements choose one of several flows of control.

```
int x;       // ill-formed, redeclaration of x
}

/* end example */

4 The value of a condition that is an initialized declaration in a statement other than a switch statement is the value of the declared variable contextually converted to bool (clause 4). If that conversion is ill-formed, the program is ill-formed. The value of a condition that is an initialized declaration in a switch statement is the value of the declared variable if it has integral or enumeration type, or of that variable implicitly converted to integral or enumeration type otherwise. The value of a condition that is an expression is the value of the expression, contextually converted to bool for statements other than switch; if that conversion is ill-formed, the program is ill-formed. The value of the condition will be referred to as simply “the condition” where the usage is unambiguous.

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

6.4.1 The if statement

1 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.??

6.4.2 The switch statement

1 The switch statement causes control to be transferred to one of several statements depending on the value of a condition.

2 The condition shall be of integral type, enumeration type, or of a class type for which a single non-explicit 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:

```
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.

3 There shall be at most one label of the form

```
default :
```
within a switch statement.

4 Switch statements can be nested; a case or default label is associated with the smallest switch enclosing it.

5 When the switch statement is executed, its condition is evaluated and compared with each case constant. If one of the case constants is equal to the value of the condition, control is passed to the statement following the matched case label. If no case constant matches the condition, and if there is a default label, control passes to the statement labeled by the default label. If no case matches and if there is no default then none of the statements in the switch is executed.

??In other words, the else is associated with the nearest un-elled if.
6.5 Iteration statements

1 Iteration statements specify looping.

\begin{verbatim}
iteration-statement:
  while ( condition ) statement
do statement while ( expression ) ;
for ( for-init-statement condition\_opt ; expression\_opt ) statement

for-init-statement:
exression-statement
  simple-declaration
\end{verbatim}

[ 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 ]

3 [ Note: The requirements on conditions in iteration statements are described in 6.4. — end note ]

4 A loop that, outside of the for-init-statement in the case of a for statement,  
  — performs no I/O operations, and  
  — does not access or modify volatile objects, and  
  — performs no synchronization or atomic operations  
may be assumed by the implementation to terminate. [ Note: This is intended to allow compiler transformations, such as removal of empty loops, even when termination cannot be proven. — 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.

Draft
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

```c
while (T t = x) statement
```

is equivalent to

```c
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:]

```c
struct A {
    int val;
    A(int i) : val(i) { }
    ~A() { }
    operator bool() { return val != 0; }
};
int i = 1;
while (A a = i) {
    // ...
    i = 0;
}
```

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

### 6.5.2 The do statement

1 The expression is contextually converted to `bool` (clause 4); if that conversion is ill-formed, the program is ill-formed.

2 In the do statement the substatement is executed repeatedly until the value of the expression becomes `false`. The test takes place after each execution of the statement.

### 6.5.3 The for statement

1 The `for` statement

```c
for (for-init-statement condition_opt ; expression_opt ) statement
```

is equivalent to
6.6 Jump statements

{  
    for-init-statement
    while ( condition ) {  
        statement
        expression ;
    }
}

except that names declared in the for-init-statement are in the same declarative-region as those declared in the condition, and except that a continue in statement (not enclosed in another iteration statement) will execute expression before re-evaluating condition. [ Note: Thus the first statement specifies initialization for the loop; the condition (6.4) specifies a test, made before each iteration, such that the loop is exited when the condition becomes false; the expression often specifies incrementing that is done after each iteration. — end note ]

2 Either or both of the condition and the expression can be omitted. A missing condition makes the implied while clause equivalent to while(true).

3 If the for-init-statement is a declaration, the scope of the name(s) declared extends to the end of the for-statement.  
   [ Example:
   int i = 42;
   int a[10];
   
   for (int i = 0; i < 10; i++)
       a[i] = i;
   int j = i;  // j = 42
   — end example ]

6.6 Jump statements

1 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
6.6.2 The continue statement

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

```
while (foo) {
    // ...
    contin: ;
} while (foo);
```

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

6.6.3 The return statement

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

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.

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

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

6.7 Declaration statement

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

```
declaration-statement:
    block-declaration
```

If an identifier introduced by a declaration was previously declared in an outer block, the outer declaration is hidden for the remainder of the block, after which it resumes its force.

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

It is possible to transfer into a block, but not in a way that bypasses declarations with initialization. A program that
6.8 Ambiguity resolution

void f() {
    // ...
    goto lx;          // ill-formed: jump into scope of a
    // ...
    ly:               // OK, jump implies destructor
        X a = 1;    // call for a followed by construction
        // ...
    lx:               // again immediately following label ly
        goto ly;
}

— end example ]

4 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 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;
    }

— end example ]

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 ]

6.8 Ambiguity resolution

There is an ambiguity in the grammar involving expression-statements and declarations: An expression-statement with a function-style explicit type conversion (5.2.3) as its leftmost subexpression can be indistinguishable from a declaration where the first declarator starts with a ( . In those cases the statement is a declaration. [ Note: 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

78) The transfer from the condition of a switch statement to a case label is considered a jump in this respect.
T(*d)(int);                  // declaration
T(e)[5];                    // declaration
T(f) = { 1, 2 };            // declaration
T(*g)(double(3));           // declaration

In the last example above, g, which is a pointer to T, is initialized to double(3). This is of course ill-formed for semantic reasons, but that does not affect the syntactic analysis. —end example—

The remaining cases are declarations. [Example:

```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); }  // T1(a) = 3,  
    int operator=(int x) { return x; }     // T2 will be declared as
    T1(int) { }                           // a variable of type T1
};
struct T2 { T2(int){ } };            // but this will not allow
int a, (*((b)(T2))(int,d));         // the last part of the
void f() {                           // declaration to parse
    // disambiguation requires this to be parsed as a declaration:
    T1(a) = 3,                        // properly since it depends
    T2(4),                            // T2 will be declared as
    (*((b)(T2(c)))(int,d));           // a variable of type T1
};
```

Draft
// on T2 being a type-name
}

— end example ]
Chapter 7  Declarations

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

\[
\text{declaration-seq:}
\text{\hspace{1em} declaration}
\text{\hspace{1em} declaration-seq declaration}
\]

\[
\text{declaration:}
\text{\hspace{1em} block-declaration}
\text{\hspace{1em} function-definition}
\text{\hspace{1em} template-declaration}
\text{\hspace{1em} explicit-instantiation}
\text{\hspace{1em} explicit-specialization}
\text{\hspace{1em} linkage-specification}
\text{\hspace{1em} namespace-definition}
\]

\[
\text{block-declaration:}
\text{\hspace{1em} simple-declaration}
\text{\hspace{1em} asm-definition}
\text{\hspace{1em} namespace-alias-definition}
\text{\hspace{1em} using-declaration}
\text{\hspace{1em} using-directive}
\text{\hspace{1em} static_assert-declaration}
\text{\hspace{1em} alias-declaration}
\]

\[
\text{alias-declaration:}
\text{\hspace{1em} using \, identifier = type-id}
\]

\[
\text{simple-declaration:}
\text{\hspace{1em} decl-specifier-seq_opt init-declarator-list_opt ;}
\]

\[
\text{static_assert-declaration:}
\text{\hspace{1em} static_assert ( constant-expression , string.literal ) ;}
\]

[ Note: asm-definitions are described in 7.4, and linkage-specifications are described in 7.5. Function-definitions are described in 8.4 and template-declarations are described in clause 14. Namespace-definitions are described in 7.3.1, using-declarations are described in 7.3.3 and using-directives are described in 7.3.4. — end note ]

The simple-declaration

\[
\text{decl-specifier-seq_opt init-declarator-list_opt ;}
\]

is divided into two parts: decl-specifiers, the components of a decl-specifier-seq, are described in 7.1 and declarators, the components of an init-declarator-list, are described in clause 8.

2  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
7.1 Specifiers

The specifiers that can be used in a declaration are

79) The “implicit int” rule of C is no longer supported.

Draft
The longest sequence of `decl-specifiers` 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 g(const int Pc);
```

— end example ]

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

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

— end example ] — end note ]

### 7.1.1 Storage class specifiers

The storage class specifiers are

```c
storage-class-specifier:
  register
  static
  extern
  mutable
```

At most one `storage-class-specifier` shall appear in a given `decl-specifier-seq`. If a `storage-class-specifier` appears in a `decl-specifier-seq`, there can be no `typedef` specifier in the same `decl-specifier-seq` and the `init-declarator-list` of the declaration shall not be empty (except for global anonymous unions, which shall be declared `static` (9.5)). The `storage-class-specifier` applies to the name declared by each `init-declarator` in the list and not to any names declared by other specifiers. A `storage-class-specifier` shall not be specified in an explicit specialization (14.7.3) or an explicit instantiation (14.7.2) directive.

The `register` specifier shall be applied only to names of objects declared in a block (6.3) or to function parameters (8.4). It 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.
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();
inline void h(); // external linkage

inline void l();
void l(); // external linkage

inline void m();
extern void m(); // external linkage

static void n();
inline void n(); // internal linkage

static int a; // a has internal linkage
int a; // error: two definitions

static int b; // b has internal linkage
extern b; // b still has internal linkage

int c; // c has external linkage
static int c; // error: inconsistent linkage
```
extern int d;  // d has external linkage
static int d;  // error: inconsistent linkage

— end example ]

8 The name of a declared but undefined class can be used in an extern declaration. Such a declaration can only be used in ways that do not require a complete class type. [Example:

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

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

— end example ]

9 The mutable specifier 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
};

— end example ]

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

7.1.2 Function specifiers [dcl.fct.spec]

1 Function-specifiers can be used only in function declarations.

    function-specifier:
    inline
    virtual
    explicit

2 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.

3 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 or conversion function within its class definition; see 12.3.1 and 12.3.2.

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;
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 { /* ... */ };
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 ]
```

7 [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
7.1 Specifiers

— end example ]

9 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 [dcl.friend]
The friend specifier is used to specify access to class members; see 11.4.

7.1.5 The constexpr specifier [dcl.constexpr]
1 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 ]

2 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).

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

Draft
— its function-body shall be a compound-statement of the form

\[
\begin{array}{l}
\{ \text{return expression; } \}
\end{array}
\]

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]

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.
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:
    constexpr 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  ]

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

The type-specifiers are

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

type-specifier-seq:
  type-specifier type-specifier-seqopt

As a general rule, at most one type-specifier is allowed in the complete decl-specifier-seq of a declaration or in a type-specifier-seq. The only exceptions to this rule are the following:

— const can be combined with any type specifier except itself.
139 Declarations 7.1 Specifiers

— `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.

3 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.\(^ {81} \)

4 [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

1 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]

2 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]

3 A pointer or reference to a cv-qualified type need not actually point or refer to a cv-qualified object, but it is treated as if it does; a const-qualified access path cannot be used to modify an object even if the object referenced is a non-const object and can be modified through some other access path. [Note: cv-qualifiers are supported by the type system so that they cannot be subverted without casting (5.2.11). — end note]

4 Except that any class member declared `mutable` (7.1.1) can be modified, any attempt to modify a `const` object during its lifetime (3.8) results in undefined behavior. [Example:

```c
const int ci = 3; // cv-qualified (initialized as required)
ci = 4; // ill-formed: attempt to modify const

int i = 2; // not cv-qualified
const int* cip; // pointer to const int
    cip = &i; // OK: cv-qualified access path to unqualified
*cip = 4; // ill-formed: attempt to modify through ptr to const

int* ip;
ip = const_cast<int*>(cip); // cast needed to convert const int* to int*
```

\(^ {81} \)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-qualifiers. The “implicit int” rule of C is no longer supported.

Draft
For another example

```cpp
struct X {
    mutable int i;
    int j;
};
struct Y {
    X x;
    Y();
};
const Y y;
y.x.i++;    // well-formed: mutable member can be modified
y.x.j++;    // ill-formed: const-qualified member modified
Y* p = const_cast<Y*>(&y);  // cast away const-ness of y
p->x.i = 99;    // well-formed: mutable member can be modified
p->x.j = 99;    // undefined: modifies a const member
```

--- end example ---

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. — end note]

### 7.1.6.2 Simple type specifiers

The simple type specifiers are

```cpp
*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
```
**7.1 Specifiers**

simple-type-specifier:
::opt nested-name-specifier opt type-name
::opt nested-name-specifier template simple-template-id
char
cchar16_t
cchar32_t
wchar_t
bool
short
int
long
signed
unsigned
float
double
void
auto
dcltype ( expression )

type-name:
class-name
enum-name
typedef-name

2 The auto specifier is a placeholder for a type to be deduced (7.1.6.4). The other *simple-type-specifiers* 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-specifiers* and the types they specify.

<table>
<thead>
<tr>
<th>Specifier(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>type-name</td>
<td>the type named</td>
</tr>
<tr>
<td>char</td>
<td>&quot;char&quot;</td>
</tr>
<tr>
<td>unsigned char</td>
<td>&quot;unsigned char&quot;</td>
</tr>
<tr>
<td>signed char</td>
<td>&quot;signed char&quot;</td>
</tr>
<tr>
<td>char16_t</td>
<td>&quot;char16_t&quot;</td>
</tr>
<tr>
<td>char32_t</td>
<td>&quot;char32_t&quot;</td>
</tr>
<tr>
<td>bool</td>
<td>&quot;bool&quot;</td>
</tr>
<tr>
<td>unsigned</td>
<td>&quot;unsigned int&quot;</td>
</tr>
<tr>
<td>signed int</td>
<td>&quot;unsigned int&quot;</td>
</tr>
<tr>
<td>signed</td>
<td>&quot;int&quot;</td>
</tr>
<tr>
<td>signed int</td>
<td>&quot;int&quot;</td>
</tr>
<tr>
<td>int</td>
<td>&quot;int&quot;</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>&quot;unsigned short int&quot;</td>
</tr>
<tr>
<td>unsigned short</td>
<td>&quot;unsigned short int&quot;</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>&quot;unsigned long int&quot;</td>
</tr>
<tr>
<td>unsigned long</td>
<td>&quot;unsigned long int&quot;</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>&quot;unsigned long long int&quot;</td>
</tr>
</tbody>
</table>

Draft
### 7.1 Specifiers Declarations

<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>signed long int</td>
<td>“long int”</td>
</tr>
<tr>
<td>signed long</td>
<td>“long int”</td>
</tr>
<tr>
<td>signed long long int</td>
<td>“long long int”</td>
</tr>
<tr>
<td>signed long long</td>
<td>“long long int”</td>
</tr>
<tr>
<td>long long int</td>
<td>“long long int”</td>
</tr>
<tr>
<td>long long int</td>
<td>“long long int”</td>
</tr>
<tr>
<td>long long int</td>
<td>“long long int”</td>
</tr>
<tr>
<td>long long int</td>
<td>“long long int”</td>
</tr>
<tr>
<td>long long int</td>
<td>“long long int”</td>
</tr>
<tr>
<td>long long int</td>
<td>“long long int”</td>
</tr>
<tr>
<td>long</td>
<td>“long int”</td>
</tr>
<tr>
<td>signed short int</td>
<td>“short int”</td>
</tr>
<tr>
<td>signed short int</td>
<td>“short int”</td>
</tr>
<tr>
<td>signed short int</td>
<td>“short int”</td>
</tr>
<tr>
<td>short int</td>
<td>“short int”</td>
</tr>
<tr>
<td>short int</td>
<td>“short int”</td>
</tr>
<tr>
<td>wchar_t</td>
<td>“wchar_t”</td>
</tr>
<tr>
<td>float</td>
<td>“float”</td>
</tr>
<tr>
<td>double</td>
<td>“double”</td>
</tr>
<tr>
<td>long double</td>
<td>“long double”</td>
</tr>
<tr>
<td>void</td>
<td>“void”</td>
</tr>
<tr>
<td>auto</td>
<td>placeholder for a type to be deduced</td>
</tr>
<tr>
<td>decltype(expression)</td>
<td>the type as defined below</td>
</tr>
</tbody>
</table>

3 When multiple simple-type-specifiers are allowed, they can be freely intermixed with other decl-specifiers in any order. [Note: It is implementation-defined whether objects of char type 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]

4 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:

```cpp
class Test {
public:
  void foo();
  int i;
private:
  struct A { double x; };
const A* a = new A();
decltype(foo()); // type is const int&
decltype(i); // type is int
decltype(a->x); // type is double
```
]
decltype((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-specifier opt template opt simple-template-id
  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 template opt 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-form[ed. — end note ]

3 The class-key or 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 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

The 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 or specified explicitly at the end of a function declarator. The name of the object being declared shall not appear in the initializer expression.
The auto type-specifier may appear with a function declarator with a late-specified return type (8.3.5) in any context where such a declarator is valid, and the use of auto is replaced by the type specified at the end of the declarator.

Otherwise, the type of the object is 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:

= assignment-expression
( assignment-expression )

[ Example:
  auto x = 5; // OK: x has type int
  const auto *v = &x, u = 6; // OK: v has type const int*, u has type const int
  static auto y = 0.0; // OK: y has type double
  static auto int z; // error: auto and static conflict
  auto int r; // OK: r has type int error: auto is not a storage-class-specifier

  — 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.

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(expr) of the following invented function template:
  template <class U> void f(const U& u);

  — end example ]

7.1.7 Alignment specifier [dcl.align]

The alignment specifier has the form

alignment-specifier:
  alignas ( constant-expression )
  alignas ( type-id )

Draft
2 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 align-
     ment 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.

3 When the alignment specifier is of the form `alignas(type-id)`, it shall have the same effect as `alignas(alignof(type-
   id))` (5.3.6).

4 When multiple alignment specifiers are specified for an object, the alignment requirement shall be set to the strictest
   specified alignment.

5 The combined effect of all alignment specifiers in a declaration shall not specify an alignment that is less strict than the
   alignment that would otherwise be required for the object being declared.

6 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]

7 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.

8 [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:

   ```
   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
   `alignof(T)`, and therefore the program will not be ill-formed. —end example]

9 [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

1 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

enum-name:
  identifier
enum-specifier:
  enum-key identifier_opt enum-base_opt { enumerator-list_opt }
  enum-key identifier_opt enum-base_opt { 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

The enumeration type declared with an enum-key of only enum is an unscoped enumeration, and its enumerators are unscoped enumerators. The enum-keys enum class and enum struct are semantically equivalent; an enumeration type declared with one of these is a scoped enumeration, and its enumerators are scoped enumerators. The 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:
  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 ]

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 prior to the closing brace 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:

— If an initializer is specified for an enumerator, the initializing value has the same type as the expression.
— If no initializer is specified for the first enumerator, the initializing value has an unspecified integral type.
— Otherwise the type of the initializing value is the same as the type of the initializing value of the preceding enumerator unless the incremented value is not representable in that type, in which case the type is an unspecified integral type sufficient to contain the incremented value.
For an enumeration whose underlying type is not fixed, the underlying type is an integral type that can represent all
the enumerator values defined in the enumeration. If no integral type can represent all the enumerator values, the
enumeration is ill-formed. It is implementation-defined which integral type is used as the underlying type except that
the underlying type shall not be larger than int unless the value of an enumerator cannot fit in an int or unsigned int. If the enumeration-list is empty, the underlying type is as if the enumeration had a single enumerator with value 0.

For an enumeration whose underlying type is fixed, the values of the enumeration are the values of the underlying
type. Otherwise, for an enumeration where \( e_{min} \) 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 promo-
tion (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;  // 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.

Each enum-name and each unscoped enumerator is declared in the scope that immediately contains the enum-specifier.
Each scoped enumerator is declared in the scope of the enumeration. These names obey the scope rules defined for all
names in (3.3) and (3.4). [Example:
enum direction { left='l', right='r' };

void g() {
    direction d; // OK
    d = left; // OK
    d = direction::right; // OK
}

enum class altitude { high='h', low='l' };

void h() {
    altitude a; // OK
    a = high; // error: high not in scope
    a = altitude::low; // OK
}

/* end example */ An enumerator declared in class scope can be referred to using the class member access operators (::, . (dot) and -> (arrow)), see 5.2.5. [Example:

struct X {
    enum direction { left='l', right='r' };
    int f(int i) { return i==left ? 0 : i==right ? 1 : 2; }
};

void g(X* p) {
    direction d; // error: direction not in scope
    int i;
    i = p->f(left); // error: left not in scope
    i = p->f(X::right); // OK
    i = p->f(p->left); // OK
    // ...
}

/* end example */

7.3 Namespaces [basic.namespace]

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 [namespace.def]

The grammar for a namespace-definition is

Draft
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
original-namespace-definition:
  \texttt{inline_{opt} namespace} identifier \{ namespace-body \}
extension-name-space-definition:
  \texttt{inline_{opt} namespace} original-namespace-name \{ namespace-body \}
unnamed-name-space-definition:
  \texttt{inline_{opt} namespace} \{ namespace-body \}
namespace-body:
  declaration-seq_{opt}

2 The \textit{identifier} in an \texttt{original-namespace-definition} shall not have been previously defined in the declarative region in which the \texttt{original-namespace-definition} appears. The \textit{identifier} in an \texttt{original-namespace-definition} is the name of the namespace. Subsequently in that declarative region, it is treated as an \texttt{original-namespace-name}.

3 The \texttt{original-namespace-name} in an \texttt{extension-namespace-definition} shall have previously been defined in an \texttt{original-namespace-definition} in the same declarative region.

4 Every \texttt{namespace-definition} shall appear in the global scope or in a namespace scope (3.3.5).

5 Because a \texttt{namespace-definition} contains declarations in its \texttt{namespace-body} and a \texttt{namespace-definition} is itself a declaration, it follows that \texttt{namespace-defintions} can be nested. \textit{Example:}

```cpp
namespace Outer {
  int i;
  namespace Inner {
    void f() { i++; } // Outer::i
    int i;
    void g() { i++; } // Inner::i
  }
}
```

\textit{— end example }]

6 The \textit{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. \textit{Example:}

```cpp
namespace Q {
  namespace V {
    void f(); // enclosing namespaces are the global namespace, Q, and Q::V
  }
  void f() { // enclosing namespaces are the global namespace, Q, and Q::V
    class C { void m(); };  
  }
}
```

Draft
7.3 Namespaces

If the optional initial `inline` keyword appears in a namespace-definition for a particular namespace, that namespace is declared to be an inline namespace. The `inline` keyword may be used on an extension-namespace-definition only if it was previously used on the original-namespace-definition for that namespace.

Members of an inline namespace can be used in most respects as though they were members of the enclosing namespace. Specifically, the inline namespace and its enclosing namespace are considered to be associated namespaces (3.4.2) of one another, and a using-directive (7.3.4) that names the inline namespace is implicitly inserted into the enclosing namespace. Furthermore, each member of the inline namespace can subsequently be explicitly instantiated (14.7.2) or explicitly specialized (14.7.3) as though it were a member of the enclosing namespace. Finally, looking up a name in the enclosing namespace via explicit qualification (3.4.3.2) will include members of the inline namespace brought in by the using-directive even if there are declarations of that name in the enclosing namespace.

These properties are transitive: if a namespace $N$ contains an inline namespace $M$, which in turn contains an inline namespace $O$, then the members of $O$ can be used as though they were members of $M$ or $N$. The set of namespaces consisting of the innermost non-inline namespace enclosing an inline namespace $O$, together with any intervening inline namespaces, is the enclosing namespace set of $O$.

7.3.1.1 Unnamed namespaces

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.\(^2\)

```
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() {
```

\(^2\) 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.
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

[namespace.memdef]

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

83) this implies that the name of the class or function is unqualified.
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);
template <class T> void f2(T);
namespace A {
    class X {
        friend void f(X); // A::f(X) is a friend
    };
    class Y {
        friend void g(); // A::g is a friend
        friend void h(int); // A::h is a friend
            // ::h not considered
        friend void f2<(int); // ::f2<(int) is a friend
    };
    // A::f, A::g and A::h are not visible here
    X x;
    void g() { f(x); } // definition of A::g
    void f(X) { /* ... */ } // definition of A::f
    void h(int) { /* ... */ } // definition of A::h
    // A::f, A::g and A::h are visible here and known to be friends
};

using A::x;

void h() {
    A::f(x);
    A::X::f(x); // error: f is not a member of A::X
    A::X::Y::g(); // error: g is not a member of A::X::Y
}

— end example ]

7.3.2 Namespace alias [namespace.alias]

A namespace-alias-definition declares an alternate name for a namespace according to the following grammar:

```
namespace-alias-definition:
    identifier
    namespace-alias-definition = qualified-name-specifier ;
    qualified-name-specifier:
        ::opt namespace-name
```

The identifier in a namespace-alias-definition is a synonym for the name of the namespace denoted by the qualified-name-specifier and becomes a namespace-alias. [Note: when looking up a namespace-name in a namespace-alias-definition, only namespace names are considered, see 3.4.6. — end note]
In a declarative region, a namespace-alias-definition can be used to redefine a namespace-alias declared in that declarative region to refer only to the namespace to which it already refers. [Example: the following declarations are well-formed:

```cpp
namespace Company_with_very_long_name { /* ... */ }
namespace CWVLN = Company_with_very_long_name;
namespace CWVLN = Company_with_very_long_name; // OK: duplicate
namespace CWVLN = CWVLN;
```

— end example ]

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 typename_opt ::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 ] If a using-declaration names a constructor (3.4.3.1), it implicitly declares a set of constructors in the class in which the using-declaration appears (12.9); otherwise the name specified in a using-declaration is a synonym for the name of some entity declared elsewhere.

Every using-declaration is a declaration and a member-declaration and so can be used in a class definition. [Example:

```cpp
struct B {
    void f(char);
    void g(char);
    enum E { e };
    union { int x; };
};

struct D : B {
    using B::f;
    void f(int) { f('c'); } // calls B::f(char)
    void g(int) { g('c'); } // recursively calls D::g(int)
};
```

— end example ]

In a using-declaration used as a member-declaration, the nested-name-specifier shall name a base class of the class being defined. Such a using-declaration If such a using-declaration names a constructor, the nested-name-specifier shall name a direct base class of the class being defined; otherwise it introduces the set of declarations found by member name lookup (10.2, 3.4.3.1). [Example:
class C {
    int g();
};

class D2 : public B {
    using B::f;  // OK: B is a base of D2
    using B::e;  // OK: e is an enumerator of base B
    using B::x;  // OK: x is a union member of base B
    using C::g;  // error: C isn’t a base of D2
};

— end example ]

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:

    struct A {
        template <class T> void f(T);
        template <class T> struct X { };  
    };
    struct B : A {
        using A::f<double>;  // ill-formed
        using A::X<int>;    // ill-formed
    };

    — end example ]

6 A using-declaration shall not name a namespace.

7 A using-declaration shall not name a scoped enumerator.

8 A using-declaration for a class member shall be a member-declaration. [Example:

    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.
    }

Draft
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
}

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:

```cpp
namespace A {
    int i;
}

namespace A1 {
    using A::i;
    using A::i;        // OK: double declaration
}

void f() {
    using A::i;
    using A::i;        // error: double declaration
}

struct B {
    int i;
};

struct X : B {
    using B::i;
    using B::i;        // error: double member declaration
};

— end example ]

Draft
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:

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

using A::f; // f is a synonym for A::f;
// that is, for A::f(int).
namespace A {
  void f(char);
}

void foo() {
  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-declarations`. [Example:

```cpp
namespace A {
  int x;
}

namespace B {
  int i;
  struct g { }; // OK: hides struct g
  struct x { };
  void f(int);
  void f(double);
  void g(char);
}

void func() {
  int i;
  using B::i; // error: i declared twice
  void f(char);
```
using B::f;  // OK: each f is a function
f(3.5);  // calls B::f(double)
using B::g;
g('a');  // calls B::g(char)
struct g g1;  // g1 has class type B::g
using B::x;
using A::x;  // OK: hides struct B::x
x = 99;  // assigns to A::x
struct x x1;  // x1 has class type B::x
}

— end example ]

If a function declaration in namespace scope or block scope has the same name and the same parameter 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-declarations 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-declarations, the function call is ill-formed. [Example:

namespace B {
    void f(int);
    void f(double);
}
namespace C {
    void f(int);
    void f(double);
    void f(char);
}

void h() {
    using B::f;  // B::f(int) and B::f(double)
    using C::f;  // C::f(int), C::f(double), and C::f(char)
    f('h');  // calls C::f(char)
    f(1);  // error: ambiguous: B::f(int) or C::f(int)?
    void f(int);  // error: f(int) conflicts with C::f(int) and B::f(int)
}

— end example ] — end note ]

When a using-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). [Note: For using-declarations that name a constructor, see 12.9. — end note ] [Example:

struct B {
    virtual void f(int);
    virtual void f(char);
    void g(int);
    void h(int);
};
struct D : B {
    using B::f;
    void f(int);  // OK: D::f(int) overrides B::f(int);
    using B::g;
    void g(char); // OK
    using B::h;
    void h(int);  // OK: D::h(int) hides B::h(int)
};

void k(D* p)
{
    p->f(1);      // calls D::f(int)
    p->f('a');    // calls D::f(char)
    p->g(1);      // calls B::g(int)
    p->g('a');    // calls D::g(char)
}

— end example ]

16 [ Note: two using-declarations 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-declarations, 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 The access rules for inheriting constructors are specified in 12.9; otherwise all 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,

```cpp
struct A { int x();
struct B : A {
struct C : A {
    using A::x;
    int x(int);
};

struct D : B, C {
    using C::x;
    int x(double);
};
```
int f(D* d) {
    return d->x();  // ambiguous: B::x or C::x
}

— end note ]

19 The alias created by the using-declaration has the usual accessibility for a member-declaration. [Note: A using-declaration that names a constructor does not create aliases; see 12.9 for the pertinent accessibility rules. — end note ] [Example:

```cpp
class A {
    private:
        void f(char);
    public:
        void f(int);
    protected:
        void g();
};

class B : public A {
    using A::f;  // error: A::f(char) is inaccessible
    public:
        using A::g;  // B::g is a public synonym for A::g
};

— end example ]

20 [ Note: use of access-declarations (11.3) is deprecated; member using-declarations provide a better alternative. — end note ]

21 If a using-declaration uses the keyword typename and specifies a dependent name (14.6.2), the name introduced by the using-declaration is treated as a typedef-name (7.1.3).

7.3.4 Using directive [namespace.udir]

using-directive:
    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;
        }
    }
}
```

Draft
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-directives, the effect is as if the using-directives 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;
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.  

[ Example:  

[84] During name lookup in a class hierarchy, some ambiguities may be resolved by considering whether one member hides the other along some
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 \texttt{asm} declaration [dcl.asm]

An \texttt{asm} declaration has the form

\begin{verbatim}
asm-definition:
    \texttt{asm} ( string-literal ) ;
\end{verbatim}

The \texttt{asm} declaration is conditionally-supported; its meaning is implementation-defined. \textit{[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 \textit{language linkage}. \textit{[Note: Some of the properties associated with an entity with language linkage are specific to each implementation and are not described here. For example, a particular language linkage may be associated with a particular form of representing names of objects and functions with external linkage, or with a particular calling convention, etc. — end note] The default language linkage of all function types, function names, and variable names is C++ language linkage. Two function types with different language linkages are distinct types even if they are otherwise identical.}

paths (10.2). There is no such disambiguation when considering the set of names found as a result of following \textit{using-directives}.
Linkage (3.5) between C++ and non-C++ code fragments can be achieved using a linkage-specification:

\[
\text{linkage-specification:}
\begin{align*}
\text{extern } & \text{s\textit{tring-literal} } \{ \text{declaration-seq}_{opt} \} \\
\text{extern } & \text{s\textit{tring-literal} declaration}
\end{align*}
\]

The \textit{string-literal} indicates the required language linkage. This International Standard specifies the semantics for the \textit{string-literals} "C" and "C++". Use of a \textit{string-literal} other than "C" or "C++" is conditionally-supported, with implementation-defined semantics. [\textit{Note:} Therefore, a linkage-specification with a \textit{string-literal} that is unknown to the implementation requires a diagnostic. — \textit{end note}] [\textit{Note:} It is recommended that the spelling of the \textit{string-literal} be taken from the document defining that language. For example, Ada (not ADA) and Fortran or FORTRAN, depending on the vintage. — \textit{end note}]

Every implementation shall provide for linkage to functions written in the C programming language, "C", and linkage to C++ functions, "C++". [\textit{Example:}
\begin{verbatim}
complex sqrt(complex); // C++ linkage by default
 extern "C" {
  double sqrt(double); // C linkage
 }
\end{verbatim}
— \textit{end example}]

Linkage specifications nest. When linkage specifications nest, the innermost one determines the language linkage. A linkage specification does not establish a scope. A \textit{linkage-specification} shall occur only in namespace scope (3.3). In a \textit{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 \textit{linkage-specification}. [\textit{Example:}
\begin{verbatim}
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();
 FUNC f2; // the name f2 has C++ language linkage and the
 // function's type has C language linkage
 extern "C" FUNC f3; // the name of function f3 and the function's type
 // have C language linkage
 void (**pf2)(FUNC*); // the name of the variable pf2 has C++ linkage and
 // the type of pf2 is pointer to C++ function that
 // takes one parameter of type pointer to C function
 extern "C" {
  static void f4(); // the name of the function f4 has
  // internal linkage (not C language
  // linkage) and the function's type
  // has C language linkage.
 }
 extern "C" void f5() {
  extern void f4(); // OK: Name linkage (internal)
  // and function type linkage (C
  // language linkage) gotten from
  // previous declaration.
\end{verbatim}

Draft
7.5 Linkage specifications

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 ]

If two declarations of the same function or object specify different linkage-specifications (that is, the linkage-specifications of these declarations specify different string-literals), 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.
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—

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—

[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—

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

1 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

2 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.

3 Each *init-declarator* in a declaration is analyzed separately as if it was in a declaration by itself.\(^\text{85}\)

4 Declarators have the syntax

   declarator:  
   direct-declarator  
   ptr-operator declarator

\(^{85}\) A declaration with several declarators is usually equivalent to the corresponding sequence of declarations each with a single declarator. That is

   T D1, D2, ..., Dn;

   is usually equivalent to

   T D1; T D2; ... T Dn;

   where T is a *decl-specifier-seq* and each D 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

   ```c
   struct S ... ;
   S S, T; // declare two instances of struct S
   ```

   which is not equivalent to

   ```c
   struct S ... ;
   S S;
   S T; // error
   ```
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.

```plaintext
type-id:
  type-specifier-seq abstract-declarator_opt

abstract-declarator:
  ptr-operator abstract-declarator_opt
direct-abstract-declarator
...
```

A class-name has special meaning in a declaration of the class of that name and when qualified by that name using the scope resolution operator :: (5.1, 12.1, 12.4).
direct-abstract-declarator:
  direct-abstract-declarator\_opt ( parameter-declaration-clause )
  cv-qualifier-seq\_opt ref-qualifier\_opt exception-specification\_opt
  direct-abstract-declarator\_opt ( parameter-declaration-clause )
  cv-qualifier-seq\_opt ref-qualifier\_opt exception-specification\_opt \rightarrow type\_id
  direct-abstract-declarator\_opt [ constant-expression\_opt ]

It is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. [Example:

```
int // int i
int * // int *pi
int *[3] // int *p[3]
int [3] // int (*p3i)[3]
int *() // int *f()
int (*)(double) // int (*pf)(double)
```

]name respectively the types “int,” “pointer to int,” “array of 3 pointers to int,” “pointer to array of 3 int,” “function of (no parameters) returning pointer to int,” and “pointer to a function of (double) returning int.” — end example]

[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:

```
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 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:

```
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:
8.2 Ambiguity resolution

```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
}
```

4 For another example,

```c
template <class T>
struct S {
    T *p;
};
S<int()> x;               // type-id
S<int(1)> y;              // expression (ill-formed)
```

5 For another example,

```c
void foo() {
    sizeof(int(1));     // expression
    sizeof(int());      // type-id (ill-formed)
}
```

6 For another example,

```c
void foo() {
    (int(1));           // expression
    (int())1;           // type-id (ill-formed)
}
```

— end example ]

7 Another ambiguity arises in a parameter-declaration-clause of a function declaration, or in a type-id that is the operand of a sizeof or typeid operator, when a type-name is nested in parentheses. In this case, the choice is between the declaration of a parameter of type pointer to function and the declaration of a parameter with redundant parentheses around the declarator-id. The resolution is to consider the type-name as a simple-type-specifier rather than a declarator-id. [Example:

```c
class C { };
void f(int(C)) { }       // void f(int(*fp)(C c)) { }
// not: void f(int C);

int g(C);

void foo() {
    f(1);                 // error: cannot convert 1 to function pointer
    f(g);                // OK
}
```
For another example,

```cpp
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 of the class or namespace to which the qualifier refers (or of an inline namespace within that scope (7.3.1)), 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]

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`.

Thus, a declaration of a particular identifier has the form

```cpp
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.

First, the `decl-specifier-seq` determines a type. In a declaration

```cpp
T D
```

the `decl-specifier-seq` `T` determines the type `T`. [Example: in the declaration

```cpp
int unsigned i;
```

the type specifiers `int` `unsigned` determine the type “`unsigned int`” (7.1.6.2). — end example]

In a declaration `T D` where `D` is an unadorned identifier the type of this identifier is “`T`.”

In a declaration `T D` where `D` has the form

```cpp
(D1)
```

the type of the contained `declarator-id` is the same as that of the contained `declarator-id` in the declaration

```cpp
T D1
```
Parentheses do not alter the type of the embedded `declarator-id`, but they can alter the binding of complex declarators.

### 8.3.1 Pointers

1. In a declaration `T D` where `D` has the form
   
   ```
   * cv-qualifier-seq opt D1
   ```

   and the type of the identifier in the declaration `T D1` is "derived-declarator-type-list T," then the type of the identifier of `D` is "derived-declarator-type-list cv-qualifier-seq pointer to T." The cv-qualifiers apply to the pointer and not to the object pointed to.

2. [Example: the declarations]

   ```
   const int ci = 10, *pc = &ci, *const cpc = pc, **ppc;
   int i, *p, *const cp = &i;
   ```

   declare `ci`, a constant integer; `pc`, a pointer to a constant integer; `cpc`, a constant pointer to a constant integer; `ppc`, a pointer to a pointer to a constant integer; `i`, an integer; `p`, a pointer to integer; and `cp`, a constant pointer to integer. The value of `ci`, `cpc`, and `cp` cannot be changed after initialization. The value of `pc` can be changed, and so can the object pointed to by `cp`. Examples of some correct operations are

   ```
   i = ci;
   *cp = ci;
   pc++;
   pc = cpc;
   pc = p;
   ppc = &pc;
   ```

   Examples of ill-formed operations are

   ```
   ci = 1; // error
   ci++;
   *pc = 2; // error
   cp = &ci; // error
   cpc++;
   p = pc; // error
   ppc = &p; // error
   ```

   Each is unacceptable because it would either change the value of an object declared `const` or allow it to be changed through a cv-unqualified pointer later, for example:

   ```
   *ppc = &ci; // OK, but would make p point to ci ...
   // ... because of previous error
   *p = 5; // clobber ci
   ```

   — end example ]

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

In a declaration T D where D has either of the forms

& 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:

typedef int& A;
const A aref = 3;  // ill-formed; non-const reference initialized with rvalue

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.

2 A reference type that is declared using & is called an lvalue reference, and a reference type that is declared using && is called an rvalue reference. Lvalue references and rvalue references are distinct types. Except where explicitly noted, they are semantically equivalent and commonly referred to as references.

3 [Example:

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.

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,

struct link {
  link* next;
};

link* first;

void h(link*& p) {  // p is a reference to pointer
  p->next = first;
  first = p;
8.3 Meaning of declarators

```c
p = 0;
}

void k() {
    link* q = new link;
    h(q);
}
```

declares `p` to be a reference to a pointer to `link` so `h(q)` will leave `q` with the value zero. See also 8.5.3. — end example]

4 It is unspecified whether or not a reference requires storage (3.7).

5 There shall be no references to references, no arrays of references, and no pointers to references. The declaration of a reference shall contain an initializer (8.5.3) except when the declaration contains an explicit extern specifier (7.1.1), is a class member (9.2) declaration within a class definition, or is the declaration of a parameter or a return type (8.3.5); see 3.1. A reference shall be initialized to refer to a valid object or function. [Note: in particular, a null reference cannot exist in a well-defined program, because the only way to create such a reference would be to bind it to the “object” obtained by 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

[dlm.mptr]

1 In a declaration `T D` where `D` has the form

```c
::opt nested-name-specifier * 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`.”

2 [Example:

```c
struct X {
    void f(int);
    int a;
};
struct Y;

int X::* pmi = &X::a;
void (X::* pmf)(int) = &X::f;
double X::* pmd;
char Y::* pmc;
```

declares `pmi`, `pmf`, `pmd` and `pmc` to be a pointer to a member of `X` of type `int`, a pointer to a member of `X` of type `void(int)`, a pointer to a member of `X` of type `double` and a pointer to a member of `Y` of type `char` respectively. The declaration of `pmd` is well-formed even though `X` has no members of type `double`. Similarly, the declaration of `pmc` is well-formed even though `Y` is an incomplete type. `pmi` and `pmf` can be used like this:

```c
X obj;
// ...
obj.*pmi = 7;       // assign 7 to an integer
```
175 Declarators

8.3 Meaning of declarators

// member of obj
(obj.*pmf)(7);  // call a function member of obj
// with the argument 7

— end example ]

3 A pointer to member shall not point to a static member of a class (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 [dcl.array]

1 In a declaration \(T \ D\) where \(D\) has the form

\[
D_1 \ [\text{constant-expression}_{opt}\ ]
\]

and the type of the identifier in the declaration \(T \ D_1\) is “\(\text{derived-declarator-type-list} \ T\),” then the type of the identifier of \(D\) is an array type; if the type of the identifier of \(D\) contains the 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 “\(\text{derived-declarator-type-list} \ \text{array of} \ N \ T\).” 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 “\(\text{derived-declarator-type-list} \ \text{array of unknown bound of} \ T\),” an incomplete object type. The type “\(\text{derived-declarator-type-list} \ \text{array of} \ N \ T\)” is a different type from the type “\(\text{derived-declarator-type-list} \ \text{array of unknown bound of} \ T\),” see 3.9. Any type of the form “\(\text{cv-qualifier-seq} \ \text{array of} \ N \ T\)” is adjusted to “\(\text{array of} \ N \ \text{cv-qualifier-seq} \ T\),” and similarly for “\(\text{array of unknown bound of} \ T\).” [Example:

\[
\text{typedef int A[5], AA[2][3];} \\
\text{typedef const A CA;} \\
\text{typedef const AA CAA;} \\
\]

// type is “array of 5 const int”
// type is “array of 2 array of 3 const int”

— end example ] [Note: an “\(\text{array of} \ N \ \text{cv-qualifier-seq} \ T\)” 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 “\(\text{array of} \ N \ T\).”

[Example:

\[
\text{float fa[17], *afp[17];}
\]

declares an array of float numbers and an array of pointers to float numbers. For another example,
8.3 Meaning of declarators

```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 

5 [ 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 $E_1[E_2]$ is identical to $*(E_1)+(E_2)$. Because of the conversion rules that apply to $+$, if $E_1$ is an array and $E_2$ an integer, then $E_1[E_2]$ refers to the $E_2$-th member of $E_1$. 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 a $(n-1)$-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.

8 [ 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 $(x+1)$, 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 ]

9 [ Note: it follows from all this that arrays in C++ are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations. — end note ]

8.3.5 Functions

1 In a declaration $T D$ where $D$ has the form
```c
    D1 ( parameter-declaration-clause ) cv-qualifier-seq_opt ref-qualifier_opt exception-specification_opt
```

and the type of the contained declarator-id in the declaration $T D1$ is “derived-declarator-type-list $T$,” the type of the declarator-id in $D$ is “derived-declarator-type-list function of (parameter-declaration-clause ) cv-qualifier-seq_opt ref-qualifier_opt returning $T$.” A type of this form is a function type.  

2 In a declaration $T D$ where $D$ has the form
```c
    D1 ( parameter-declaration-clause ) cv-qualifier-seq_opt ref-qualifier_opt exception-specification_opt \rightarrow type-id
```

and the type of the contained declarator-id in the declaration $T D1$ is “derived-declarator-type-list $T$,” $T$ shall be the single type-specifier auto and the derived-declarator-type-list shall be empty. Then the type of the declarator-id in $D$ is

---

86 As indicated by the syntax, cv-qualifiers are a significant component in function return types.
“function of (parameter-declaration-clause) cv-qualifier-seq_opt ref-qualifier_opt returning type-id.” Such a function type has a late-specified return type.

3 The type-id in this form includes the longest possible sequence of abstract-declarators. [Note: This resolves the ambiguous binding of array and function declarators. [Example:

```c
auto f() -> int(*)[4]; // function returning a pointer to array[4] of int
// not function returning array[4] of pointer to int
```

—end example] —end note]

4 A type of either form is a function type. [87]

```
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
```

5 The parameter-declaration-clause determines the arguments that can be specified, and their processing, when the function is called. [Note: the parameter-declaration-clause is used to convert the arguments specified on the function call; see 5.2.2. —end note] If the parameter-declaration-clause is empty, the function takes no arguments. 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 (14.5.3), the number of arguments shall be equal to or greater than the number of parameters that do not have a default argument and are not function parameter packs. Where syntactically correct and where “...” is not part of an abstract-declarator, “,...” is synonymous with “...”. [Example: the declaration

```c
int printf(const char*, ...);
```

declares a function that can be called with varying numbers and types of arguments.

```c
printf("hello world");
printf("a=%%d b=%%d", a, b);
```

However, the first argument must be of a type that can be converted to a const char* —end example] [Note: the standard header <stdarg> contains a mechanism for accessing arguments passed using the ellipsis (see 5.2.2 and 18.8). — end note]

6 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

---

[87] As indicated by syntax, cv-qualifiers are a significant component in function return types.
“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-qualifiers 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-specifiers 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.

7 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] A ref-qualifier 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 return type, the parameter-type-list, the ref-qualifier, 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]

8 [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]

9 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 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).

10 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
```

88) 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.
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 ]

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 ]

[ Example: the declaration

```c
int i,
    *pi,
    f(),
    *fpi(int),
    (*pif)(const char*, const char*),
    (*fpif(int))(int);
```

declares an integer `i`, a pointer `pi` to an integer, a function `f` taking no arguments and returning an integer, a function `fpi` taking an integer argument and returning a pointer to an integer, a pointer `pif` to a function which takes two pointers to constant characters and returns an integer, a function `fpif` taking an integer argument and returning a pointer to a function that takes an integer argument and returns an integer. It is especially useful to compare `fpi` and `pif`. The binding of `*fpi(int)` is `*(fpi(int))`, so the declaration suggests, and the same construction in an expression requires, the calling of a function `fpi`, and then using indirection through the (pointer) result to yield an integer. In the declarator `(*pif)(const char*, const char*)`, the extra parentheses are necessary to indicate that indirection through a pointer to a function yields a function, which is then called. — end example ] [Note: typedefs and late-specified return types are sometimes convenient when the return type of a function is complex. For example, the function `fpif` above could have been declared

```c
typedef int IFUNC(int);
IFUNC* fpif(int);
```

or

```c
auto fpif(int)->int(*)(int)
```

A late-specified return type is most useful for a type that would be more complicated to specify before the `declarator-id`:

```c
template <class T, class U> auto add(T t, U u) -> decltype(t + u);
```

rather than

```c
template <class T, class U> auto add(T t, U u) ...
```
8.3 Meaning of declarators

```
template <class T, class U> decltype((*(T*)0) + (*(U*)0)) add(T t, U u);
```

— end note ]

13 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:
```
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.89)

8.3.6 Default arguments [dcl.fct.default]

1 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
```
void point(int = 3, int = 4);
```

declares a function that can be called with zero, one, or two arguments of type int. It can be called in any of these ways:
```
point(1,2); point(1); point();
```

The last two calls are equivalent to point(1,4) and point(3,4), respectively. — end example ]

3 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.90)

4 For non-template functions, default arguments can be added in later declarations of a function in the same scope. Declarations in different scopes have completely distinct sets of default arguments. That is, declarations in inner scopes

89) 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).

90) This means that default arguments cannot appear, for example, in declarations of pointers to functions, references to functions, or typedef declarations.
do not acquire default arguments from declarations in outer scopes, and vice versa. In a given function declaration, all
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, ...);  // OK, ellipsis is not a parameter so it can follow
// a parameter with a default argument
void f(int, int);
void f(int, int = 7);
void h() {
    f(3);          // OK, calls f(3, 7)
    void f(int = 1, int);  // error: does not use default
    // from surrounding scope
}
void m() {
    void f(int, int);  // has no defaults
    f(4);          // error: wrong number of arguments
    void f(int, int = 5);  // OK
    f(4);          // OK, calls f(4, 5);
    void f(int, int = 5);  // error: cannot redefine, even to
    // same value
}
void n() {
    f(6);          // OK, calls f(6, 7)
}

— end example ]
```

For a given inline function defined in different translation units, the accumulated sets of default arguments at the end of the translation units shall be the same; see 3.2. If a friend declaration specifies a default argument expression, that declaration shall be a definition and shall be the only declaration of the function or function template in the translation unit.

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))
    }
}

— end example ] [Note: in member function declarations, names in default argument expressions are looked up as
Draft
described in 3.4.1. Access checking applies to names in default argument expressions as described in clause 11. — end note]

6 Except for member functions of class templates, the default arguments in a member function definition that appears outside of the class definition are added to the set of default arguments provided by the member function declaration in the class definition. 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:

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:

  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:

  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:

  int a;
  int f(int a, int b = a); // error: parameter a
  // used as default argument
typedef int I;
  int g(float I, int b = I(2)); // error: parameter I found
  int h(int a, int b = sizeof(a)); // error, parameter a used
  // in default argument

Draft
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.]

```cpp
int b;
class X {
  int a;
  int mem1(int i = a); // error: non-static member a
                     // used as default argument
  int mem2(int i = b); // OK; use X::b
  static int b;
};
```

The declaration of `X::mem2()` is meaningful, however, since no object is needed to access the static member `X::b`. Classes, objects, and members are described in clause 9. [Example:] A default argument is not part of the type of a function. [Example:]

```cpp
int f(int = 0);
void h() {
  int j = f(1);
  int k = f(); // OK, means f(0)
}
int (*p1)(int) = &f;
int (*p2)() = &f; // error: type mismatch
```

—end example— When a declaration of a function is introduced by way of a using-declaration (7.3.3), any default argument information associated with the declaration is made known as well. If the function is redeclared thereafter in the namespace with additional default arguments, the additional arguments are also known at any point following the redeclaration where the using-declaration is in scope.

A virtual function call (10.3) uses the default arguments in the declaration of the virtual function determined by the static type of the pointer or reference denoting the object. An overriding function in a derived class does not acquire default arguments from the function it overrides. [Example:]

```cpp
struct A {
  virtual void f(int a = 7);
};
struct B : public A {
  void f(int a);
};
void m() {
  B* pb = new B;
  A* pa = pb;
  pa->f(); // OK, calls pa->B::f(7)
  pb->f(); // error: wrong number of arguments for B::f()
}
```

Draft
8.4 Function definitions

Function definitions have the form

```
function-definition:
  decl-specifier-seq_opt declarator function-body
  decl-specifier-seq_opt declarator = default ;
  decl-specifier-seq_opt declarator = delete ;

function-body:
  ctor-initializer_opt compound-statement
  function-try-block
```

Any informal reference to the body of a function should be interpreted as a reference to the non-terminal `function-body`.

The `declarator` in a `function-definition` shall have the form

```
D1 ( parameter-declaration-clause ) cv-qualifier-seq_opt ref-qualifier_opt exception-specification_opt
```

as described in §8.3.5. A function shall be defined only in namespace or class scope.

[Example: a simple example of a complete function definition is

```c
int max(int a, int b, int c) {
    int m = (a > b) ? a : b;
    return (m > c) ? m : c;
}
```

Here `int` is the `decl-specifier-seq`; `max(int a, int b, int c)` is the `declarator`; `{ /* ... */ }` is the `function-body`. — end example]}

A `ctor-initializer` is used only in a constructor; see §12.1 and §12.6.

A `cv-qualifier-seq` or a `ref-qualifier` (or both) can be part of a non-static member function declaration, non-static member function definition, or pointer to member function only (§8.3.5); see §9.3.2.

[Note: unused parameters need not be named. For example,

```c
void print(int a, int) {
    std::printf("a = %d\n",a);
}
```

— end note]

In the `function-body`, a `function-local predefined variable` denotes a 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

```
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.\(^{91}\)

\(^{91}\)Implementations are permitted to provide additional predefined variables with names that are reserved to the implementation (§17.4.3.2.2). If a predefined variable is not used (3.2), its string value need not be present in the program image.
A function definition of the form:

\[
\text{decl-specifier-seq} \quad \text{opt} \quad \text{declarator} = \text{default} ;
\]

is called an \textit{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 \textit{that would be implicitly defined as deleted shall not be explicitly defaulted}. A special member function is \textit{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. \textit{Note:} while an implicitly-declared special member function is \textit{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. \textit{— end note} \[ Example:

\begin{verbatim}
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
};
inline nontrivial3::~nontrivial3() = default;  // not first declaration
\end{verbatim}
\textit{— end example} \]

10 A function definition of the form:

\[
\text{decl-specifier-seq} \quad \text{opt} \quad \text{declarator} = \text{delete} ;
\]
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:

```c
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

```c
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.

```c
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

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).
Automatic, register, static, and external variables of namespace scope can be initialized by arbitrary expressions involving literals and previously declared variables and functions. [Example:

```c
int f(int);
int a = 2;
int b = f(a);
int c(b);
```

— end example ]

[Note: default argument expressions are more restricted; see 8.3.6.

The order of initialization of static objects is described in 3.6 and 6.7. —end note]

To zero-initialize an object of type \( T \) means:

- if \( T \) is `std::nullptr_t`, the object is set to the value of `nullptr`;
- otherwise, 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 \( T \);
- 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 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-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);
- 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-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-provided constructor, then every non-static data member and base-class component of \( T \) is value-initialized;
- if \( T \) is an array type, then each element is value-initialized;
- otherwise, the object is zero-initialized.

A program that calls for default-initialization or value-initialization of an entity of reference type is ill-formed. 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.

---

92) As specified in 4.10, converting an integral constant expression whose value is 0 to a pointer type results in a null pointer value.
93) This member must not be static, by virtue of the requirements in 9.5.
94) Value-initialization for such a class object may be implemented by zero-initializing the object and then calling the default constructor.
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]

An object whose initializer is an empty set of parentheses, i.e., (), shall be value-initialized. [Note: since () is not permitted by the syntax for initializer, X a(); is not the declaration of an object of class X, but the declaration of a function taking no argument and returning an X. The form () is permitted in certain other initialization contexts (5.3.4, 5.2.3, 12.6.2). — end note]

If no initializer is specified for an object, and the object is of (possibly cv-qualified) non-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-provided default constructor. Otherwise, if no initializer is specified for a non-static object, the object and its subobjects, if any, have an indeterminate initial value95); if the object or any of its subobjects are of const-qualified type, the program is ill-formed.

An initializer for a static member is in the scope of the member’s class. [Example: int a;

struct X {
    static int a;
    static int b;
};

int X::a = 1;
int X::b = a; //X::b = X::a

— end example]

The form of initialization (using parentheses or =) is generally insignificant, but does matter when the initializer or 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.

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 copy-initialization and is equivalent to the form T x = a;

The initialization that occurs in new expressions (5.3.4), static_cast expressions (5.2.9), functional notation type conversions (5.2.3), and base and member initializers (12.6.2) is called direct-initialization and is equivalent to the form T x(a);

If T is a scalar type, then a declaration of the form T x = { a }; is equivalent to

95) This does not apply to aggregate objects with automatic storage duration initialized with an incomplete brace-enclosed initializer-list; see 8.5.1.
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). If the conversion cannot be done or is ambiguous, the initialization is ill-formed. The function selected is called with the initializer expression as its argument; if the function is a constructor, the call initializes a temporary of the cv-unqualified version of the destination type. The temporary is 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.]

— end note ]
An *initializer-clause* followed by an ellipsis is a pack expansion (14.5.3).

### 8.5.1 Aggregates

An **aggregate** is an array or a class (clause 9) with no user-provided constructors (12.1), no private or protected non-static data members (clause 11), no base classes (clause 10), and no virtual functions (10.3).

When an aggregate is initialized the **initializer** can contain an **initializer-clause** consisting of a brace-enclosed, comma-separated list of **initializer-clauses** 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. 

```cpp
struct A {
    int x;
    struct B {
        int i;
        int j;
    } b;
} a = { 1, { 2, 3 } };  
```

initializes `a.x` with 1, `a.b.i` with 2, `a.b.j` with 3. — end example]

An aggregate that is a class can also be initialized with a single expression not enclosed in braces, as described in 8.5.

An array of unknown size initialized with a brace-enclosed **initializer-list** containing *n* **initializer-clauses**, where *n* shall be greater than zero, is defined as having *n* elements (8.3.4).

```cpp
int x[] = { 1, 3, 5 };  
```

declares and initializes `x` as a one-dimensional array that has three elements since no size was specified and there are three initializers. — end example]

An empty initializer list `{}` shall not be used as the **initializer-clause** for an array of unknown bound. \(^\text{96}\)

Static data members and anonymous bit fields are not considered members of the class for purposes of aggregate initialization. 

```cpp
struct A {
    int i;
    static int s;
    int j;
    int :17;
    int k;
} a = { 1, 2, 3 };  
```

Here, the second initializer 2 initializes `a.j` and not the static data member `A::s`, and the third initializer 3 initializes `a.k` and not the anonymous bit field before it. — end example]

An **initializer-list** is ill-formed if the number of **initializer-clauses** exceeds the number of members or elements to initialize. 

```cpp
char cv[4] = { 'a', 'm', 'd', 'f', 0 };  // error  
```

\(^{96}\) The syntax provides for empty **initializer-lists**, but nonetheless C++ does not have zero length arrays.
is ill-formed. — end example ]

7 If there are fewer *initializer-clauses* in the list than there are members in the aggregate, then each member not explicitly initialized shall be value-initialized (8.5). [ Example:

```c
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 ]

8

```c
struct S { };
struct A {
 S s;
 int i;
} a = { { }, 3 };
```

If an aggregate class `C` contains a subaggregate member `m` that has no members for purposes of aggregate initialization, the *initializer-clause* for `m` shall not be omitted from an *initializer-list* for an object of type `C` unless the *initializer-clauses* for all members of `C` following `m` are also omitted. [ Example:

```c
struct S { } s;
struct A {
 S s1;
 int i1;
 S s2;
 int i2;
 S s3;
 int i3;
} a = {
 { }, // Required initialization
 0,
 s, // Required initialization
 0
}; // Initialization not required for A::s3 because A::i3 is also not initialized
```

— end example ]

9 If an incomplete or empty *initializer-list* leaves a member of reference type uninitialized, the program is ill-formed.

10 When initializing a multi-dimensional array, the *initializer-clauses* 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 }
};
```

Draft
initializes the first column of \( y \) (regarded as a two-dimensional array) and leaves the rest zero. — end example ]

Braces can be elided in an initializer-list as follows. If the initializer-list begins with a left brace, then the succeeding comma-separated list of initializer-clauses initializes the members of a subaggregate; it is erroneous for there to be more initializer-clauses than members. If, however, the initializer-list for a subaggregate does not begin with a left brace, then only enough initializer-clauses from the list are taken to initialize the members of the subaggregate; any remaining initializer-clauses are left to initialize the next member of the aggregate of which the current subaggregate is a member. [ Example:

\[
\begin{align*}
\text{float y[4][3]} & = \{ \\
& \{ 1, 3, 5 \}, \\
& \{ 2, 4, 6 \}, \\
& \{ 3, 5, 7 \}, \\
& \};
\end{align*}
\]

is a completely-braced initialization: 1, 3, and 5 initialize the first row of the array \( y[0] \), namely \( y[0][0] \), \( y[0][1] \), and \( y[0][2] \). Likewise the next two lines initialize \( y[1] \) and \( y[2] \). The initializer ends early and therefore \( y[3] \)'s elements are initialized as if explicitly initialized with an expression of the form \text{float()}\), that is, are initialized with \( 0.0 \). In the following example, braces in the initializer-list are elided; however the initializer-list has the same effect as the completely-braced initializer-list of the above example,

\[
\text{float y[4][3]} = \{ \\
1, 3, 5, 2, 4, 6, 3, 5, 7 \\
\};
\]

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:

\[
\begin{align*}
\text{struct A} & = \{ \\
& \text{int } i; \\
& \text{operator int()}; \\
& \}; \\
\text{struct B} & = \{ \\
& \text{A a1, a2;} \\
& \text{int z;} \\
& \}; \\
& \text{A a;} \\
& \text{B b = \{ 4, a, a \};}
\end{align*}
\]

Braces are elided around the initializer-clause for \( b.a1.i \). \( b.a1.i \) is initialized with 4, \( b.a2 \) is initialized with \( a \), \( b.z \) is initialized with whatever \text{a.operator int()} returns. — end example ]
[Note: An aggregate array or an aggregate class may contain members of a class type with a user-provided constructor (12.1). Initialization of these aggregate objects is described in 12.6.1. —end note]

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 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:

```c
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

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:

```c
char msg[] = "Syntax error on line %s\n"; 
```

shows a character array whose members are initialized with a string-literal. Note that because ‘\n’ is a single character and because a trailing ‘\0’ is appended, sizeof(msg) is 25. —end example]

There shall not be more initializers than there are array elements. [Example:

```c
char cv[4] = "asdf"; // error
```

is ill-formed since there is no space for the implied trailing ‘\0’. —end example]

8.5.3 References

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 
```


```c
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`”\(^{(97)}\) (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:

\(^{(97)}\) This requires a conversion function (12.3.2) returning a reference type.

Draft
double d = 2.0;
double& rd = d; // rd refers to d
cost double& rcd = d; // rcd refers to d

struct A { }
struct B : A { } b;
&A& ra = b; // ra refers to a subobject in b
cost && 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 : A { } b;
extern B f();
cost && rca = f(); // Bound to the & 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 array value (see 3.10).

— 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:
cost double& rcd2 = 2; // rcd2 refers to temporary with value 2.0
double&& rcd3 = 2; // rcd3 refers to temporary with value 2.0
cost volatile int cvi = 1;
cost 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

A class is a type. Its name becomes a class-name (9.1) within its scope.

```
class-name:
  identifier
  simple-template-id
```

Class-specifiers and elaborated-type-specifiers (7.1.6.3) are used to make class-names. An object of a class consists of a (possibly empty) sequence of members and base class objects.

```
class-specifier:
  class-head { member-specificationopt }

class-head:
  class-key identifier opt base-clause opt
  class-key nested-name-specifier identifier base-clause opt
  class-key nested-name-specifieropt simple-template-id base-clause opt
```

A class-specifier where the class-head omits the optional identifier defines an unnamed class.

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.

Complete objects and member subobjects of class type shall have nonzero size. [Note: class objects can be assigned, passed as arguments to functions, and returned by functions (except objects of classes for which copying has been restricted; see 12.8). Other plausible operators, such as equality comparison, can be defined by the user; see 13.5. — end note ]

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 ]

A trivial class is a class that:

- has a trivial default constructor (12.1),

[98]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.99)

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 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
    int i;
    int j;
    ~SL();
};
```
]

99) 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).
199 Classes

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 {
    // ...
};
```

```c
stat gstat; // use plain stat to
// define variable
```

Draft
### 9.1 Class names

```c
int stat(struct stat*);  // redeclare stat as function

void f() {
    struct stat* ps;  // struct prefix needed
    // to name struct stat
    stat(ps);  // call stat()
}
```

— end example ] A declaration consisting solely of class-key identifier; is either a redeclaration of the name in the current scope or a forward declaration of the identifier as a class name. It introduces the class name into the current scope. [Example:

```c
struct s { int a; };

void g() {
    struct s;  // hide global struct s
    // 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:

```c
class Vector;

class Matrix {
    // ...
    friend Vector operator*(const Matrix&, const Vector&);
};

class Vector {
    // ...
    friend Vector operator*(const Matrix&, const Vector&);
};

Declaration of friends is described in 11.4, operator functions in 13.5. — end example ] — end note] 3

[Note: An elaborated-type-specifier (7.1.6.3) can also be used as a type-specifier as part of a declaration. It differs from a class declaration in that if a class of the elaborated name is in scope the elaborated name will refer to it. —end note] 3

```c
[ Example:

```c
struct s { int a; };

void g(int s) {
    struct s* p = new struct s;  // global s
    p->a = s;  // local s
}

— end example ]
```

Draft
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

```cpp
member-specification:
    member-declaration member-specificationopt
    access-specifier : member-specificationopt

member-declaration:
    declSpecifier-seqopt member-declarator-listopt ;
    function-definition ;opt
    ::opt nested-name-specifier templateopt unqualified-id ;
    using-declaration
    static_assert-declaration
    template-declaration

member-declarator-list:
    member-declarator
    member-declarator-list , member-declarator

member-declarator:
    declarator pure-specifieropt
    declarator constant-initializeropt
    identifieropt : constant-expression

pure-specifier:
    = 0

constant-initializer:
    = constant-expression
```

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 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,

Draft
9.2 Class members

exception-specifications (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 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 or register 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 with the same access control (clause 11) are allocated so that later members have higher addresses within a class object. The order of allocation of non-static data members with different access control is unspecified (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.

14 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.

15 Two 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 \(3.9\).

16 Two 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 \(3.9\).

17 If a standard-layout union contains two or more standard-layout structs that share a common initial sequence, and if the standard-layout union object currently contains one of these standard-layout structs, it is permitted to inspect the common initial part of any of them. Two 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.

18 A pointer to a standard-layout struct object, suitably converted using a \texttt{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 standard-layout struct object, but not at its beginning, as necessary to achieve appropriate alignment. — end note \]

### 9.3 Member functions

1 Functions declared in the definition of a class, excluding those declared with a \texttt{friend} specifier \(11.4\), are called member functions of that class. A member function may be declared \texttt{static} in which case it is a \texttt{static} member function of its class \(9.4\); otherwise it is a \texttt{non-static} member function of its class \(9.3.1, 9.3.2\).

2 A member function may be defined \(8.4\) in its class definition, in which case it is an \texttt{inline} member function \(7.1.2\), or it may be defined outside of its class definition if it has already been declared but not defined in its class definition. A member function definition that appears outside of the class definition shall appear in a namespace scope enclosing the class definition. Except for member function definitions that appear outside of a class definition, and except for explicit specializations of member functions of class templates and member function templates \(14.7\) appearing outside of the class definition, a member function shall not be redeclared.

3 An \texttt{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 \texttt{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 \texttt{3.5.} — end note \]

4 There shall be at most one definition of a non-inline member function in a program; no diagnostic is required. There may be more than one \texttt{inline} member function definition in a program. See \texttt{3.2} and \texttt{7.1.2}.
9.3 Member functions

If the definition of a member function is lexically outside its class definition, the member function name shall be qualified by its class name using the :: operator. [Note: a name used in a member function definition (that is, in the parameter-declaration-clause including the default arguments (8.3.6) or in the member function body) is looked up as described in 3.4. — end note] [Example:

```c
struct X {
    typedef int T;
    static T count;
    void f(T);
};
void X::f(T t = count) {} 
```

The member function `f` of class `X` is defined in global scope; the notation `X::f` specifies that the function `f` is a member of class `X` and in the scope of class `X`. In the function definition, the parameter type `T` refers to the typedef member `T` declared in class `X` and the default argument `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.

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 \(X\), 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 \((\ast\textit{this})\) \((9.3.2)\) as the postfix-expression to the left of the \(\text{.} \) operator. 

\[\text{Note: if } C \text{ is not } X \text{ or a base class of } X, \text{ the class member access expression is ill-formed. — end note}\] 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:]

\[
\text{struct tnode} \\
\text{    char tword[20];} \\
\text{    int count;} \\
\text{    tnode *left;} \\
\text{    tnode *right;} \\
\text{    void set(char*, tnode* l, tnode* r);} \\
\};
\]

\[
\text{void tnode::set(char* w, tnode* l, tnode* r)} \{ \\
\text{    count = strlen(w)+1;} \\
\text{    if (sizeof(tword)<=count)} \\
\text{        perror("tnode string too long");} \\
\text{    strcpy(tword,w);} \\
\text{    left = l;} \\
\text{    right = r;} \\
\};
\]

\[
\text{void f(tnode n1, tnode n2)} \{ \\
\text{    n1.set("abc",&n2,0);} \\
\text{    n2.set("def",0,0);} \\
\};
\]

In the body of the member function \(\text{tnode::set}\), the member names \(\text{tword}\), \(\text{count}\), \(\text{left}\), and \(\text{right}\) refer to members of the object for which the function is called. Thus, in the call \(\text{n1.set("abc",\&n2,0)}\), \(\text{tword}\) refers to \(\text{n1.tword}\), and in the call \(\text{n2.set("def",0,0)}\), it refers to \(\text{n2.tword}\). The functions \(\text{strlen}, \text{perror, and strcpy}\) are not members of the class \(\text{tnode}\) and should be declared elsewhere.\(^{[100]}\) — end example]

A non-static member function may be declared const, volatile, or const volatile. These cv-qualifiers affect the type of the this pointer \((9.3.2)\). They also affect the function type \((8.3.5)\) of the member function; a member function declared const is a const member function, a member function declared volatile is a volatile member function and a member function declared const volatile is a const volatile member function. [Example:]

\[
\text{struct X} \\
\text{    void g()}\ \text{const;} \\
\text{    void h()}\ \text{const volatile;} \\
\};
\]

\(\text{X::g}\) is a const member function and \(\text{X::h}\) is a const volatile member function. — end example]

A non-static member function may be declared with a ref-qualifier \((8.3.5)\); see \(13.3.1\).  

\(^{[100]}\)See, for example, <cstring> \((21.5)\).
9.4 Static members

A non-static member function may be declared virtual (10.3) or pure virtual (10.4).

9.3.2 The this pointer

In the body of a non-static (9.3) member function, the keyword this is an 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 \( \text{const} \), the type of this is \( \text{const} \ X^* \), if the member function is declared \( \text{volatile} \), the type of this is \( \text{volatile} \ X^* \), and if the member function is declared \( \text{const} \ \text{volatile} \), the type of this is \( \text{const} \ \text{volatile} \ X^* \).

In a \( \text{const} \) member function, the object for which the function is called is accessed through a \( \text{const} \) access path; therefore, a \( \text{const} \) 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++; }
    int h() const { return a++; } // error
};

int s::f() const { return a; }
```

The \( a++ \) in the body of \( s::h() \) is ill-formed because it tries to modify (a part of) the object for which \( s::h() \) is called. This is not allowed in a \( \text{const} \) member function because \( \text{this} \) is a pointer to \( \text{const} \); that is, \( *\text{this} \) has \( \text{const} \) type. — end example]

Similarly, \( \text{volatile} \) semantics (7.1.6.1) apply in \( \text{volatile} \) member functions when accessing the object and its non-static data members.

A \( \text{cv-qualified} \) member function can be called on an object-expression (5.2.5) only if the object-expression is as \( \text{cv-qualified} \) or less-\( \text{cv-qualified} \) than the member function. [Example:

```c
void k(s& x, const s& y) {
    x.f();
    x.g();
    y.f();
    y.g(); // error
}
```

The call \( y.g() \) is ill-formed because \( y \) is \( \text{const} \) and \( s::g() \) is a non-\( \text{const} \) member function, that is, \( s::g() \) is less-\( \text{qualified} \) than the object-expression \( y \). — end example]

Constructors (12.1) and destructors (12.4) shall not be declared \( \text{const} \), \( \text{volatile} \) or \( \text{const} \ \text{volatile} \). [Note: However, these functions can be invoked to create and destroy objects with \( \text{cv-qualified} \) types, see (12.1) and (12.4). — end note]

9.4 Static members

A data or function member of a class may be declared \( \text{static} \) in a class definition, in which case it is a \( \text{static} \) member of the class.
2 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:

```c
struct process {
    static void reschedule();
};

process& g();

void f() {
    process::reschedule(); // OK: no object necessary
    g().reschedule();     // g() is called
}
```

— end example ]

3 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:

```c
int g();
struct X {
    static int g();
};
struct Y : X {
    static int i;
};
int Y::i = g(); // equivalent to Y::g();
```

— end example ]

4 If an unqualified-id (5.1) is used in the definition of a static member following the member's declarator-id, and name lookup (3.4.1) finds that the unqualified-id refers to a static member, enumerator, or nested type of the member's class (or of a base class of the member’s class), the unqualified-id is transformed into a qualified-id expression in which the nested-name-specifier names the class scope from which the member is referenced. [Note: See 5.1 for restrictions on the use of non-static data members and non-static member functions. — end note ]

5 Static members obey the usual class member access rules (clause 11). When used in the declaration of a class member, the static specifier shall only be used in the member declarations that appear within the member-specification of the class definition. [Note: it cannot be specified in member declarations that appear in namespace scope. — end note ]

9.4.1 Static member functions [class.static.mfct]

[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
9.5 Unions

by all the objects of the class.

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:

```c
class process {
    static process* run_chain;
    static process* running;
};
```

```c
process* process::running = get_main();
process* process::run_chain = running;
```

The static data member run_chain of class process is defined in global scope; the notation process::run_chain specifies that the member run_chain is a member of class process and in the scope of class process. In the static data member definition, the initializer expression refers to the static data member running of class process. — end example]

[Note: once the static data member has been defined, it exists even if no objects of its class have been created. [Example: in the example above, run_chain and running exist even if no objects of class process are created by the program. — end example] — end note]

If a static data member is of const literal type, its declaration in the class definition can specify a constant-initializer. 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 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.

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).

9.5 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 standard-layout union contains several standard-layout structs that share a common initial sequence (9.2), and if an object of this standard-layout union type contains one of the standard-layout structs, it is permitted to inspect the common initial sequence of any of 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) shall not be a member of a union, nor shall an array of such objects. If a union contains a non-static data member or a member of reference type the program is ill-formed. [Note: If any non-static data member of a union has a non-trivial default constructor (12.1), copy constructor (12.8), copy assignment operator (12.8), or destructor (12.4), the corresponding member function of the union must be user-declared or it will be implicitly deleted (8.4) for the union. — end note]

Example: Consider the following union:

```cpp
union U {
    int i;
    float f;
    std::string s;
};
```

Since std::string (21.2) declares non-trivial versions of all of the special member functions, U will have an implicitly deleted default constructor, copy constructor, copy assignment operator, and destructor. To use U, some or all of these member functions must be user-declared.

Consider an object u of a union type U having non-static data members m of type M and n of type N. If M has a non-trivial destructor and N has a non-trivial constructor (for instance, if they declare or inherit virtual functions), the active member of u can be safely switched from m to n using the destructor and placement new operator as follows:

```
u.m.~M();
new (&u.n) N;
```

— end example —

A union of the form

```cpp
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:

```cpp
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.

4 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]

5 A union-like class is a union or a class that has an anonymous union as a direct member. A union-like class \( X \) has a set of variant members. If \( X \) is a union its variant members are the non-static data members; otherwise, its variant members are the non-static data members of all anonymous unions that are members of \( X \).

9.6 Bit-fields [class.bit]

1 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 with a value greater than or equal to zero. The value of the integral constant expression may be larger than the number of bits in the object representation (3.9) of the bit-field’s type; in such cases the extra bits are used as padding bits and do not participate in the value representation (3.9) of the bit-field. Allocation of bit-fields within a class object is implementation-defined. Alignment of bit-fields is implementation-defined. Bit-fields are packed into some addressable allocation unit. [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. Bit-fields are assigned right-to-left on some machines, left-to-right on others. —end note]

2 A declaration for a bit-field that omits the `identifier` declares an unnamed bit-field. Unnamed bit-fields are not members and cannot be initialized. [Note: an unnamed bit-field is useful for padding to conform to externally-imposed layouts. —end note] As a special case, an unnamed bit-field with a width of zero specifies alignment of the next bit-field at an allocation unit boundary. Only when declaring an unnamed bit-field may the value of the `constant-expression` be equal to zero.

3 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]

4 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:
enum BOOL { FALSE=0, TRUE=1 };
struct A {
    BOOL b:1;
};
A a;
void f() {
    a.b = TRUE;
    if (a.b == TRUE) // yields true
        { /* ... */ }
}

— end example ]

9.7  Nested class declarations [class.nest]

1 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:

    int x;
    int y;

    struct enclose {
        int x;
        static int s;

        struct inner {
            void f(int i) {
                int a = sizeof(x); // OK: operand of sizeof is an unevaluated operand
                x = i; // error: assign to enclose::x
                s = i; // OK: assign to enclose::s
                ::x = i; // OK: assign to global x
                y = i; // OK: assign to global y
            }
            void g(enclose* p, int i) {
                p->x = i; // OK: assign to enclose::x
            }
        }
    };

    inner* p = 0; // error: inner not in scope

— end example ]

2 Member functions and static data members of a nested class can be defined in a namespace scope enclosing the definition of their class. [ Example:

    struct enclose {
        struct inner {

Draft
9.8 Local class declarations

A class can be declared within a function definition; such a class is called a **local** class. The name of a local class is local to its enclosing scope. The local class is in the scope of the enclosing scope, and has the same access to names outside the function as does the enclosing function. Declarations in a local class can use only type names, static variables, **extern** variables and functions, and enumerators from the enclosing scope. [Example:]

```cpp
int x;
void f() {
    static int s;
    int x;
    extern int g();

    struct local {
        int g() { return x; }  // error: x is *auto* has automatic storage duration
        int h() { return s; }  // OK
        int k() { return ::x; } // OK
        int l() { return g(); } // OK
    };
}

declarer* p = 0;  // error: local not in scope
```

--- end example ---

If class X is defined in a namespace scope, a nested class Y may be declared in class X and later defined in the definition of class X or be later defined in a namespace scope enclosing the definition of class X. [Example:]

```cpp
class E {
    class I1;  // forward declaration of nested class
    class I2;
    class I1 {};
    // definition of nested class
}
class E::I2 {};
// definition of nested class
```

--- end example ---

Like a member function, a friend function (11.4) defined within a nested class is in the lexical scope of that class; it obeys the same rules for name binding as a static member function of that class (9.4), but it has no special access rights to members of an enclosing class.

---

Draft
2 An enclosing function has no special access to members of the local class; it obeys the usual access rules (clause 11). Member functions of a local class shall be defined within their class definition, if they are defined at all.

3 If class X is a local class a nested class Y may be declared in class X and later defined in the definition of class X or be later defined in the same scope as the definition of class X. A class nested within a local class is a local class.

4 A local class shall not have static data members.

9.9 Nested type names

1 Type names obey exactly the same scope rules as other names. In particular, type names defined within a class definition cannot be used outside their class without qualification. [Example:

```c
struct X {
    typedef int I;
    class Y { /* ... */ };  
    I a;
};

I b;       // error
Y c;       // error
X::Y d;    // OK
X::I e;    // OK
```

—end example]
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:
  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:

```
struct Base {
  int a, b, c;
};

struct Derived : Base {
```
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]

[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]
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 ]

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.

![Diagram of class lattice](image)

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).

For another example,

```cpp
class V { /* ... */};
class A : virtual public V { /* ... */};
class B : virtual public V { /* ... */};
class C : public A, public B { /* ... */};
```

for an object c of class type C, a single subobject of type V is shared by every base subobject of c that has a virtual base class of type V. Given the class C defined above, an object of class C will have one subobject of class V, as shown below.
7 A class can have both virtual and non-virtual base classes of a given type.

class B { /* ... */ };
class X : virtual public B { /* ... */ };
class Y : virtual public B { /* ... */ };
class Z : public B { /* ... */ };
class AA : public X, public Y, public Z { /* ... */ };

For an object of class AA, all virtual occurrences of base class B in the class lattice of AA correspond to a single B subobject within the object of type AA, and every other occurrence of a (non-virtual) base class B in the class lattice of AA corresponds one-to-one with a distinct B subobject within the object of type AA. Given the class AA defined above, class AA has two subobjects of class B: Z’s B and the virtual B shared by X and Y, as shown below.

— end example ]

10.2 Member name lookup [class.member.lookup]

1 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).

2 The following steps define the result of name lookup for a member name f in a class scope C.

3 The lookup set for f in C, called S(f, C), consists of two component sets: the declaration set, a set of members named f; and the subobject set, a set of subobjects where declarations of these members (possibly including using-declarations) were found. In the declaration set, using-declarations are replaced by the members they designate, and type declarations (including injected-class-names) are replaced by the types they designate. S(f, C) is calculated as follows:

4 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: ]

```
struct A { int x; };           // S(x,A) = { { A::x }, [ A ] }
struct B { float x; };        // S(x,B) = { { B::x }, [ B ] }
struct C: public A, public B { }; // S(x,C) = { invalid, [ A in C, B in C ] }
struct D: public virtual C { }; // S(x,D) = S(x,C)
struct E: public virtual C { char x; }; // S(x,E) = { { E::x }, [ E ] }
struct F: public D, public E { }; // S(x,F) = S(x,E)
int main() {
    F f;
    f.x = 0;                  // OK, lookup finds E::x
}
```

S(x, F) is unambiguous because the A and B base subobjects of D are also base subobjects of E, so S(x, D) is discarded in the first merge step. — end example ]

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: ]

```
struct A {
    int f();
};

struct B {
    int f();
};

struct C : A, B {
    int f() { return A::f() + B::f(); }
};
```

Draft
9  [Note: A static member, a nested type or an enumerator defined in a base class T can unambiguously be found even if an object has more than one base class subobject of type T. Two base class subobjects share the non-static member subobjects of their common virtual base classes. —end note] [Example:

```c
struct V {
    int v;
};
struct A {
    int a;
    static int s;
    enum { e };  
};
struct B : A, virtual V {};
struct C : A, virtual V {};
struct D : B, C {};

void f(D* pd) {
    pd->v++;       // OK: only one v (virtual)
    pd->s++;       // OK: only one s (static)
    int i = pd->e;  // OK: only one e (enumerator)
    pd->a++;       // error, ambiguous: two a in D
}
```

—end example]

10  [Note: When virtual base classes are used, a hidden declaration can be reached along a path through the subobject lattice that does not pass through the hiding declaration. This is not an ambiguity. The identical use with non-virtual base classes is an ambiguity; in that case there is no unique instance of the name that hides all the others. —end note] [Example:

```c
struct V { int f(); int x; };
struct W { int g(); int y; };
struct B : virtual V, W {
    int f(); int x;
    int g(); int y;
};
struct C : virtual V, W {};
struct D : B, C { void glorp(); };
```

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
struct V { };  
struct A { };  
struct B : A, virtual V { };  
struct C : A, virtual V { };  
struct D : B, C { };  

void g() {
    D d;
    B* pb = &d;  // error, ambiguous: C's A or B's A?
    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 {
    void f(double);
};
struct I1: B1 { }
struct I2: B1 { }
struct D: I1, I2, B2 {
    using B1::f;
    using B2::f;
    void g() {
        f(); // Ambiguous conversion of this
        f(0); // Unambiguous (static)
    }
```
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), cv-qualification, and ref-qualifier (or absence of same) as \(Base::vf\) is declared, then \(Derived::vf\) is also virtual (whether or not it is so declared) and it overrides \(Base::vf\). For convenience we say that any virtual function overrides itself. 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
  c.C::f(); // calls A::f because of the using-declaration
}
```

— end example ]

Note: a virtual member function does not have to be visible to be overridden, for example,

```cpp
struct B {
  virtual void f();
};
struct D : B {
  void f(int);
}
```

---

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.
the function \texttt{f(int)} in class \texttt{D} hides the virtual function \texttt{f()} in its base class \texttt{B}; \texttt{D::f(int)} is not a virtual function. However, \texttt{f()} declared in class \texttt{D2} has the same name and the same parameter list as \texttt{B::f()}, and therefore is a virtual function that overrides the function \texttt{B::f()} even though \texttt{B::f()} is not visible in class \texttt{D2}. — \textit{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 \textit{covariant} with the classes of the functions. If a function \texttt{D::f} overrides a function \texttt{B::f}, the return types of the functions are covariant if they satisfy the following criteria:

\begin{itemize}
  \item both are pointers to classes or references to classes\textsuperscript{(102)}
  \item the class in the return type of \texttt{B::f} is the same class as the class in the return type of \texttt{D::f}, or is an unambiguous and accessible direct or indirect base class of the class in the return type of \texttt{D::f}
  \item both pointers or references have the same cv-qualification and the class type in the return type of \texttt{D::f} has the same cv-qualification as or less cv-qualification than the class type in the return type of \texttt{B::f}.
\end{itemize}

6 If the return type of \texttt{D::f} differs from the return type of \texttt{B::f}, the class type in the return type of \texttt{D::f} shall be complete at the point of declaration of \texttt{D::f} or shall be the class type \texttt{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:

\begin{verbatim}
class B { }
class D : private B { friend class Derived; }
struct Base {
  virtual void vf1();
  virtual void vf2();
  virtual void vf3();
  virtual B* vf4();
  virtual B* vf5();
  void f();
};

struct No_good : public Base {
  D* vf4(); // error: B (base class of D) inaccessible
};

class A;
struct Derived : public Base {
  void vf1(); // virtual and overrides Base::vf1()
  void vf2(int); // not virtual, hides Base::vf2()
  char vf3(); // error: invalid difference in return type only
  D* vf4(); // OK: returns pointer to derived class
\end{verbatim}

\textsuperscript{102)}Multi-level pointers to classes or references to multi-level pointers to classes are not allowed.

Draft
10.3 Virtual functions

A* vf5();         // error: returns pointer to incomplete class
void f();
);

void g() {
  Derived d;
  Base* bp = &d;
    // standard conversion:
    // Derived* to Base*
  bp->vf1();  // calls Derived::vf1()
  bp->vf2();  // calls Base::vf2()
  bp->f();   // calls Base::f() (not virtual)
  B* p = bp->vf4();  // calls Derived::pf() and converts the
                     // result to B*
  Derived* dp = &d;
  D* q = dp->vf4();  // calls Derived::pf() and does not
                     // convert the result to B*
  dp->vf2();  // ill-formed: argument mismatch
}

— end example ]

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;
    }

Draft
// 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:

```cpp
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.

```cpp
struct VB1a : virtual A { // does not declare f
};

struct Da : VB1a, VB2 {
};

void foe() {
    VB1a* vblap = new Da;
vblap->f(); // calls VB2::f
}
```

— end example ]

Explicit qualification with the scope operator (5.1) suppresses the virtual call mechanism. [ Example:
class B { public: virtual void f(); };  
class D : public B { public: void f(); };  

void D::f() { /* ... */ B::f(); }

Here, the function call in D::f really does call B::f and not D::f. —end example

A function with a deleted definition (8.4) shall not override a function that does not have a deleted definition. Likewise, a function that does not have a deleted definition shall not override a function with a deleted definition.

10.4 Abstract classes

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 { /* */ /* */ };  
class shape {  
  point center;  
  public:  
    point where() { return center; }  
    void move(point p) { center=p; draw(); }  
    virtual void rotate(int) = 0; // pure virtual  
    virtual void draw() = 0; // pure virtual  
};
```

—end example] [Note: a function declaration cannot provide both a pure-specifier and a definition —end note] [Example:

```cpp
struct C {  
  virtual void f() = 0 { }; // ill-formed  
};
```

—end example]

An abstract class shall not be used as a parameter type, as a function return type, or as the type of an explicit conversion. Pointers and references to an abstract class can be declared. [Example:

```cpp
shape x; // error: object of abstract class  
shape* p; // OK  
shape f(); // error  
void g(shape); // error  
shape& h(shape&); // OK
```
A class is abstract if it contains or inherits at least one pure virtual function for which the final overrider is pure virtual.

*Example:*

```cpp
class ab_circle : public shape {
    int radius;
public:
    void rotate(int) { }
    // ab_circle::draw() is a pure virtual
};
```

Since `shape::draw()` is a pure virtual function, `ab_circle::draw()` is a pure virtual by default. The alternative declaration,

```cpp
class circle : public shape {
    int radius;
public:
    void rotate(int) { }
    void draw();  // a definition is required somewhere
};
```

would make class `circle` nonabstract and a definition of `circle::draw()` must be provided. — *end example*

*Note:* an abstract class can be derived from a class that is not abstract, and a pure virtual function may override a virtual function which is not pure. — *end note*

Member functions can be called from a constructor (or destructor) of an abstract class; the effect of making a virtual call (10.3) to a pure virtual function directly or indirectly for the object being created (or destroyed) from such a constructor (or destructor) is undefined.
Chapter 11  Member access control  [class.access]

1 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.

2 A member of a class can also access all the names to which the class has access. A local class of a member function
   may access the same names that the member function itself may access.\(^{(03)}\)

3 Members of a class defined with the keyword class are private by default. Members of a class defined with the
   keywords struct or union are public by default. [Example:
   ```
   class X {
       int a; // X::a is private by default
   }
   
   struct S {
       int a; // S::a is public by default
   }
   ```
   — end example ]

4 Access control is applied uniformly to all names, whether the names are referred to from declarations or expressions.
   [ Note: access control applies to names nominated by friend declarations (11.4) 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,
     ```
     class A {
         class B { }; // B::BB is public by default
     }
     
     typedef B BB;
     ```
   — end example ]

\(^{(03)}\) Access permissions are thus transitive and cumulative to nested and local classes.
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.

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-specifiers 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:

```cpp
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 { 
};
```

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-specifiers must be deferred until the entire base-specifier-list has been seen. — end example]

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.

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:

```cpp
class B { 
};
template <class T> class C {
```
11.1 Access specifiers

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:

```
class X {
    int a; // X::a is private by default: class used
    public:
        int b; // X::b is public
        int c; // X::c is public
};
```

— end example ]

Any number of access specifiers is allowed and no particular order is required. [Example:

```
struct S {
    int a; // S::a is public by default: struct used
    protected:
        int b; // S::b is protected
    private:
        int c; // S::c is private
    public:
        int d; // S::d is public
};
```

— end example ]

[ Note: the effect of access control on the order of allocation of data members is described in 9.2. — end note ]

When a member is redeclared within its class definition, the access specified at its redeclaration shall be the same as at its initial declaration. [Example:

```
struct S {
    class A;
    private:
        class A { }; // error: cannot change access

    D <C<B> >* d; // access error, C::TT is protected
```

— 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.

Note: A member of a private base class might be inaccessible as an inherited member name, but accessible directly. Because of the rules on pointer conversions (4.10) and explicit casts (5.4), a conversion from a pointer to a derived class to a pointer to an inaccessible base class might be ill-formed if an implicit conversion is used, but well-formed if an explicit cast is used. For example,

---

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.
class B {
public:
    int mi; // non-static member
    static int si; // static member
};
class D : private B {
};
class DD : public D {
    void f();
};

void DD::f() {
    mi = 3; // error: mi is private in D
    si = 3; // error: si is private in D
    ::B b;
    b.mi = 3; // OK (b.mi is different from this->mi)
    b.si = 3; // OK (b.si is different from this->si)
    ::B::si = 3; // OK
    ::B* bp1 = this; // error: B is a private base class
    ::B* bp2 = (::B*)this; // OK with cast
    bp2->mi = 3; // OK: access through a pointer to B.
}

— end note ]

4 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:

    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
    }

Draft
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 ;.\textsuperscript{105}

\textsuperscript{105} Access declarations are deprecated; member using-declarations (7.3.3) provide a better means of doing the same things. In earlier versions of the

Draft
[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();
};
```

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 `df` (public), and `x` and `g` (protected). Thus the external function `ff` has access only to `c`, `z`, `z1`, `e`, and `df`. If `D` were a protected or private base class of `X`, `xf` would have the same privileges as before, but `ff` would have no access at all. —end example]

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 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
the class. A class specifies its friends, if any, by way of friend declarations. Such declarations give special access rights to the friends, but they do not make the nominated friends members of the befriending class. [Example: the following example illustrates the differences between members and friends:

```cpp
class X {
    int a;
    friend void friend_set(X*, int);
public:
    void member_set(int);
};

void friend_set(X* p, int i) { p->a = i; }
void X::member_set(int i) { a = i; }

void f() {
    X obj;
    friend_set(&obj,10);
    obj.member_set(10);
}
```

— end example ]

2 Declaring a class to be a friend implies that the names of private and protected members from the class granting friendship can be accessed in the base-specifiers and member declarations of the befriended class. [Example:

```cpp
class A {
    class B { }
    friend class X;
};

struct X : A::B { // OK: A::B accessible to friend
    A::B mx;       // OK: A::B accessible to member of friend
    class Y {
        A::B my;  // OK: A::B accessible to nested member of friend
    }
};
```

— end example ] A class shall not be defined in a friend declaration. [Example:

```cpp
class X {
    enum { a=100 };  
    friend class Y;
};

class Y {
    int v[X::a];     // OK, Y is a friend of X
};

class Z {
    int v[X::a];     // error: X::a is private
};
```
A friend declaration that does not declare a function shall have one of the following forms:

```
friend elaborated-type-specifier;
friend simple-type-specifier;
friend typename-specifier;
```

[Note: a friend declaration may be the declaration in a template-declaration (clause 14, 14.5.4). — end note] If the type specifier in a friend declaration designates a (possibly cv-qualified) class type, that class is declared as a friend; otherwise, the friend declaration is ignored. [Example:]

```
class C;
typedef C Ct;

class X1 {
  friend C;        // OK: class C is a friend
};

class X2 {
  friend Ct;       // OK: class C is a friend
  friend D;        // error: no type-name D in scope
  friend class D;   // OK: elaborated-type-specifier declares new class
};

template <typename T> class R {
  friend T;
};

R<C> rc;        // class C is a friend of R<C>
R<int> Ri;      // OK: "friend int;" is ignored
```

— 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:]

```
class Y {
  friend X::*foo(int);
  friend X::*X(char);   // constructors can be friends
  friend X::*~X();      // destructors can be friends
};
```

— end example ]

A function can be defined in a friend declaration of a class if and only if the class is a non-local class (9.8), the function name is unqualified, and the function has namespace scope. [Example:]

```
class M {
  friend void f() { }        // definition of global f, a friend of M,
};
```
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
  }
};
```

— end example ]

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:

```cpp
class X;
void a();
void f() {
  class Y;
```
extern void b();
class A {
friend class X;  // OK, but X is a local class, not ::X
friend class Y;  // OK
friend class Z;  // OK, introduces local class Z
friend void a(); // error, ::a is not considered
friend void b(); // OK
friend void c(); // error
};
X *px;  // OK, but ::X is found
Z *pz;  // error, no Z is found
}

— end example ]

11.5 Protected member access

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). 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*);
};

void fr(B* pb, D1* p1, D2* p2) {
  pb->i = 1;              // ill-formed
  p1->i = 2;              // ill-formed
  p2->i = 3;              // OK (access through a D2)
  p2->B::i = 4;           // OK (access through a D2, even though
                          // naming class is B)
  int B::* pm1_B = &B::i; // ill-formed
  int B::* pm1_B2 = &D2::i; // OK (type of &D2::i is int B::*
  B::* j = 5;             // OK (because refers to static member)
  D2::* j = 6;            // OK (because refers to static member)
}

\[106\] This additional check does not apply to other members, e.g. static data members or enumerator member constants.

Draft
11.6 Access to virtual functions

The access rules (clause 11) for a virtual function are determined by its declaration and are not affected by the rules for a function that later overrides it. [Example:

```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,
  pd->f(); // error: D::f() is private
}
```

— end example ]
in the example above) is in general not known.

11.7 Multiple access

If a name can be reached by several paths through a multiple inheritance graph, the access is that of the path that gives most access. [*Example:*

```cpp
class W { public: void f(); };  
class A : private virtual W { };  
class B : public virtual W { };  
class C : public A, public B {  
  void f() { W::f(); } // OK  
};
```

Since `W::f()` is available to `C::f()` along the public path through `B`, access is allowed. — *end example*]

11.8 Nested classes

A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules (clause 11) shall be obeyed. [*Example:*

```cpp
class E {  
  int x;  
  class B { };  

  class I {  
    B b; // OK: E::I can access E::B  
    int y;  
    void f(E* p, int i) {  
      p->x = i; // OK: E::I can access E::x  
    }  
  };

  int g(I* p) {  
    return p->y; // error: I::y is private  
  }
};
```

— *end example*]
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. [Note: The implementation will implicitly declare these member functions for a some class types 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 See 12.1, 12.4 and 12.8. —end note] 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.]

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

1 Constructors do not have names. A special declarator syntax using an optional sequence of function-specifiers (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:

```c
struct S {
    S(); // declares the constructor
};

S::S() { } // defines the constructor
```
—end example]
A constructor is used to initialize objects of its class type. Because constructors do not have names, they are never found during name lookup; however an explicit type conversion using the functional notation (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 constructor shall not be declared with a `ref-qualifier`.

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 having no parameters is implicitly declared. An implicitly-declared default constructor is an `inline public` member of its class. For a union-like class that has a variant member with a non-trivial default constructor, an implicitly-declared default constructor is defined as deleted (8.4). A default constructor is trivial if it is 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.

A 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 no `ctor-initializer` (12.6.2) and an empty `compound-statement`. If that user-written default constructor would be ill-formed, the program is ill-formed. If that user-written default constructor would satisfy the requirements of a `constexpr` constructor (7.1.5), the implicitly-defined default constructor is `constexpr`. Before the non-user-provided default constructor for a class is implicitly defined, all the non-user-provided default constructors for its base classes and its non-static data members shall have been implicitly defined. [Note: an implicitly-declared default constructor has an `exception-specification` (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), or 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.
12 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]

13 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]

14 [Note: some language constructs have special semantics when used during construction; see 12.6.2 and 12.7. — end note]

15 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]

1 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 shall 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:
    X(int);
};
```
12.2 Temporary objects

X(const X&);
~X();
};

X f(X);

void g() {
  X a(1);
  X b = f(X(2));
  a = f(a);
}

2 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 }

3 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.

4 There are two contexts in which temporaries are destroyed at a different point than the end of the full-expression. The first context is when a default constructor is called to initialize an element of an array. If the constructor has one or more default arguments, the destruction of every temporary created in a default argument expression is sequenced before the construction of the next array element, if any.

5 The second context is when a reference is bound to a temporary. The temporary to which the reference is bound or the temporary that is the complete object of a subobject to which the reference is bound persists for the lifetime of the reference except 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 obj1 is an object with the same storage duration as the temporary and created before the temporary is created the temporary shall be destroyed before obj1 is destroyed; if obj2 is an object with the same storage duration as the temporary and created after the temporary is created the temporary shall be destroyed after obj2 is destroyed. [Example:

```c++
struct S {
  S();
  S(int);
  friend S operator+(const S&, const S&);
```

Draft
the expression $C(16) + C(23)$ creates three temporaries. A first temporary $T_1$ to hold the result of the expression $C(16)$, a second temporary $T_2$ to hold the result of the expression $C(23)$, and a third temporary $T_3$ to hold the result of the addition of these two expressions. The temporary $T_3$ is then bound to the reference $cr$. It is unspecified whether $T_1$ or $T_2$ is created first. On an implementation where $T_1$ is created before $T_2$, it is guaranteed that $T_2$ is destroyed before $T_1$. The temporaries $T_1$ and $T_2$ 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 $T_3$ 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 $T_3$ is destroyed takes into account the destruction order of other objects with static storage duration. That is, because $obj1$ is constructed before $T_3$, and $T_3$ is constructed before $obj2$, it is guaranteed that $obj2$ is destroyed before $T_3$, and that $T_3$ is destroyed before $obj1$. — end example ]

12.3 Conversions [class.conv]

Type conversions of class objects can be specified by constructors and by conversion functions. These conversions are called user-defined conversions and are used for implicit type conversions (clause 4), for initialization (8.5), and for explicit type conversions (5.4, 5.2.9).

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 ]

At most one user-defined conversion (constructor or conversion function) is implicitly applied to a single value.

[Example:

        struct X {
            operator int();
        };

        struct Y {
            operator X();
        };

        Y a;
        int b = a; // error
        // a.operator X().operator int() not tried
        int c = X(a); // OK: a.operator X().operator int()

        — end example ]

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:
struct X {
  operator int();
};

struct Y : X {
  operator char();
};

void f(Y& a) {
  if (a) {
    // ill-formed:
    // X::operator int() or Y::operator char()
  }
}

--- end example

### 12.3.1 Conversion by constructor

A constructor declared without the function-specifier `explicit` 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
struct X {
  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))
}

--- end example
```

---

An explicit constructor constructs objects just like non-explicit constructors, but does so only where the direct-initialization syntax (8.5) or where casts (5.2.9, 5.4) are explicitly used. A default constructor may be an explicit constructor; such a constructor will be used to perform default-initialization or value-initialization (8.5). [Example:

```cpp
struct Z {
  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
```

---

Draft
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

\[
\text{conversion-function-id:} \\
\quad \text{operator conversion-type-id} \\
\quad \text{conversion-type-id:} \\
\quad \text{type-specifier-seq conversion-declarator_opt} \\
\quad \text{conversion-declarator:} \\
\quad \text{ptr-operator conversion-declarator_opt}
\]

specifies a conversion from \( X \) to the type specified by the \text{conversion-type-id}. Such member functions are called conversion functions. Classes, enumerations, and \text{typedef-name}s shall not be declared in the \text{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 \text{conversion-type-id}.” A conversion function is never used to convert a (possibly cv-qualified) object to the (possibly cv-qualified) same object type (or a reference to it), to a (possibly cv-qualified) base class of that type (or a reference to it), or to (possibly cv-qualified) \text{void}.

107) Even though never directly called to perform a conversion, such conversion functions can be declared and can potentially be reached through a call to a virtual conversion function in a base class.

Example:

\[
\text{struct X \{} \\
\quad \text{operator int();} \\
\text{\};}
\]

\[
\text{void f(X a) \{} \\
\quad \text{int i = int(a);} \\
\quad \text{i = (int)a;} \\
\quad \text{i = a;} \\
\text{\}}
\]

In all three cases the value assigned will be converted by \( X::\text{operator int()} \). — end example

A conversion function may be explicit (7.1.2), in which case it is only considered as a user-defined conversion for direct-initialization (8.5). Otherwise, user-defined conversions are not restricted to use in assignments and initializations.

Example:

\[
\text{class Y \{ \};} \\
\text{struct Z \{} \\
\quad \text{explicit operator Y() const;} \\
\text{\};}
\]

\[
\text{void h(Z z) \{}
\]

107) 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.
Y y1(z);    // OK: direct-initialization
Y y2 = z;   // ill-formed: copy-initialization
Y y3 = (Y)z; // OK: cast notation
}

void g(X a, X b) {
  int i = (a) ? 1+a : 0;
  int j = (a&&b) ? a+b : i;
  if (a) {
  }
}

— end example ]

3 The conversion-type-id shall not represent a function type nor an array type. The conversion-type-id in a conversion-
function-id is the longest possible sequence of conversion-declarators. [ 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 ]

4 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. A destructor shall not 
be declared with a ref-qualifier.

3 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. If the class is a union-like class that has a variant member with a non-trivial 
destructor, an implicitly-declared destructor is defined as deleted (8.4). A destructor is trivial if it is 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.

4 Otherwise, the destructor is non-trivial.

5 A 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 non-user-provided destructor for a class is implicitly defined, all the 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]

6 After executing the body of the destructor and destroying any automatic objects allocated within the body, a destructor for class X calls the destructors for X’s direct non-variant 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).

7 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.

9 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.

10 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]

11 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:
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(); // 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,

    void* operator new(std::size_t, void* p) { return p; }
struct X {
    X(int);
    ~X();
};
void f(X* p);

    void g() { // rare, specialized use:
        char* buf = new char[sizeof(X)];
        X* p = new(buf) X(222); // use buf[] and initialize
        f(p);
        p->X::~X(); // cleanup
    }

— end note ]

13 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 ]

14 [ Note: the notation for explicit call of a destructor can be used for any scalar type name (5.2.4). Allowing this makes it
possible to write code without having to know if a destructor exists for a given type. For example,
typedef int I;
I* p;
p->I::~I();

— end note ]

12.5 Free store

1 Any allocation function for a class T is a static member (even if not explicitly declared static).

2 [ Example:

    class Arena;
    struct B {
      void* operator new(std::size_t, Arena*);
    };  
    struct D1 : B {
    };

    Arena* ap;
    void foo(int i) {
      new (ap) D1;   // calls B::operator new(std::size_t, Arena*)  
      new D1[i];    // calls ::operator new[](std::size_t)  
      new D1;       // ill-formed: ::operator new(std::size_t) hidden
    }

    — end example ]

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).  

108) Otherwise, if the delete-expression is used to deallocate an object of class T or array thereof, the static and dynamic types of the object shall be identical and the deallocation function’s name is looked up in the scope of T. If this lookup fails to find the name, 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.

5 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.  

109) Any deallocation function for a class X is a static member (even if not explicitly declared static). [ Example:

108) 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.

109) 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.
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
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 ]
```

Here, storage for the non-array object of class D is deallocated by D::operator delete(), due to the virtual destructor. — end note ] [Note: virtual destructors have no effect on the deallocation function actually called when the cast-expression of a delete-expression refers to an array of objects of class type. For example,

```cpp
struct B {
    virtual ~B();
    void operator delete(void*, std::size_t);
};

struct D : B {
    void operator delete(void*);
};

void f() {
    B* bp = new D;
    delete bp;       // I: uses D::operator delete(void*)
}
```

```cpp
struct B {
    virtual ~B();
    void operator delete[](void*, std::size_t);
};

struct D : B {
    void operator delete[](void*, std::size_t);
};

void f(int i) {
    D* dp = new D[i];
    delete[] dp;      // uses D::operator delete[](void*, std::size_t)
    B* bp = new D[i];
    delete[] bp;      // undefined behavior
}
```
8 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

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

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:

```c
class complex {
    complex();
    complex(double);
    complex(double,double);
};

complex sqrt(complex,complex);

complex a(1); // initialize by a call of
    // complex(double)

complex b = a; // initialize by a copy of a

complex c = complex(1,2); // construct complex(1,2)
    // using complex(double,double)
    // copy 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 `initializers` 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-expressions in an initializer-list are paired with the aggregate members they initialize. — end note] [Example:

```c
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,

```c
struct X {
    int i;
    float f;
    complex c;
} x = { 99, 88.8, 77.7 };  
```

Here, x.i is initialized with 99, x.f is initialized with 88.8, and complex::complex(double) is called for the initialization of x.c. — end example] [Note: braces can be elided in the initializer-list for any aggregate, even if the aggregate has members of a class type with user-defined type conversions; see 8.5.1. — end note]

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  
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  
```

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 { };  
```

Draft
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:

struct C {
    C( int ) { }  // 1: non-delegating constructor
    C(): C(42) { }  // 2: delegates to 1
    C( char c ) : C(42.0) { }  // 3: ill-formed due to recursion with 4
    C( double d ) : C('a') { }  // 4: ill-formed due to recursion with 3
};

— 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:

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.

3 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:

struct B1 { B1(int); /* ... */ };  
struct B2 { B2(int); /* ... */ };  
struct D : B1, B2 {
    D(int);
    B1 b;
    const int c;
};

Draft
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 `non-variant` data member of (possibly cv-qualified) class type (or array thereof) or a base class, and the entity class is a non-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-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) 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 `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]

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:
A();
A(int);
}

struct B : virtual V {
B();
B(int);
};

struct C : A, B, virtual V {
C();
C(int);
};

A::A(int i) : V(i) {
/
∗
...
∗
/

B::B(int i) {
/
∗
...
∗
/

C::C(int i) {
/
∗
...
∗
/

V v(1);  // use V(int)
A a(2);  // use V(int)
B b(3);  // use V()
C c(4);  // use V()

— end example ]

Names in the expression-list of a mem-initializer are evaluated in the scope of the constructor for which the mem-initializer is specified. [Example:

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-initializers for base classes have completed, the result of the operation is undefined. [Example:

class A {
public:
  A(int);
};
class B : public A {
    int j;
public:
    int f();
    B() : A(f()), // undefined: calls member function
        // but base A not yet initialized
        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
           // but base C not yet initialized
           i(f()) { } // well-defined: bases are all initialized
};

— end example ]

9 [ 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 ]

10 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:

    template<class... Mixins>
    class X : public Mixins... {
        public:
        X(const Mixins&... mixins) : Mixins(mixins)... { }
    };

— end example ]

12.7 Construction and destruction [class ctor]

For an object of non-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:

    struct X { int i; };
    struct Y : X { Y(); }; // non-trivial
    struct A { int a; };
    struct B : public A { int j; Y y; }; // non-trivial

Draft
extern B bobj;
B* pb = &bobj; // OK
int* p1 = &bobj.a; // undefined, refers to base class member
int* p2 = &bobj.y.i; // undefined, refers to member's member
A* pa = &bobj; // undefined, upcast to a base class type
B bobj; // definition of bobj
extern X xobj;
int* p3 = &xobj.i; // OK, X is a trivial class
X xobj;

For another example,
struct W { int j; };
struct X : public virtual W { };
struct Y {
    int *p;
    X x;
    Y() : p(&x.j) { // undefined, x is not yet constructed
    }
};

— end example ]

To explicitly or implicitly convert a pointer (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:

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
    }
};

Draft
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
struct V {
    virtual void f();
    virtual void g();
};

struct A : virtual V {
    virtual void f();
};

struct B : virtual V {
    virtual void g();
    B(V*, A*);
};

struct D : A, B {
    virtual void f();
    virtual void g();
    D() : B((A*)this, this) { }
};

B::B(V* v, A* a) {
    f();  // calls V::f, not A::f
    g();  // calls B::g, not D::g
    v->g();  // v is base of B, the call is well-defined, calls B::g
    a->f();  // undefined behavior; a's type not a base of B
}
```

—end example

The typeid operator (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.

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:

```cpp
struct V {
    virtual void f();
};

struct A : virtual V { }

struct B : virtual V {
    B(V*, A*);
};

struct D : A, B {
    D() : B((A*)this, this) {} };

B::B(V* v, A* a) {
    typeid(*this); // type_info for B
    typeid(*v); // well-defined: *v has type V, a base of B
    // yields type_info for B
    typeid(*a); // undefined behavior: type A not a base of B
    dynamic_cast<B*>(v); // well-defined: v of type V*, V base of B
    // results in B*
    dynamic_cast<B*>(a); // undefined behavior;
    // a has type A*, A not a base of B
}
```

— end example ]

## 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).[10] [Example: X::X(const X&) and X::X(X&, int=1) are copy constructors.

```cpp
struct X {
    X(int);
    X(const X&, int = 1);
};
```

[10] 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
12.8 Copying class objects

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:
struct X {
    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:
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]

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:
struct S {
    template<typename T> S(T);
};
S f();

void g() {
    S a( f() );    // does not instantiate member template
}

— end example ]

If the class definition does not explicitly declare a copy constructor, one is declared implicitly. If the class is a union-like class that has a variant member with a non-trivial copy constructor, an implicitly-declared copy constructor is defined as deleted (8.4). Thus, for the class definition
struct X {
    X(const X&, int);
};
a copy constructor is implicitly-declared. If the user-declared constructor is later defined as
X::X(const X& x, int i =0) { /* ... */

Draft
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

\[ X::X(const\ X&) \]

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\&. \(^{(11)}\)

Otherwise, the implicitly declared copy constructor will have the form

\[ X::X(X&) \]

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 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 A 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. \(^{(12)}\) [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 non-user-provided copy constructor for a class is implicitly defined, all 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;

\(^{(11)}\) This implies that the reference parameter of the implicitly-declared copy constructor cannot bind to a volatile lvalue; see C.1.8.

\(^{(12)}\) See 8.5 for more details on direct and copy initialization.
12.8 Copying class objects

Virtual base class subobjects shall be copied only once by the implicitly-defined copy constructor (see 12.6.2).

A user-declared copy assignment operator X::operator= is a non-static non-template member function of class X with exactly one parameter of type X, X&, const X&, volatile X& or const volatile X&.\(^{[13]}\) \[Note: an overloaded assignment operator must be declared to have only one parameter; see 13.5.3. \] \[Note: more than one form of copy assignment operator may be declared for a class. \] \[end note\] \[Note: if a class X only has a copy assignment operator with a parameter of type X&, an expression of type const X cannot be assigned to an object of type X. \] \[Example:\]

```c
struct X {
    X();
    X& operator=(X&);
};
const X cx;
X x;
void f() {
    x = cx; // error: X::operator=(X&) cannot assign cx into x
}
```

\[end example\] \[end note\]

If the class definition does not explicitly declare a copy assignment operator, one is declared implicitly. If the class is a union-like class that has a variant member with a non-trivial copy assignment operator, an implicitly-declared copy assignment operator is defined as deleted (8.4). 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.\(^{[14]}\)

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

\(^{[13]}\) 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.

\(^{[14]}\) This implies that the reference parameter of the implicitly-declared copy assignment operator cannot bind to a volatile lvalue; see C.1.8.

Draft
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.

11 A copy assignment operator for class X is *trivial* if it is 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*.

12 A 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:

- 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 non-user-provided copy assignment operator for a class is implicitly defined, all 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]

13 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:

```
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 \texttt{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
— in a \texttt{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 \texttt{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 \texttt{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 \texttt{exception-declaration}.

\begin{quote}
\texttt{Example:}
\begin{verbatim}
class Thing {
  public:
    Thing();
    ~Thing();
    Thing(const Thing&);
  
  Thing f() {
    Thing t;
    return t;
  }

  Thing t2 = f();
\end{verbatim}
\end{quote}

Here the criteria for elision can be combined to eliminate two calls to the copy constructor of class \texttt{Thing}: the copying of the local automatic object \texttt{t} into the temporary object for the return value of function \texttt{f()} and the copying of that temporary object into object \texttt{t2}. Effectively, the construction of the local object \texttt{t} can be viewed as directly initializing the global object \texttt{t2}, and that object’s destruction will occur at program exit. —end example]
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:

```cpp
class Thing {
    public:
        Thing();
        ~Thing();
        Thing(Thing&&);
    private:
        Thing(const Thing&);
};

Thing f(bool b) {
    Thing t;
    if (b)
        throw t;           // OK: Thing(Thing&&) used (or elided) to throw t
    return t;              // OK: Thing(Thing&&) used (or elided) to return t
}

Thing t2 = f(false);    // OK: Thing(Thing&&) used (or elided) to construct of t2
```

—end example

12.9 Inheriting Constructors [class.inhctor]

A using-declaration (7.3.3) that names a constructor implicitly declares a set of inheriting constructors. The candidate set of inherited constructors from the class X named in the using-declaration consists of actual constructors and notional constructors that result from the transformation of defaulted parameters as follows:

1. all non-template constructors of X, and
2. for each non-template constructor of X that has at least one parameter with a default argument, the set of constructors that results from omitting any ellipsis parameter specification and successively omitting parameters with a default argument from the end of the parameter-type-list, and
3. all constructor templates of X, and
4. for each constructor template of X that has at least one parameter with a default argument, the set of constructor templates that results from omitting any ellipsis parameter specification and successively omitting parameters with a default argument from the end of the parameter-type-list.

The constructor characteristics of a constructor or constructor template are

1. the template parameter list (14.1), if any.
For each non-template constructor in the candidate set of inherited constructors other than a constructor having no parameters or a copy constructor having a single parameter, a constructor is implicitly declared with the same constructor characteristics unless there is a user-declared constructor with the same signature in the class where the using-declaration appears. Similarly, for each constructor template in the candidate set of inherited constructors, a constructor template is implicitly declared with the same constructor characteristics unless there is an equivalent user-declared constructor template (14.5.6.1) in the class where the using-declaration appears. [Note: Default arguments are not inherited. — end note]

A constructor so declared has the same access as the corresponding constructor in X. It is deleted if the corresponding constructor in X is deleted (8.4).

[Note: Default and copy constructors may be implicitly declared as specified in 12.1 and 12.8. — end note]

Example:

```cpp
struct B1 {
    B1(int);
};

struct B2 {
    B2(int = 13, int = 42);
};

struct D1 : B1 {
    using B1::B1;
};

struct D2 : B2 {
    using B2::B2;
};
```

The candidate set of inherited constructors in D1 for B1 is

- `B1(const B1&)`
- `B1(int)`

The set of constructors present in D1 is

- `D1()`, implicitly declared default constructor, ill-formed if used
- `D1(const D1&)`, implicitly declared copy constructor, not inherited
- `D1(int)`, implicitly declared inheriting constructor

The candidate set of inherited constructors in D2 for B2 is

Draft
— B2(const B2&)
— B2(int = 13, int = 42)
— B2(int = 13)
— B2()

The set of constructors present in D2 is
— D2(), implicitly declared default constructor, not inherited
— D2(const D2&), implicitly declared copy constructor, not inherited
— D2(int, int), implicitly declared inheriting constructor
— D2(int), implicitly declared inheriting constructor

[Note: If two using-declarations declare inheriting constructors with the same signatures, the program is ill-formed (9.2, 13.1), because an implicitly-declared constructor introduced by the first using-declaration is not a user-declared constructor and thus does not preclude another declaration of a constructor with the same signature by a subsequent using-declaration. [Example:]

```cpp
struct B1 {
    B1(int);
};

struct B2 {
    B2(int);
};

struct D1 : B1, B2 {
    using B1::B1;
    using B2::B2;
    // ill-formed: attempts to declare D1(int) twice
}

struct D2 : B1, B2 {
    using B1::B1;
    using B2::B2;
    D2(int);  // OK: user declaration supersedes both implicit declarations
}
```

[end example] — end note]

An inheriting constructor for a class is implicitly defined when it is used (3.2) to create an object of its class type (1.8). An implicitly-defined inheriting constructor performs the set of initializations of the class that would be performed by a user-written inline constructor for that class with a mem-initializer-list whose only mem-initializer has a mem-initializer-id that names the base class named in the nested-name-specifier of the using-declaration and an expression-list as specified below, and where the compound-statement in its function body is empty (12.6.2). If that user-written constructor would be ill-formed, the program is ill-formed. Each expression in the expression-list is of the form static_cast<T&&>(p), where p is the name of the corresponding constructor parameter and T is the declared type of p.
Example:

```cpp
template< class T >
struct D : T {
   using T::T;  // declares all constructors from class T
   D() { std::clog << "Destroying wrapper" << std::endl; }
};
```

Class template D wraps any class and forwards all of its constructors, while writing a message to the standard log whenever an object of class D is destroyed. —end example—
Chapter 13  Overloading

1 When two or more different declarations are specified for a single name in the same scope, that name is said to be overloaded. 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.

2 When an overloaded function name is used in a call, which overloaded function declaration is being referenced is determined by comparing the types of the arguments at the point of use with the types of the parameters in the overloaded declarations that are visible at the point of use. This function selection process is called overload resolution and is defined in 13.3. [Example:

```cpp
double abs(double);
int abs(int);

abs(1);  // calls abs(int);
abs(1.0);  // calls abs(double);
```
— end example ]

13.1 Overloadable declarations

1 Not all function declarations can be overloaded. Those that cannot be overloaded are specified here. A program is ill-formed if it contains two such non-overloadable declarations in the same scope. [Note: this restriction applies to explicit declarations in a scope, and between such declarations and declarations made through a using-declaration (7.3.3). It does not apply to sets of functions fabricated as a result of name lookup (e.g., because of using-directives) or overload resolution (e.g., for operator functions). — end note ]

2 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 ]

— Member function declarations with the same name and the same parameter-type-list as well as member function
template declarations with the same name, the same parameter-type-list, and the same template parameter lists,
cannot be overloaded if any of them, but not all, have a ref-qualifier (8.3.5). [ Example:

class Y {
  void h() &; 
  void h() const &; // OK
  void h() &&; // OK, all declarations have a ref-qualifier
  void i() &;
  void i() const; // ill-formed, prior declaration of i
};

— end example ]

3 [ 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) { /* ... */ }  
  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 declara-
tions. [ 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:

```c
int f(char*);
int f(char[]); // same as f(char*);
int f(char[7]); // same as f(char*);
int f(char[9]); // same as f(char*);

int g(char(*)[10]);
int g(char[5][10]); // same as g(char(*)[10]);
int g(char[7][10]); // same as g(char(*)[10]);
int g(char(*)[20]); // different from g(char(*)[10]);
```

--- end example

Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent. That is, the function type is adjusted to become a pointer to function type (8.3.5).

Example:

```c
void h(int());
void h(int (*)()); // redeclaration of h(int())
void h(int x()); { } // definition of h(int())
void h(int (**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:

```c
typedef const int cInt;

int f (int);
int f (const int); // redeclaration of f(int)
int f (int) { ... } // definition of f(int)
int f (cInt) { ... } // error: redefinition of f(int)
```

--- end example

Only the const and volatile type-specifiers at the outermost level of the parameter type specification are ignored in this fashion; const and volatile type-specifiers buried within a parameter type specification are significant and can be used to distinguish overloaded function declarations.\[^{116}\] 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:

\[^{116}\] 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:

```c
struct B {
    int f(int);
};

struct D : B {
    int f(char*);
};
```

Here D::f(char*) hides B::f(int) rather than overloading it.

```c
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:

```c
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
}
```

```c
void caller () {
    extern void callee(int, int);
    {
        extern void callee(int); // hides callee(int, int)
    }
```
Different versions of an overloaded member function can be given different access rules. Example:

```cpp
class buffer {
    private:
        char* p;
        int size;
    protected:
        buffer(int s, char* store) { size = s; p = store; }
    public:
        buffer(int s) { p = new char[size = s]; }
};
```

— end example —

13.3 Overload resolution  

Overload resolution is a mechanism for selecting the best function to call given a list of expressions that are to be the arguments of the call and a set of candidate functions that can be called based on the context of the call. The selection criteria for the best function are the number of arguments, how well the arguments match the parameter-type-list of the candidate function, how well (for non-static member functions) the object matches the 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 —

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:
13.3 Overload resolution

— 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).

— 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

— “lvalue reference to cv X” for functions declared without a ref-qualifier or with the & ref-qualifier

— “rvalue reference to cv X” for functions declared with the && ref-qualifier

where X is the class of which the function is a member and cv is the cv-qualification on the member function declaration.

[ Example: for a const member function of class X, the extra parameter is assumed to have type “reference to const X”.
— end example ] For conversion functions, the function is considered to be a member of the class of the 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.

Draft
For non-static member functions declared without a `ref-qualifier`, an additional rule applies:

— even if the implicit object parameter is not `const`-qualified, an rvalue 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]

6 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 ]

7 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

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.

2 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.

3 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

---

[117] The process of argument deduction fully determines the parameter types of the function template specializations, i.e., the parameters of function template specializations contain no template parameter types. Therefore the function template specializations can be treated as normal (non-template) functions for the remainder of overload resolution.
to 13.3.1.1.1 is a non-static member function, the program is ill-formed. \(^{118}\) (The resolution of &F in other contexts is described in 13.4.)

### 13.3.1.1 Call to named function

Of interest in 13.3.1.1.1 are only those function calls in which the postfix-expression ultimately contains a name that denotes one or more functions that might be called. Such a postfix-expression, perhaps nested arbitrarily deep in parentheses, has one of the following forms:

\[
\text{postfix-expression:} \\
\quad \text{postfix-expression . id-expression} \\
\quad \text{postfix-expression } \rightarrow \text{id-expression} \\
\quad \text{primary-expression}
\]

These represent two syntactic subcategories of function calls: qualified function calls and unqualified function calls.

1. In qualified function calls, the name to be resolved is an id-expression and is preceded by an \(\rightarrow\) or \(\cdot\) operator. Since the construct \(A \rightarrow B\) is generally equivalent to \((\ast A) . B\), 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 \(\cdot\) operator. Furthermore, clause 13 assumes that the postfix-expression that is the left operand of the \(\cdot\) operator has type “\(cv\ T\)" where \(T\) denotes a class. \(^{119}\) 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 \(\cdot\) operator in the normalized member function call as the implied object argument (13.3.1).

2. In unqualified function calls, the name is not qualified by an \(\rightarrow\) or \(\cdot\) 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 \(\text{this}\) (9.3.2) is in scope and refers to class \(T\), or a derived class of \(T\), then the implied object argument is \((\ast \text{this})\). If the keyword \(\text{this}\) is not in scope or refers to another class, then a contrived object of type \(T\) becomes the implied object argument. \(^{120}\) 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 “\(cv\ T\)" and 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 \((E) . \text{operator()}\).

In addition, for each non-explicit conversion function declared in \(T\) of the form

\[
\text{operator conversion-type-id () cv-qualifier ;}
\]

\(^{118}\) 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.

\(^{119}\) Note that cv-qualifiers on the type of objects are significant in overload resolution for both lvalue and class rvalue objects.

\(^{120}\) 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.
where \textit{cv-qualifier} is the same cv-qualification as, or a greater cv-qualification than, \textit{cv}, and where \textit{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 \textit{surrogate call function} with the unique name \textit{call-function} and having the form

\[
\text{R \ call-function ( \ conversion-type-id \ F, P1 \ a1, \ldots, \ Pn \ an)} \{ \text{return F (a1,\ldots, an); } \}
\]

is also considered as a candidate function. Similarly, surrogate call functions are added to the set of candidate functions for each non-explicit conversion function declared in a base class of \textit{T} provided the function is not hidden within \textit{T} by another intervening declaration\footnote{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.}.

3 If such a surrogate call function is selected by overload resolution, the corresponding conversion function will be called to convert \textit{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 (\textit{E}). \textit{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:

\begin{verbatim}
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{verbatim}

--- 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. \textit{Note:} because \ldots, \*, 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:

\begin{verbatim}
struct String {
    String (const String&);
    String (char*);
    operator char* ();
}
\end{verbatim}

--- end note \[Example:
};
String operator + (const String&, const String&);

void f(void) {
    char* p = "one" + "two";  // ill-formed because neither
    // operand has user-defined type
    int I = 1 + 1;  // Always evaluates to 2 even if
    // user-defined types exist which
    // would perform the operation.
}

— end example ]

If either operand has a type that is a class or an enumeration, a user-defined operator function might be declared that implements this operator or a user-defined conversion can be necessary to convert the operand to a type that is appropriate for a built-in operator. In this case, overload resolution is used to determine which operator function or built-in operator is to be invoked to implement the operator. Therefore, the operator notation is first transformed to the equivalent function-call notation as summarized in Table 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 13.3.3.122) [Example:

```c
struct A {
    operator int();
};
A operator+(const A&, const A&);
void m() {
    A a, b;
    a + b; // operator+(a,b) chosen over int(a) + int(b)
}
```

— end example ]

7 If a built-in candidate is selected by overload resolution, the operands 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.123)

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.

10 [ Note: the lookup rules for operators in expressions are different than the lookup rules for operator function names in a function call, as shown in the following example:

```c
struct A { }
void operator + (A, A);

struct B {
    void operator + (B);
    void f ();
```

122) If the set of candidate functions is empty, overload resolution is unsuccessful.
123) 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

};
A a;

void B::f() {
    operator+ (a,a); // error: global operator hidden by member
    a + a; // OK: calls global operator+
}

— end note ]

13.3.1.3 Initialization by constructor [over.match.ctor]

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]

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 non-explicit 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.

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]

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 non-explicit conversion functions that are not hidden within S and yield type T or a type that can be converted to type T via a standard conversion sequence

Draft
(13.3.3.1.1) are candidate functions. For direct-initialization, those explicit conversion functions that are not hidden within \( S \) and yield type \( T \) or a type that can be converted to type \( T \) with a qualification conversion (4.4) are also candidate functions. Conversion functions that return a cv-qualified type are considered to yield the cv-unqualified version of that type for this process of selecting candidate functions. Conversion functions that return “reference to \( \text{cv2} \ X \)” return lvalues or rvalues, depending on the type of reference, of type “\( \text{cv2} \ X \)” and are therefore considered to yield \( X \) for this process of selecting candidate functions.

2 The argument list has one argument, which is the initializer expression. [Note: this argument will be compared against the implicit object parameter of the conversion functions. — end note]

13.3.1.6 Initialization by conversion function for direct reference binding

1 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 “\( \text{cv1} \ T \)” is the underlying type of the reference being initialized, and “\( \text{cv} \ S \)” 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, except that for copy-initialization, only the non-explicit conversion functions are considered. Those that are not hidden within \( S \) and yield type “lvalue reference to \( \text{cv2} \ T2 \)”, where “\( \text{cv1} \ T \)” is reference-compatible (8.5.3) with “\( \text{cv2} \ T2 \)”, are candidate functions.

2 The argument list has one argument, which is the initializer expression. [Note: this argument will be compared against the implicit object parameter of the conversion functions. — end note]

13.3.2 Viable functions

1 From the set of candidate functions constructed for a given context (13.3.1), a set of viable functions is chosen, from which the best function will be selected by comparing argument conversion sequences for the best fit (13.3.3). The selection of viable functions considers relationships between arguments and function parameters other than the ranking of conversion sequences.

2 First, to be a viable function, a candidate function shall have enough parameters to agree in number with the arguments in the list.

- 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).\(^\text{124}\) For the purposes of overload resolution, the parameter list is truncated on the right, so that there are exactly \( m \) parameters.

3 Second, for \( F \) to be a viable function, there shall exist for each argument an implicit conversion sequence (13.3.3.1) that converts that argument to the corresponding parameter of \( F \). If the parameter has reference type, the implicit conversion sequence includes the operation of binding the reference, and the fact that a reference to non-\( \text{const} \) cannot be bound to an rvalue can affect the viability of the function (see 13.3.3.1.4).

\(^\text{124}\) According to 8.3.6, parameters following the \((m+1)\)-st parameter must also have default arguments.
13.3 Overload resolution

13.3.3 Best Viable Function

Define \( ICS_i(F) \) as follows:

- if \( F \) is a static member function, \( ICS_i(F) \) is defined such that \( ICS_i(F) \) is neither better nor worse than \( ICS_i(G) \) for any function \( G \), and, symmetrically, \( ICS_i(G) \) is neither better nor worse than \( ICS_i(F) \)\(^{125}\); otherwise,

- let \( ICS_i(F) \) 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 \( i \), \( ICS_i(F_1) \) is not a worse conversion sequence than \( ICS_i(F_2) \), and then

- for some argument \( j \), \( ICS_j(F_1) \) is a better conversion sequence than \( ICS_j(F_2) \), 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
```

\[-end example\]

2 If there is exactly one viable function that is a better function than all other viable functions, then it is the one selected by overload resolution; otherwise the call is ill-formed\(^ {126}\).

\[Example:\]

```c
void Fcn(const int*, short);
void Fcn(int*, int);
```

\(^{125}\) 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.

\(^{126}\) 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.

Draft
int i;
short s = 0;

void f() {
    Fcn(&i, s); // is ambiguous because
    // &i → int* is better than &i → const int*
    // but s → short is also better than s → int
    Fcn(&i, 1L); // calls Fcn(int*, int), because
    // &i → int* is better than &i → const int*
    // and 1L → short and 1L → int are indistinguishable
    Fcn(&i, 'c'); // calls Fcn(int*, int), because
    // &i → int* is better than &i → const int*
    // and c → int is better than c → short
}

— end example }

If the best viable function resolves to a function for which multiple declarations were found, and if at least two
of these declarations — or the declarations they refer to in the case of using-declarations — specify a default
argument that made the function viable, the program is ill-formed. [Example:

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,
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. The implicit conversion sequence for that case is the identity sequence; it contains no “conversion” from const A to A. — end example] When the parameter has a class type and the argument expression has the same type, the implicit conversion sequence is an identity conversion. When the parameter has a class type and the argument expression has a derived class type, the implicit conversion sequence is a derived-to-base Conversion from the derived class to the base class. [Note: there is no such standard conversion; this derived-to-base Conversion exists only in the description of implicit conversion sequences. — end note] A derived-to-base Conversion has Conversion rank (13.3.3.1.1).

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(27). If a function that uses the ambiguous conversion sequence is selected as the best viable

(27) 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.

```c++
class 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]

[Note: As described in clause 4, a standard conversion sequence is either the Identity conversion by itself (that is, no conversion) or consists of one to three conversions from the other four categories. At most one conversion from each category is allowed in a single standard conversion sequence. If there are two or more conversions in the sequence, the conversions are applied in the canonical order: Lvalue Transformation, Promotion or Conversion, Qualification Adjustment. — end note]

Each conversion in Table 11 also has an associated rank (Exact Match, Promotion, or Conversion). These are used to rank standard conversion sequences (13.3.3.2). The rank of a conversion sequence is determined by considering the rank of each conversion in the sequence and the rank of any reference binding (13.3.3.1.4). If any of those has Conversion rank, the sequence has Conversion rank; otherwise, if any of those has Promotion rank, the sequence has Promotion rank; otherwise, the sequence has Exact Match rank.

### 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).

```cpp
class A { A (B&); };  // ambiguous because b → C via constructor and
class B { operator A () {}; };  // b → k via constructor or conversion function.
class C { C (B&); };  
void f(A) {}  
void f(C) {}  
B b;  
f(b);  
// 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).
```
3 If the user-defined conversion is specified by a specialization of a conversion function template, the second standard conversion sequence shall have exact match rank.

4 A conversion of an expression of class type to the same class type is given Exact Match rank, and a conversion of an expression of class type to a base class of that type is given Conversion rank, in spite of the fact that a 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:

```c
struct A {}
struct B : public A {} b;
int f(A&);
int f(B&);
int i = f(b); // calls f(B&), an exact match, rather than
              // f(A&), a conversion
```

—end example] If the parameter binds directly to the result of applying a conversion function to the argument expression, the implicit conversion sequence is a user-defined conversion sequence (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.

2 When a parameter of reference type is not bound directly to an argument expression, the conversion sequence is the one required to convert the argument expression to the underlying type of the reference according to 13.3.3.1. Concep-
tually, 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-\texttt{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-\texttt{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 \texttt{int}” parameter can be a viable candidate even if the corresponding argument is an \texttt{int} bit-field. The formation of implicit conversion sequences treats the \texttt{int} bit-field as an \texttt{int} lvalue and finds an exact match with the parameter. If the function is selected by overload resolution, the call will nonetheless be ill-formed because of the prohibition on binding a non-\texttt{const} lvalue reference to a bit-field (8.5.3). — end example ]

The binding of a reference to an expression that is \textit{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 \textit{better conversion sequence} and \textit{better conversion}. If an implicit conversion sequence \texttt{S1} is defined by these rules to be a better conversion sequence than \texttt{S2}, then it is also the case that \texttt{S2} is a \textit{worse conversion sequence} than \texttt{S1}. If conversion sequence \texttt{S1} is neither better than nor worse than conversion sequence \texttt{S2}, \texttt{S1} and \texttt{S2} are said to be \textit{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 \texttt{S1} is a better conversion sequence than standard conversion sequence \texttt{S2} if

— \texttt{S1} is a proper subsequence of \texttt{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 \texttt{S1} is better than the rank of \texttt{S2}, or \texttt{S1} and \texttt{S2} have the same rank and are distinguishable by the rules in the paragraph below, or, if not that,

— \texttt{S1} and \texttt{S2} differ only in their qualification conversion and yield similar types \texttt{T1} and \texttt{T2} (4.4), respectively, and the cv-qualification signature of type \texttt{T1} is a proper subset of the cv-qualification signature of type \texttt{T2}, and \texttt{S1} is not the deprecated string literal array-to-pointer conversion (4.2). [Example:

\begin{verbatim}
int f(const int *);
int f(int *);
\end{verbatim}

Draft
13.3 Overload resolution

```c
int i;
int j = f(&i);  // calls f(int*)

— end example } or, if not that,

— S1 and S2 are reference bindings (8.5.3) and neither refers to an implicit object parameter of a non-static member function declared without a ref-qualifier, and 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:

```c
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&&)

struct A {
    A& operator<<(int);
    void p() &;
    void p() &&;
};
A& operator<<(A&&, char);
A() << 1;  // calls A::operator<<(int)
A() << 'c';  // calls operator<<(A&&, char)
A a;
a << 1;  // calls A::operator<<(int)
a << 'c';  // calls operator<<(A&&, char)
A().p();  // calls A::p()&&
a.p();  // calls A::p&

— 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:

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

int i;
int j = f(i);  // calls f(int &)
int k = g(i);  // ambiguous

struct X {
    void f() const;
    void f();
};
```

Draft
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. [ Example:

struct A {
    operator short();
} a;
int f(int);
int f(float);
int i = f(a);  // calls f(int), because short → int is
              // better than short → float.

— end example ]

4 Standard conversion sequences are ordered by their ranks: an Exact Match is a better conversion than a Promotion, which is a better conversion than a Conversion. Two conversion sequences with the same rank are indistinguishable unless one of the following rules applies:

— 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:

struct A {};
struct B : public A {};
struct C : public B {};
C *pc;
int f(A *);
int f(B *);
int i = f(pc);  // calls f(B*)

— end example ]

— 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*,

Draft
— binding of an expression of type B to a reference of type A& is better than binding an expression of type C to a reference of type 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.

[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.

Draft
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);
int (*pf)(double) = &f; // selects f(double)
int (*pf)(int) = &f; // selects f(int)
int (*pf)(...) = &f; // error: type mismatch
int (&rf)(int) = f; // selects f(int)
int (&rfd)(double) = f; // selects f(double)
void g() {
    (int (*)(int))&f; // cast expression as selector
}
```

The initialization of pf.e is ill-formed because no \( f() \) with type \( \text{int}(\ldots) \) has been declared, and not because of any ambiguity. For another example,

```c
struct X {
    int f(int);
    static int f(long);
};
int (X::*p1)(int) = &X::f; // OK
int (*p2)(int) = &X::f; // error: mismatch
int (X::*p3)(long) = &X::f; // OK
int (X::*p4)(long) = &X::f; // error: mismatch
int (X::*p5)(int) = &X::f; // error: wrong syntax for
// pointer to member
int (*p6)(long) = &X::f; // OK
```

— end example ]

[Note: if \( f() \) and \( g() \) are both overloaded functions, the cross product of possibilities must be considered to resolve \( f(g) \), or the equivalent expression \( f(g) \). — end note ]

[Note: 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)(D*) = &f; // error

void g(D*);
void (*p2)(B*) = &g; // error
```

— end note ]

13.5 Overloaded operators

A function declaration having one of the following operator-function-ids as its name declares an operator function. A function template declaration having one of the following operator-function-ids as its name declares an operator function.
13.5 Overloaded operators

function template. A specialization of an operator function template is also an operator function. An operator function is said to implement the operator named in its operator-function-id.

operator-function-id:

operator operator

operator: one of

new delete new[] delete[]
+
-

* / 
%

^& |

~
!

>= <=

+= -= *= /= %= ^= &= |= <<= >> >>=

operator-function-id:

new delete new[] delete[]
+
-

* / 
%

^& |

~
!

>= <=

+= -= *= /= %= ^= &= |= <<= >> >>=

( ) []

[ 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

+ - * &

can be overloaded.

3 The following operators cannot be overloaded:

.* :: ?:

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:

    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 ≡ 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.

Draft
An operator function cannot have default arguments (8.3.6), except where explicitly stated below. Operator functions cannot have more or fewer parameters than the number required for the corresponding operator, as described in the rest of this subclause.

Operators not mentioned explicitly in subclauses 13.5.3 through 13.5.7 act as ordinary unary and binary operators obeying the rules of 13.5.1 or 13.5.2.

13.5.1 Unary operators

A prefix unary operator shall be implemented by a non-static member function (9.3) with no parameters or a non-member function with one parameter. Thus, for any prefix unary operator @, @x can be interpreted as either x.\texttt{operator}@() or \texttt{operator}@\texttt{(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. \[\text{Note: consequently, a unary operator can hide a binary operator from an enclosing scope, and vice versa. \textit{— end note}}\]

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 \texttt{x.\texttt{operator}@\texttt{(y)}} or \texttt{operator}@\texttt{(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 \texttt{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. \[\text{Note: for a derived class } D \text{ with a base class } B \text{ for which a virtual copy assignment has been declared, the copy assignment operator in } D \text{ does not override } B\text{'s virtual copy assignment operator.}\text{[\textit{Example:}}

\begin{verbatim}
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
}
\end{verbatim}
13.5 Overloaded operators

//D::operator=(const D&)
}

— end example ] — end note ]

13.5.4 Function call

operator() shall be a non-static member function with an arbitrary number of parameters. It can have default arguments. It implements the function call syntax

```
postfix-expression ( expression-list_opt )
```

where the `postfix-expression` evaluates to a class object and the possibly empty `expression-list` matches the parameter list of an `operator()` member function of the class. Thus, a call `x(arg1,...)` is interpreted as `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

```
postfix-expression [ expression ]
```

Thus, a subscripting expression `x[y]` is interpreted as `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 `->`.

```
postfix-expression -> template_opt id-expression
postfix-expression -> pseudo-destructor-name
```

An expression `x->m` is interpreted as `(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 (13.3.3).

13.5.7 Increment and decrement

The user-defined function called `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 `int`) or a non-member function with two parameters (the second of which shall be of type `int`), it defines the postfix increment operator `++` for objects of that type. When the postfix increment is called as a result of using the `++` operator, the `int` argument will have value zero.\[128\] [ Example:

```c
struct X {
    X& operator++(); // prefix ++a
    X operator++(int); // postfix a++
};
```

\[128\] Calling `operator++` explicitly, as in expressions like `a.operator++(2)`, has no special properties: The argument to `operator++` is 2.
struct Y { };  
Y& operator++(Y&);  // prefix ++b  
Y operator++(Y&, int);  // postfix b++  

void f(X a, Y b) {  
  ++a;  // a.operator++();  
  a++;  // a.operator++(0);  
  ++b;  // operator++(b);  
  b++;  // operator++(b, 0);  
  a.operator++();  // explicit call: like ++a;  
  a.operator++(0);  // explicit call: like a++;  
  operator++(b);  // explicit call: like ++b;  
  operator++(b, 0);  // explicit call: like b++;  
}  

— end example ]

The prefix and postfix decrement operators -- are handled analogously.

### 13.6 Built-in operators

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]

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]

For every pair \((T, VQ)\), where \(T\) is an arithmetic type, and \(VQ\) is either volatile or empty, there exist candidate operator functions of the form

\[
VQ T& \quad \text{operator}++(VQ T&); \\
T \quad \text{operator}++(VQ T, \text{int});
\]

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

\[
VQ T& \quad \text{operator}--(VQ T&); \\
T \quad \text{operator}--(VQ T, \text{int});
\]
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
\[
\begin{align*}
& T\& \text{operator}*(T\&);
\end{align*}
\]

For every function type \(T\), there exist candidate operator functions of the form
\[
\begin{align*}
& T\& \text{operator}*(T\&);
\end{align*}
\]

For every type \(T\), there exist candidate operator functions of the form
\[
\begin{align*}
& T* \text{operator}+(T*);
\end{align*}
\]

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 promoted integral type \(T\), there exist candidate operator functions of the form
\[
\begin{align*}
& T \text{operator}~(T);
\end{align*}
\]

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-seqs}, there exist candidate operator functions of the form
\[
\begin{align*}
& \text{CV12 \& operator}->*(\text{CV1 C1\&}, \text{CV2 T C2::\&});
\end{align*}
\]

where \(\text{CV12}\) is the union of \(\text{CV1}\) and \(\text{CV2}\).

For every pair of promoted arithmetic types \(L\) and \(R\), there exist candidate operator functions of the form
\[
\begin{align*}
& LR \text{operator}*(L, R); \\
& LR \text{operator}/(L, R); \\
& LR \text{operator}+(L, R); \\
& LR \text{operator}-(L, R); \\
& \text{bool operator}<=(L, R); \\
& \text{bool operator}>=(L, R); \\
& \text{bool operator}>=(L, R); \\
& \text{bool operator}===(L, R); \\
& \text{bool operator}!==(L, R);
\end{align*}
\]

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

\[
\begin{align*}
\text{bool } & \text{ operator<}(T, T); \\
\text{bool } & \text{ operator>}(T, T); \\
\text{bool } & \text{ operator<=}(T, T); \\
\text{bool } & \text{ operator>=}(T, T); \\
\text{bool } & \text{ operator==}(T, T); \\
\text{bool } & \text{ operator!=}(T, T);
\end{align*}
\]

For every pointer to member type $T$, there exist candidate operator functions of the form

\[
\begin{align*}
\text{bool } & \text{ operator==}(T, T); \\
\text{bool } & \text{ operator!=}(T, T);
\end{align*}
\]

For every pair of promoted integral types $L$ and $R$, there exist candidate operator functions of the form

\[
\begin{align*}
\text{LR } & \text{ operator%}(L, R); \\
\text{LR } & \text{ operator&}(L, R); \\
\text{LR } & \text{ operator^}(L, R); \\
\text{LR } & \text{ operator|}(L, R); \\
\text{L } & \text{ operator<<(L, R);} \\
\text{L } & \text{ operator>>(L, R);}
\end{align*}
\]

where $LR$ is the result of the usual arithmetic conversions between types $L$ and $R$.

For every triple $(L, VQ, R)$, where $L$ is an arithmetic type, $VQ$ is either $\text{volatile}$ or empty, and $R$ is a promoted arithmetic type, there exist candidate operator functions of the form

\[
\begin{align*}
\text{VQ Lk } & \text{ operator==(VQ Lk, R);} \\
\text{VQ Lk } & \text{ operator==}(VQ Lk, R); \\
\text{VQ Lk } & \text{ operator/=(VQ Lk, R);} \\
\text{VQ Lk } & \text{ operator+=}(VQ Lk, R); \\
\text{VQ Lk } & \text{ operator-=}(VQ Lk, R);
\end{align*}
\]

For every pair $(T, VQ)$, where $T$ is any type and $VQ$ is either $\text{volatile}$ or empty, there exist candidate operator functions of the form

\[ T*VQk \text{ operator=(T*VQk, T*);} \]

For every pair $(T, VQ)$, where $T$ is an enumeration or pointer to member type and $VQ$ is either $\text{volatile}$ or empty, there exist candidate operator functions of the form

Draft
13.6 Built-in operators

\[ VQ \ T k \ \text{operator=} (VQ \ T k, T); \]

21 For every pair \((T, VQ)\), where \(T\) is a cv-qualified or cv-unqualified object type and \(VQ\) is either \textit{volatile} or empty, there exist candidate operator functions of the form

\[
T*VQk \ \text{operator}+= (T*VQk, \text{std::ptrdiff_t}); \\
T*VQk \ \text{operator}==(T*VQk, \text{std::ptrdiff_t}); \\
\]

22 For every triple \((L, VQ, R)\), where \(L\) is an integral type, \(VQ\) is either \textit{volatile} or empty, and \(R\) is a promoted integral type, there exist candidate operator functions of the form

\[
VQ L k \ \text{operator}%=(VQ L k, R); \\
VQ L k \ \text{operator}=(VQ L k, R); \\
VQ L k \ \text{operator}>==(VQ L k, R); \\
VQ L k \ \text{operator}^=(VQ L k, R); \\
VQ L k \ \text{operator}^=(VQ L k, R); \\
\]

23 There also exist candidate operator functions of the form

\[
\text{bool} \ \text{operator}!(bool); \\
\text{bool} \ \text{operator}&& (bool, bool); \\
\text{bool} \ \text{operator}|| (bool, bool); \\
\]

24 For every pair of promoted arithmetic types \(L\) and \(R\), there exist candidate operator functions of the form

\[
LR \ \text{operator}?(bool, L, R); \\
\]

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. —\textit{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

\[
T \ \text{operator}?(bool, T, T); \\
\]
Chapter 14  Templates

1  A template defines a family of classes or functions, or an alias for a family of types.

   template-declaration:
       export_opt template < template-parameter-list > declaration

   template-parameter-list:
       template-parameter
       template-parameter-list , template-parameter

   [ Note: The > token following the template-parameter-list of a template-declaration may be the product of replacing a >> token by two consecutive > tokens (14.2). — end note ]

The declaration in a template-declaration shall
   — 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 ]
5 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.

6 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.

7 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 shall not follow the definition of that template. The export keyword shall not be used in a friend declaration.

8 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.

9 A non-exported non-class template shall be defined in every translation unit in which it is implicitly instantiated (14.7.1), unless the corresponding specialization is explicitly instantiated (14.7.2) in some translation unit; no diagnostic is required.

10 [ 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 ]

11 A template-declaration that declares a template alias (14.5.7) shall not be exported.

14.1 Template parameters

The syntax for template-parameters 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 ]

2 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
parameter may be a class template. For example,

```cpp
template<class T> class myarray { /* ... */ };
```

```cpp
template<class K, class V, template<class T> class C = myarray>
class Map {
    C<K> key;
    C<V> value;
};
```

— end note ]

3 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;
```

```cpp
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 ]

4 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.

5 [ Note: other types are disallowed either explicitly below or implicitly by the rules governing the form of template-arguments (14.3). — end note ] The top-level cv-qualifiers on the template-parameter are ignored when determining its type.

6 A non-type non-reference template-parameter is 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
    &x; // error: address of non-reference template-parameter
    &i; // OK
}
```

Draft
14.1 Template parameters

\begin{verbatim}
int& ri = i;    // error: non-const reference bound to temporary
const int& cri = i;  // OK: const reference bound to temporary
\}

— end example ]

A non-type \textit{template-parameter} shall not be declared to have floating point, class, or void type. [ \textit{Example:} ]

\begin{verbatim}
template<double d> class X;  // error
template<double* pd> class Y; // OK
\end{verbatim}

— end example ]

A non-type \textit{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. [ \textit{Example:} ]

\begin{verbatim}
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{verbatim}

— end example ]

A \textit{default template-argument} is a \textit{template-argument} (14.3) specified after = in a \textit{template-parameter}. A \textit{default template-argument} may be specified for any kind of \textit{template-parameter} (type, non-type, template) that is not a template parameter pack. A \textit{default template-argument} may be specified in a template declaration. A \textit{default template-argument} shall not be specified in the \textit{template-parameter-lists} of the definition of a member of a class template that appears outside of the member’s class. A \textit{default template-argument} shall not be specified in a friend class template declaration. If a friend function template declaration specifies a default \textit{template-argument}, that declaration shall be a definition and shall be the only declaration of the function template in the translation unit.

\begin{verbatim}
\end{verbatim}

The set of default \textit{template-arguments} available for use with a template declaration or definition is obtained by merging the default arguments from the definition (if in scope) and all declarations in scope in the same way default function arguments are (8.3.6). [ \textit{Example:} ]

\begin{verbatim}
template<class T1, class T2 = int> class A;
template<class T1 = int, class T2> class A;
\end{verbatim}

is equivalent to

\begin{verbatim}
template<class T1 = int, class T2 = int> class A;
\end{verbatim}

— end example ]

If a \textit{template-parameter} of a class template has a default \textit{template-argument}, each subsequent \textit{template-parameter} shall either have a default \textit{template-argument} supplied or be a template parameter pack. If a \textit{template-parameter} of a class template is a template parameter pack, it shall be the last \textit{template-parameter}. [ \textit{Note:} These are not requirements for function templates because template arguments might be deduced (14.8.2). [ \textit{Example:} ]

Draft
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 ] — end note ]

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 {
  /* */
};
template<class T> void f(T* p = new T);
```

— 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 > // syntax error
class X { /* */ };
```

```cpp
template<int i = (3 > 4) > // OK
class Y { /* */ };;
```

— end example ]

A template-parameter of a template template-parameter is permitted to have a default template-argument. When such default arguments are specified, they apply to the template template-parameter in the scope of the template template-parameter. [Example:

```cpp
template <class T = float> struct B {};
template <template <class TT = float> class T> struct A {
  inline void f();
  inline void g();
};
template <template <class TT> class T> void A<T>::f() {
  T t; // error - TT has no default template argument
}
template <template <class TT = char> class T> void A<T>::g() {
  T t; // OK - T<char>
}
```

— end example ]

If a template-parameter is a type-parameter with an ellipsis prior to its optional identifier or is a parameter-declaration that declares a parameter pack (8.3.5), then the template-parameter is a template parameter pack (14.5.3). [Example:
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 ...
  template-argument-list , template-argument ...

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 >\[130\] is taken as the ending delimiter rather than a greater-than operator. Similarly, the first non-nested >> is treated as two consecutive but distinct > tokens, the first of which is taken as the end of the template-argument-list and completes the template-id. [ Note: The second > token produced by this replacement rule may terminate an enclosing template-id construct or it may be part of a different construct (e.g. a cast). — end note ] [ Example:

template<int i> class X { /* ... */ };
X< 1>2 > x1; // syntax error
X<(1>2)> x2; // OK

template<class T> class Y { /* ... */ };
Y<X<1>> x3; // OK, same as Y<X<1>> x3;
Y<X<6>>1>> x4; // syntax error
Y<X<(6>>1)>> x5; // OK

130] A > that encloses the type-id of a dynamic_cast, static_cast, reinterpret_cast or const_cast, or which encloses the template-arguments of a subsequent template-id, is considered nested for the purpose of this description.

Draft
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:

```cpp
struct X {
    template<std::size_t> X* alloc();
    template<std::size_t> static X* adjust();
};
template<class T> void f(T* p) {
    T* p1 = p->alloc<200>(); // ill-formed: < means less than
    T* p2 = p->template alloc<200>(); // OK: < starts template argument list
    T::adjust<100>(); // ill-formed: < means less than
    T::template adjust<100>(); // OK: < starts template argument list
}
```
— end example ]

5 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 not 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 [temp.arg]

1 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-arguments. [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]; }
};

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. [Example:

```cpp
template<class T> void f();
template<int I> void f();

void g() {
    f<int()>(); // int() is a type-id: call the first f()
}
— end example ]

The name of a template-argument shall be accessible at the point where it is used as a template-argument. [Note: if the name of the template-argument is accessible at the point where it is used as a template-argument, there is no further access restriction in the resulting instantiation where the corresponding template-parameter name is used. — end note ] [Example:

```cpp
template<class T> class X {
    static T t;
};

class Y {
    private:
        struct S { /* ... */ };
        X<S> x; // OK: S is accessible
        // x<Y::S> has a static member of type Y::S
        // OK: even though Y::S is private
    };

    X<Y::S> y; // error: S not accessible
— end example ]

For a template-argument that is a class type or a class template, the template definition has no special access rights to the members of the template-argument. [Example:

```cpp
template <template <class TT> class T> class A {
    typename T<int>::S s;
};

template <class U> class B {
    private:
        struct S { /* ... */ };
When template argument packs or default template-arguments are used, a template-argument list can be empty. In that case the empty <> brackets shall still be used as the template-argument-list. [Example:

```cpp
template<class T = char> class String;
String<> p;          // OK: String<char>
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-arguments. [Example:

```cpp
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-arguments do not match the template-parameters are ignored. If none of the function templates have matching template-parameters, 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

A template-argument for a template-parameter which is a type shall be a type-id.

A type without linkage (3.5) shall not be used as a template-argument for a template type-parameter.

[ Example:

```cpp
template <class T> class X { /* ... */};
void f() {
  struct S { /* ... */};
}
```

Draft
14.3 Template arguments

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 “rvalue reference to cv T” creates the type T [ Example:

template <class T> class X {
    ...;
};
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&&

— end example ]

14.3.2 Template non-type arguments

1 A template-argument for a non-type, non-template template-parameter shall be one of:
   — an integral constant expression; 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.

2 [ 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:

Draft
template<class T, char* p> class X {
    X();
    X(const char* q) { /* ... */ }
};

X<int, "Studebaker"> x1;  // error: string literal as template-argument
char p[] = "Vivisectionist";
X<int, p> x2;             // OK

— end example ] — end note ]

3 [ 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 ]

4 [ 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:

    template<const int& CRI> struct B { /* ... */ }; 

        B<1> b2; // error: temporary would be required for template argument

        int c = 1;
        B<c> b1; // OK

— end example ] — end note ]

5 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; if the template-argument is of type std::nullptr_t, the null pointer conversion (4.10) is applied. [ Note: In particular, neither the null pointer conversion for a zero-valued integral constant expression (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
A template-parameter of pointer type. However, both \((\text{int}*)0\) and \texttt{nullptr} are valid template-arguments 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 shall be an lvalue.

— For a non-type template-parameter of type pointer to function, the function-to-pointer conversion (4.3) is applied; if the template-argument is of type \texttt{std::nullptr_t}, the null pointer conversion (4.10) 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, if the template-argument is of type \texttt{std::nullptr_t}, the null member pointer conversion (4.11) is applied; otherwise, 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; if the template-argument is of type \texttt{std::nullptr_t}, the null member pointer conversion (4.11) is applied.

Example:

```cpp
template<\texttt{const \texttt{int} \* pci}> \texttt{struct X \{ \texttt{\ldots } \};}
\texttt{int ai[10];}
\texttt{X<ai> xi; \quad \texttt{// array to pointer and qualification conversions}}

\texttt{struct Y \{ \texttt{\ldots } \};}
\texttt{template<\texttt{const Y \& b}> \texttt{struct Z \{ \texttt{\ldots } \};}}
\texttt{Y y;}
\texttt{Z<y> z; \quad \texttt{// no conversion, but note extra cv-qualification}}

\texttt{template<\texttt{int \& (pa)[5]> \texttt{struct W \{ \texttt{\ldots } \};}}
\texttt{int b[5];}
\texttt{W<b> w; \quad \texttt{// no conversion}}

\texttt{void f(char);}
\texttt{void f(int);}

\texttt{template<\texttt{void (*pf)(int)> \texttt{struct A \{ \texttt{\ldots } \};}}
\texttt{A<\&f> a; \quad \texttt{// selects f(int)}}

— end example
```

14.3 Template arguments

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.

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:

```cpp
template<class T> class A { 
    int x;
};
template<class T> class A<T*> { // partial specialization
    long x;
};
template<template<class U> class V> class C {
    V<int> y;
    V<int*> z;
};
C<A> c; // V<int> within C<A> uses the primary template,
// so c.y.x has type int
// V<int*> within C<A> uses the partial specialization,
// so c.z.x has type long
```

— end example ]

[Example:

```cpp
template<class T> class A { /* ... */};
template<class T, class U = T> class B { /* ... */};
template <class ... Types> class C { /* ... */};
template<template<class> class P> class X { /* ... */};
template<template<class ...> class Q> class Y { /* ... */};
X<A> xa; // OK
X<B> xb; // ill-formed: default arguments for the parameters of a template argument are ignored
X<C> xc; // ill-formed: a template parameter pack does not match a template parameter
Y<A> ya; // ill-formed: a template parameter pack does not match a template parameter
Y<B> yb; // ill-formed: a template parameter pack does not match a template parameter
Y<C> yc; // OK
```

— end example ]

A template-argument matches a template template-parameter (call it P) when each of the template parameters in the template-parameter-list of the template-argument’s corresponding class template or template alias (call it A) matches the corresponding template parameter in the template-parameter-list of P. When P’s template-parameter-list contains a template parameter pack (14.5.3), the template parameter pack will match zero or more template parameters or template parameter packs in the template-parameter-list of A with the same type and form as the template parameter pack in P (ignoring whether those template parameters are template parameter packs) [Example:

```cpp
template <class T> struct eval;
```
14.4 Type equivalence

Two template-ids refer to the same class or function if

— their template-names refer to the same template, and
— their corresponding type template-arguments are the same type, and
— their corresponding non-type template arguments of integral or enumeration type have identical values, and
— their corresponding non-type template-arguments of pointer type refer to the same external object or function or are both the null pointer value, and
— their corresponding non-type template-arguments of pointer-to-member type refer to the same class member or are both the null member pointer value, and
— their corresponding non-type template-arguments of reference type refer to the same external object or function, and
— their corresponding template template-arguments refer to the same template.

[Example:

```cpp
template<
    class E, int size>
    class buffer { /* ... */};

buffer<
    char, 2 * 512> x;
buffer<
    char, 1024> y;
```

declares x and y to be of the same type, and

```cpp
template<
    class T, void(*err_fct)()>
    class list { /* ... */};
list<
    int, &error_handler1> x1;
list<
    int, &error_handler2> x2;
list<
    int, &error_handler2> x3;
list<
    char, &error_handler2> x4;
```

declares x2 and x3 to be of the same type. Their type differs from the types of x1 and x4.
template<template<class> T> struct X { };  
template<class> struct Y { };  
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 [temp.decls]

1 A template-id, that is, the template-name followed by a template-argument-list shall not be specified in the declaration of a primary template declaration. [Example:

    template<class T1, class T2, int I> class A<T1, T2, I> { }; // error
    template<class T1, int I> void sort<T1, I>(T1 data[I]); // error

—end example ] [ Note: however, this syntax is allowed in class template partial specializations (14.5.5). —end note ]

2 For purposes of name lookup and instantiation, default arguments 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.

3 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 [temp.class]

1 A class template defines the layout and operations for an unbounded set of related types. [Example: a single class template List might provide a common definition for list of int, list of float, and list of pointers to Shapes. —end example ]

[ Example: An array class template might be declared like this:

    template<class T> class Array {
    T* v;
    int sz;
    public:
    explicit Array(int);
    T& operator[](int);
    T& elem(int i) { return v[i]; }
    };

2 The prefix template <class T> specifies that a template is being declared and that a type-name T will be used in the declaration. In other words, Array is a parameterized type with T as its parameter. —end example ]

3 When a member function, a member class, a 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-parameters 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
14.5 Template declarations

in the template parameter list of the member. Each template parameter pack shall be expanded with an ellipsis in the

```c++
template<class T1, class T2> struct A {
    void f1();
    void f2();
};
```

template<class T2, class T1> void A<T2,T1>::f1() { } // OK

template<class T2, class T1> void A<T1,T2>::f2() { } // error

```c++
template<class ... Types> struct B {
    void f3();
    void f4();
};
```

template<class ... Types> void B<Types ...>::f3() { } // OK

template<class ... Types> void B<Types>::f4() { } // error

--- end example --

4 In a redeclaration, partial specialization, explicit specialization or explicit instantiation of a class template, the class-key
shall agree in kind with the original class template declaration (7.1.6.3).

14.5.1.1 Member functions of class templates

A member function of a class template may be defined outside of the class template definition in which it is declared. [Example:

```c++
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:

```c++
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-arguments for a member function of a class template are determined by the template-arguments of the
type of the object for which the member function is called. [Example: the template-argument for Array<T>::operator []() will be determined by the Array to which the subscripting operation is applied.

Draft
14.5 Template declarations

1 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,

```
template<class T> struct A {
    class B;
};
A<int>::B* b1;       // OK: requires A to be defined but not A::B
```

— end note ]

14.5.1.3 Static data members of class templates

1 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:

```
template<class T> class X {
    static T s;
};
template<class T> T X<T>::s = 0;
```

— end example ]

14.5.2 Member templates

1 A template can be declared within a class or class template; such a template is called a member template. A member template can be defined within or outside its class definition or class template definition. A member template of a class template that is defined outside of its class template definition shall be specified with the template-parameters of the class template followed by the template-parameters of the member template. [Example:

```
template<class T> struct string {
    template<class T2> int compare(const T2&);
    template<class T2> string(const string<T2>& s) { /* ... */ }
};
```

```
template<class T> template<class T2> int string<T>::compare(const T2& s) {
    // ...
}
```

— end example ]

2 A local class 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

template <class T> struct AA {
    template <class C> virtual void g(C); // error
    virtual void f(); // OK
};

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 ]

3 A member function template shall not be virtual. [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 ]

4 A specialization of a member function template does not override a virtual function from a base class. [Example:

```cpp
struct A {
    template <class T> operator T*();
};

template <class T> A::operator T*(){ return 0; }
```

— 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; }
```

— specialization

Draft
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]

1 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> eror; // error: 0 is not a type

— end example ]

2 A function parameter pack is a function parameter that accepts zero or more function arguments. [ Example:
    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 ]

3 A parameter pack is either a template parameter pack or a function parameter pack.

4 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:

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:

    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>::type T2; // error: different number of arguments specified for Args1 and Args2  

    template<class ... Args> void g(Args ... args) {
      f(const_cast<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:

```cpp
template<class T> class task;
template<class T> task<T>* preempt(task<T>*);

template<class T> class task {
  friend void next_time();
  friend void process(task<T>*);
  friend task<T>* preempt<T>(task<T>*);
  template<class C> friend int func(C);

  friend class task<int>;
  template<class P> friend class frd;
};
```

Here, each specialization of the `task` class template has the function `next_time` as a friend; because `process` does not have explicit template-arguments, each specialization of the `task` class template has an appropriately typed function `process` as a friend, and this friend is not a function template specialization; because the friend `preempt` has an explicit template-argument `T`, each specialization of the `task` class template has the appropriate specialization of the function template `preempt` as a friend; and each specialization of the `task` class template has all specializations of the function template `func` as friends. Similarly, each specialization of the `task` class template has the class template specialization `task<int>` as a friend, and has all specializations of the class template `frd` as friends. — end example

2 A friend template may be declared within a class or class template. A friend function template may be defined within a class or class template, but a friend class template may not be defined in a class or class template. In these cases, all specializations of the friend class or friend function template are friends of the class or class template granting friendship.

Example:

```cpp
class A {
  template<class T> friend class B; // OK
  template<class T> friend void f(T){ /* ... */ } // OK
};

— end 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.

2 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 shall 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.

3 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).

4 [Example:

```cpp
template<class T1, class T2, int I> class A { }; // #1
template<class T, int I> class A<T, T*, I> { }; // #2
template<class T1, class T2, int I> class A<T1*, T2, I> { }; // #3
template<class T> class A<int, T*, 5> { }; // #4
template<class T1, class T2, int I> class A<T1, T2*, I> { }; // #5
```

The first declaration declares the primary (unspecialized) class template. The second and subsequent declarations declare partial specializations of the primary template. — end example]

5 The template parameters are specified in the angle bracket enclosed list that immediately follows the keyword `template`. For partial specializations, the template argument list is explicitly written immediately following the class template name. For primary templates, this list is implicitly described by the template parameter list. Specifically, the order of the template arguments is the sequence in which they appear in the template parameter list. [Example: the template argument list for the primary template in the example above is `<T1, T2, I>`. — end example] [Note: the template argument list shall not be specified in the primary template declaration. For example,

```cpp
template<class T1, class T2, int I> class A<T1, T2, I> { }; // error
```

— end note]

6 A class template partial specialization may be declared or redeclared in any namespace scope in which its definition may be defined (14.5.1 and 14.5.2). [Example:

```cpp
template<class T> struct A {
    struct C {
        template<class T2> struct B { }; // partial specialization of A<T>::C::B<T2>
    };
};
```

```cpp
A<short>::C::B<int*> absip; // uses partial specialization
```

— Draft
Partial specialization declarations themselves are not found by name lookup. Rather, when the primary template name is used, any previously declared partial specializations of the primary template are also considered. One consequence is that a using-declaration which refers to a class template does not restrict the set of partial specializations which may be found through the using-declaration. [Example:

```cpp
namespace N {
    template<class T1, class T2> class A { }; // primary template
}

using N::A; // refers to the primary template

namespace N {
    template<class T> class A<T, T*> { }; // partial specialization
}

A<int,int*> a; // uses the partial specialization, which is found through
               // the using declaration which refers to the primary template
```

— end example ]

A non-type argument is non-specialized if it is the name of a non-type parameter. All other non-type arguments are specialized.

Within the argument list of a class template partial specialization, the following restrictions apply:

— A partially specialized non-type argument expression shall not involve a template parameter of the partial specialization except when the argument expression is a simple identifier. [Example:

```cpp
template <int I, int J> struct A {};  
template <int I> struct A<I+5, I*2> {}; // error

template <int I, int J> struct B {}; 
template <int I> struct B<I, I> {};  // OK
```

— end example ]

— The type of a template parameter corresponding to a specialized non-type argument shall not be dependent on a parameter of the specialization. [Example:

```cpp
template <class T, T t> struct C {};  
template <class T> struct C<T, 1>; // error

template< int X, int (*array_ptr)[X] > class A {}; 
int array[5]; 
template< int X > class A<X,&array> { }; // error
```

— end example ]

— 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.\(^\text{132}\) 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).\(^\text{2}\)

\[\begin{align*}
A<\text{int, int, 1}> & \quad a1; \quad \text{// uses #1} \\
A<\text{int, int*, 1}> & \quad a2; \quad \text{// uses #2, T is int, I is 1} \\
A<\text{int, char*, 5}> & \quad a3; \quad \text{// uses #4, T is char} \\
A<\text{int, char*, 1}> & \quad a4; \quad \text{// uses #5, T1 is int, T2 is char, I is 1} \\
A<\text{int*, int*, 2}> & \quad a5; \quad \text{// ambiguous: matches #3 and #5}
\end{align*}\]

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.\(^\text{3}\)

In a type name that refers to a class template specialization, (e.g., \(A<\text{int, int, 1}>\)) the argument list shall match the template parameter list of the primary template. The template arguments of a specialization are deduced from the arguments of the primary template.

### 14.5.5.2 Partial ordering of class template specializations

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

\(^\text{132}\) There is no way in which they could be used.
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:

```cpp
template<int I, int J, class T> class X { }; // #1
template<int I, int J> class X<I, J, int> { }; // #2

template<int I> class X<I, I, 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:

```cpp
// primary template
template<class T, int I> struct A {
    void f();
};

template<class T, int I> void A<T,I>::f() {} // #A

// class template partial specialization
template<class T> struct A<T,2> {
    void f();
    void g();
    void h();
};

// member of class template partial specialization
template<class T> void A<T,2>::g() {} // #B

// explicit specialization
template<> void A<char,2>::h() {} // #C
```

```cpp
int main() {
    A<char,0> a0;
}
```


\begin{verbatim}
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 \texttt{f} for \texttt{A<T,2>}
             // the primary template is not used here
}

— end example ]

2 If a member template of a class template is partially specialized, the member template partial specializations are member templates of the enclosing class template; if the enclosing class template is instantiated (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:

\begin{verbatim}
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 [temp.fct]

1 A function template defines an unbounded set of related functions. [ Example: a family of sort functions might be declared like this:

\begin{verbatim}
template<class T> class Array { }
template<class T> void sort(Array<T>&);

— end example ]

2 A function template can be overloaded with other function templates and with 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.\(^{\text{133)}}\)

### 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); // calls f<int>(int*)
}

// file2.c
template<class T>
    void f(T);
void h(int* p) {
    f(p); // calls f<int*>(int*)
}
```

— end example ]

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
```

— end note ]

When an expression that references a template parameter is used in the function parameter list or the return type in the declaration of a function template, the expression that references the template parameter is part of the signature of the function template. This is necessary to permit a declaration of a function template in one translation unit to be linked with another declaration of the function template in another translation unit and, conversely, to ensure that function templates that are intended to be distinct are not linked with one another. [Example:

```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
```

— end example ] [Note: Most expressions that use template parameters use non-type template parameters, but it is possible for an expression to reference a type parameter. For example, a template type parameter can be used in the `sizeof` operator. — end note ]

Two expressions involving template parameters are considered equivalent if two function definitions containing the expressions would satisfy the one definition rule (3.2), except that the tokens used to name the template parameters may differ as long as a token used to name a template parameter in one expression is replaced by another token that names the same template parameter in the other expression. [Example:

\(^{\text{133)}} 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 that are not equivalent are \textit{functionally equivalent} if, for any given set of template arguments, the evaluation of the expression results in the same value.

Two function templates are \textit{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 \textit{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.

\textit{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:

\begin{verbatim}
// 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{verbatim}

--- end note ---

14.5.6.2 Partial ordering of function templates

If a function template is overloaded, the use of a function template specialization might be ambiguous because template argument deduction (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:

\begin{itemize}
  \item during overload resolution for a call to a function template specialization (13.3.3);
  \item when the address of a function template specialization is taken;
  \item 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);
  \item when a friend function declaration (14.5.4), an explicit instantiation (14.7.2) or an explicit specialization (14.7.3) refers to a function template specialization.
\end{itemize}

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.

3 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.

4 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*)
f(float x);
g(A<int> z);
h(z); // overload resolution selects h(A<T>&)
    const A<int> z2;
    h(z2); // h(const T&) is called because h(A<T>&) is not callable
}
```
—end example]

5 The presence of unused ellipsis and default arguments has no effect on the partial ordering of function templates.

[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
}
```
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-arguments` for the `template-parameters` in the `type-id` of the template alias. [Note: A template alias name is never deduced. — end note] [Example:

```cpp
template<class T> struct Alloc {
    /* ...
};
template<class T> using Vec = vector<T, Alloc<T>>;
Vec<int> v;    // same as vector<int, Alloc<int>> v;

#define Vec template<class T> struct Vec {
#define using Vec = vector<T, Alloc<T>>;

#define Vec template<class T> void process(Vec<T>& v)
#define void process(vector<T, Alloc<T>>& w)    // error: redefinition

#define Vec template<class T>
#define void process(Vec<T>& v)
#define template<template<class> class TT>
#define void f(TT<int>);

#define Vec template<template<class,class> class TT>
#define void g(TT<int, allocator<int>>);
```

— 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:

```cpp
// no B declared here
class X;

template<class T> class Y {
```
class Z;  // forward declaration of member class

void f() {
    X* a1;  // declare pointer to X
    T* a2;  // declare pointer to T
    Y* a3;  // declare pointer to Y<T>
    Z* a4;  // declare pointer to Z
    typedef typename T::A TA;
    TA* a5;  // declare pointer to T's A
    typename T::A* a6;  // declare pointer to T's A
    T::A* a7;  // 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 */

3 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
type-name-specifier. If the qualified-id in a typename-specifier does not denote a type, the program is ill-formed.

type-name-specifier:
    typename ::opt nested-name-specifier identifier
    typename ::opt nested-name-specifier template ::opt simple-template-id

4 If a specialization of a template is instantiated for a set of template-arguments such that the qualified-id prefixed by
typename does not denote a type, the specialization is ill-formed. The usual qualified name lookup (3.4.3) is used to
find the qualified-id even in the presence of typename. [Example:

    struct A {
        struct X { };  // OK: T::X refers to B::X
        int X;
    };
    struct B {
        struct X { };  // error: T::X refers to the data member A::X not the struct A::X
    };

    template<class T> void f(T t) {
        typename T::X x;
    }

    void foo() {
        A a;
        B b;
        f(b);  // OK: T::X refers to B::X
        f(a);  // error: T::X refers to the data member A::X not the struct A::X
    }

    /* end example */
A qualified name used as the name in a 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. \[end note\]

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
};
```

\[end example\]

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
    }
};
```

\[end example\]

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). \[ Example:\]
int cnt;
public:
Set();
Set<T>(const Set<T>&);
void printall() {
    for (int i = 0; i<cnt; i++)
        cout << p[i] << '\n';
}
};

in the example, i is the local variable i declared in printall, cnt is the member cnt declared in Set, and cout is the standard output stream declared in iostream. However, not every declaration can be found this way; the resolution of some names must be postponed until the actual template-arguments are known. For example, even though the name operator<< is known within the definition of printall() and a declaration of it can be found in <iostream>, the actual declaration of operator<< needed to print p[i] cannot be known until it is known what type T is (14.6.2).
— end example

If a name does not depend on a template-parameter (as defined in 14.6.2), a declaration (or set of declarations) for that name shall be in scope at the point where the name appears in the template definition; the name is bound to the declaration (or declarations) found at that point and this binding is not affected by declarations that are visible at the point of instantiation. [Example:

    void f(char);

    template<class T> void g(T t) {
        f(1);                       // f(char)
        f(T(1));                    // dependent
        f(t);                       // dependent
        dd++;                       // not dependent
            // error: declaration for dd not found
    }

    enum E { e};
    void f(E);

    double dd;
    void h() {
        g(e);                       // will cause one call of f(char) followed
            // by two calls of f(E)
        g('a');                    // will cause three calls of f(char)
    }

    — end example ]

[Note: for purposes of name lookup, default arguments of function templates and default arguments of member functions of class templates are considered definitions (14.5). — end note]
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-parameters 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.

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-arguments of the class template specialization or partial specialization enclosed in <>. [Example:

```cpp
template<class T> class Y;
template<> class Y<int> {
  Y* p; // meaning Y<int>
  Y<char>* q; // meaning Y<char>
};
```

— end example ]

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:

```cpp
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 ]

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:

```cpp
template <class T> struct Base { };
template <class T> struct Derived: Base<int>, Base<char> {
  typename Derived::Base b; // error: ambiguous
  typename Derived::Base<double> d; // OK
};
```

— end example ]

When the normal name of the template (i.e., the name from the enclosing scope, not the injected-class-name) is used without a template-argument-list, it refers to the class template itself and not a specialization of the template. [Example:

```cpp
template <class T> class X {
  X* p; // meaning X<T>
  X<T>* p2;
};
```
6 The scope of a template-parameter extends from its point of declaration until the end of its template. A template-parameter hides any entity with the same name in the enclosing scope. [Note: this implies that a template-parameter can be used in the declaration of subsequent template-parameters and their default arguments but cannot be used in preceding template-parameters or their default arguments. For example,

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

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

```cpp
template<class T> class X : public Array<T> {
    ...}
```

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

7 A template-parameter shall not be redeclared within its scope (including nested scopes). A template-parameter shall not have the same name as the template name. [Example:

```cpp
template<class T, int i> class Y {
    int T; // error: template-parameter redeclared
    void f() {
        char T; // error: template-parameter redeclared
    }
};
```

— 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 template-parameter. [Example:

```cpp
template<class T> struct A {
    struct B { /* ... */);
    void f();
};
```

```cpp
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 ]

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;  // k's B
    a b;  // error: k's a isn't a type name
};

— end example ]

14.6.2 Dependent names [temp.dep]

Inside a template, some constructs have semantics which may differ from one instantiation to another. Such a construct depends on the template parameters. In particular, types and expressions may depend on the type and/or value of template parameters (as determined by the template arguments) and this determines the context for name lookup for certain names. Expressions may be type-dependent (on the type of a template parameter) or value-dependent (on the value of a non-type template parameter). In an expression of the form:

\[ \text{postfix-expression ( expression-list_{op} )} \]

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) {

    Draft
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

3 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:

```cpp
typedef double A;
template<class T> class B {
   typedef int A;
};
template<class T> struct X : B<T> {
   A a; // a has type double
};
```

The type name `A` in the definition of `X<T>` binds to the typedef name defined in the global namespace scope, not to the typedef name defined in the base class `B<T>`. — end example] [Example:

```cpp
struct A {
   struct B { /* ... */
   int a;
   int Y;
   };

   int a;

   template<class T> struct Y : T {
      struct B { /* ... */
      B b; // The B defined in Y
      void f(int i) { a = i; } // ::a
      Y* p;                   // Y<T>
   };

   Y<A> ya;
```

The members `A::B`, `A::a`, and `A::Y` of the template argument `A` do not affect the binding of names in `Y<A>`. — end example

14.6.2.1 Dependent types [temp.dep.type]

1 In the definition of a class template, a nested class of a class template, a member of a class template, or a member of a nested class of a class template, 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 n-th template parameter is a parameter pack, the n-th 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 n-th template argument has the value of the n-th template parameter of the class template. If the n-th template parameter is a template parameter pack, the n-th 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:

```cpp
template <class T> class A {
    A* p1; // A is the current instantiation
    A<T>* p2; // A<T> is the current instantiation
    A<T*>* p3; // A<T*> is not the current instantiation
    ::A<T>* p4; // ::A<T> is the current instantiation
    class B {
        B* p1; // B is the current instantiation
        A<T>::B* p2; // A<T>::B is the current instantiation
        typename A<T>::::B* p3; // A<T>::::B is not the current instantiation
        // 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_I>* 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 ]

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:

```cpp
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();
};
```

```cpp
template <class T> int A<T>::f() {
    return i; // i refers to a member of the current instantiation
}
```

— end example ]

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.

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

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:

\[
\text{simple-type-specifier ( expression-list_{opt} )}
\]

\[
\text{new new-placement_{opt} new-type-id new-initializer_{opt}}
\]

\[
\text{dynamic_cast < type-id > ( expression )}
\]

\[
\text{static_cast < type-id > ( expression )}
\]

\[
\text{const_cast < type-id > ( expression )}
\]

\[
\text{reinterpret_cast < type-id > ( expression )}
\]

\[
\text{(type-id) cast-expression}
\]

4 Expressions of the following forms are never type-dependent (because the type of the expression cannot be dependent):

\[
\text{literal}
\]

\[
\text{postfix-expression . pseudo-destructor-name}
\]

\[
\text{postfix-expression \rightarrow pseudo-destructor-name}
\]

\[
\text{sizeof unary-expression}
\]

\[
\text{sizeof ( type-id )}
\]

\[
\text{sizeof ... ( identifier )}
\]

\[
\text{alignof ( type-id )}
\]

\[
\text{typeid ( expression )}
\]

\[
\text{typeid ( type-id )}
\]

\[
\text{opt delete cast-expression}
\]

\[
\text{opt delete [ ] cast-expression}
\]

\[
\text{throw assignment-expression_{opt}}
\]

[Note: For the standard library macro offsetof, see 18.1. — end note]

5 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]

### 14.6.2.3 Value-dependent expressions

1 Except as described below, a constant expression is value-dependent if any subexpression is value-dependent.

2 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 literal type and is initialized with an expression that is value-dependent.

Draft
Expressions of the following form are value-dependent if the unary-expression is type-dependent or the type-id is dependent:

```
sizeof unary-expression
sizeof ( type-id )
alignof ( type-id )
```

[ Note: For the standard library macro offsetof, see 18.1. — end note ]

3 Expressions of the following form are value-dependent if either the type-id or simple-type-specifier is dependent or the expression or cast-expression is value-dependent:

```
simple-type-specifier ( expression-list_opt )
static_cast < type-id > ( expression )
const_cast < type-id > ( expression )
reinterpret_cast < type-id > ( expression )
(type-id) cast-expression
```

4 Expressions of the following form are value-dependent:

```
sizeof ... ( identifier )
```

### 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

```
qualified-id
& qualified-id
```

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

1 Non-dependent names used in a template definition are found using the usual name lookup and bound at the point they are used. [Example:

```c
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
}
```
14.6.4 Dependent name resolution

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

For a function template specialization, a member function template specialization, or a specialization for a member function or static data member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization and the context from which it is referenced depends on a template parameter, the point of instantiation of the specialization is the point of instantiation of the enclosing specialization. Otherwise, the point of instantiation for such a specialization immediately follows the namespace scope declaration or definition that refers to the specialization.

If a function template or member function of a class template is called in a way which uses the definition of a default argument of that function template or member function, the point of instantiation of the default argument is the point of instantiation of the function template or member function specialization.

For a class template specialization, a class member template specialization, or a specialization for a class member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization, if the context from which the specialization is referenced depends on a template parameter, and if the specialization is not instantiated previous to the instantiation of the enclosing template, the point of instantiation is immediately before the point of instantiation of the enclosing template. Otherwise, the point of instantiation for such a specialization immediately precedes the namespace scope declaration or definition that refers to the specialization.

If a virtual function is implicitly instantiated, its point of instantiation is immediately following the point of instantiation of its enclosing class template specialization.

An explicit instantiation definition is an instantiation point for the specialization or specializations specified by the explicit instantiation.

The instantiation context of an expression that depends on the template arguments is the set of declarations with external linkage declared prior to the point of instantiation of the template specialization in the same translation unit.

A specialization for a function template, a member function template, or of a member function or static data member of a class template may have multiple points of instantiations within a translation unit. 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...
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).

Example:

```cpp
template<typename T> struct number {
    number(int);
    friend number gcd(number x, number y) { return 0; };
};

void g() {
    number<double> a(3), b(4);
    a = gcd(a, b); // finds gcd because number<double> is an associated class, making gcd visible
                    // in its namespace (global scope)
    b = gcd(3, 4); // ill-formed: gcd is not visible
}
```

— end example

14.7 Template instantiation and specialization

The act of instantiating a function, a class, a member of a class template or a member template is referred to as template instantiation.

---

\[134\] Friend declarations do not introduce new names into any scope, either when the template is declared or when it is instantiated.
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 function template is called an instantiated member function. A member class instantiated from a member class template is called an instantiated member class.

An explicit specialization may be declared for a function template, a class template, a member of a class template or a member template. An explicit specialization declaration is introduced by `template<>`. In an explicit specialization declaration for a class template, a member of a class template or a class member template, the name of the class that is explicitly specialized shall be a `simple-template-id`. In the explicit specialization declaration for a function template or a member function template, the name of the function or member function explicitly specialized may be a `template-id`. [ Example:

```cpp
    template<class T = int> struct A {
        static int x;
    };
    template<class U> void g(U) { }
    template<> struct A<double> { }; // specialize for T == double
    template<> struct A<> { }; // specialize for T == int
    template<> void g(char) { } // specialize for U == char
    // U is deduced from the parameter type
    template<> void g<int>(int) { } // specialize for U == int
    template<> int A<char>::x = 0; // specialize for T == char

    template<class T = int> struct B {
        static int x;
    };
    template<> int B<>::x = 1; // specialize for T == int
    
    //end example ]
```

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-arguments`,

- an explicit instantiation definition shall appear at most once in a program,
- an explicit specialization shall be defined at most once in a program (according to 3.2), and
- both an explicit instantiation and a declaration of an explicit specialization shall not appear in a program unless the explicit instantiation follows a declaration of the explicit specialization.

An implementation is not required to diagnose a violation of this rule.

Each class template specialization instantiated from a template has its own copy of any static members. [ Example:

```cpp
    template<class T> class X {
        static T g;
    }
```

Draft
14.7 Template instantiation and specialization

Template instantiation and specialization

```cpp
};
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.

2 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.

3 [Example:

```cpp
template<class T> struct Z {
    void f();
    void g();
};

void h() {
    Z<int> a; // instantiation of class Z<int> required
    Z<char*> p; // instantiation of class Z<char> not required
    Z<double*> q; // instantiation of class Z<double> not required

    a.f(); // instantiation of Z<int>::f() required
    p->g(); // instantiation of Z<char>::g() required, and
            // instantiation of Z<char>::f() 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 ]

6 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 ]

7 The implicit instantiation of a class template does not cause any static data members of that class to be implicitly instantiated.

8 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 function, a member class or a static data member of a class template that does not require instantiation. It is unspecified whether or not an implementation implicitly instantiates a virtual member function of a class template if the virtual member function would not otherwise be instantiated. The use of a template specialization in a default argument shall not cause the template to be implicitly instantiated except that a class template may be instantiated where its complete type is needed to determine the correctness of the default argument. The use of a default argument in a function call causes specializations in the default argument to be implicitly instantiated.

Implicitly instantiated class 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:

```cpp
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) {
        int i = m.get("Nicholas");
    }
}
```

A call of `lt.get()` from `Map<char*,int>::get()` would place `List<int>::get()` in the namespace `N` rather than in the global namespace. — end example]

If a function template `f` is called in a way that requires a default argument 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 `f` used at that point. This analysis is called default argument instantiation. The instantiated default argument is then used as the argument of `f`.

Each default argument is instantiated independently. [Example:

```cpp
template<class T> void f(T x, T y = ydef(T())), T z = zdef(T()));

class A { }

A zdef(A);

void g(A a, A b, A c) {
    f(a, b, c);  // no default argument instantiation
    f(a, b);    // default argument z = zdef(T()) instantiated
}
```

Draft
f(a);         // ill-formed; ydef is not declared
}

— end example ]

[ Note: 14.6.4.1 defines the point of instantiation of a template specialization. — end note ]

14 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. [ Example:

```cpp
template<class T> class X {
    X<T>* p;       // OK
    X<T*> a;       // implicit generation of X<T> requires
                   // the implicit instantiation of X<T*> which requires
                   // the implicit instantiation of X<T**> which ...
};
```

— end example ]

14.7.2 Explicit instantiation [temp.explicit]

1 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. An explicit instantiation of a function template shall not use the `inline` or `constexpr` specifiers.

2 The syntax for explicit instantiation is:

```
explicit-instantiation:
    extern,opt  template declaration
```

There are two forms of explicit instantiation: an explicit instantiation definition and an explicit instantiation declaration. An explicit instantiation declaration begins with the `extern` keyword.

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 or, if that namespace is `inline` (7.3.1), any namespace from its enclosing namespace set. [ Note: regarding qualified names in declarators, see 8.3. — end note ] [ Example:

```cpp
template<class T> class Array { void mf(); };  
template class Array<char>;  
template void Array<int>::mf();

template<class T> void sort(Array<T>& v) { /* ... */ }
template void sort(Array<char>&);             // argument is deduced here
```

Draft
namespace N {
    template<class T> void f(T&) { }
}

template void N::f<int>(int&);

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:

namespace N {
    template<class T> class Y { void mf() { } }
}

template class Y<int>; // error: class template Y not visible
    // in the global namespace

using N::Y;

template class Y<int>; // 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);
7 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.

8 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.

9 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 ]

10 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.

11 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 ]

12 An explicit instantiation does not constitute a use of a default argument, so default argument instantiation is not done. [ Example:
    
    ```
    char* p = 0;
    template<class T> T g(T = &p);
    template int g<int>(int); // OK even though &p isn't an int.
    ```
    — end example ]

### 14.7.3 Explicit specialization

1 An explicit specialization of any of the following:

- non-deleted function template
- class template
- non-deleted member function of a class template
14.7 Template instantiation and specialization

— 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 \texttt{template<>}; that is:

\begin{verbatim}
explicit-specialization:
  template <> declaration
\end{verbatim}

\texttt{Example:}

\begin{verbatim}
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*>&) ;
\end{verbatim}

Given these declarations, \texttt{stream<char>} will be used as the definition of streams of \texttt{char}s; other streams will be handled by class template specializations instantiated from the class template. Similarly, \texttt{sort<char*}> will be used as the sort function for arguments of type \texttt{Array<char*>}; other \texttt{Array} types will be sorted by functions generated from the template. \textit{— end example}

2 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, nearest enclosing namespace of the template, or, if the namespace is inline (7.3.1), any namespace from its enclosing namespace set. Such a declaration may also be a definition. If the declaration is not a definition, the specialization may be defined later (7.3.1.2).

3 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. [\textit{Note:} a declaration, but not a definition of the template is required. \textit{— 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. [\textit{Example:}]

\begin{verbatim}
template<> class X<int> { /* ... */ } ; // error: X not a template
template<class T> class X ;
template<> class X<char*> { /* ... */ } ; // OK: X is a template
\end{verbatim}

\textit{— end example}

4 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
}
```

—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<>(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
#include <vector>

// Primary class template vector
export template<class T> void f(t) {
    std::vector<T> vec; // should match the specialization
    // ... */
}

// Explicit specialization of vector for vector<B>
namespace std {
    template<class T> class vector<B> { /* ... */ }; 
}

// OK: specialization in same namespace
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 #1
}
```

A template explicit specialization is in the scope of the namespace in which the template was defined. Example:

```cpp
namespace N {
    template<class T> class X { /* ... */ }; 
    template<class T> class Y { /* ... */ }; 

    template<class T> class X<int> { /* ... */ }; // OK: specialization
    // in same namespace

    template<class T> class Y<double>; // forward declare intent to
    // specialize for double
}
```

Draft
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:

```cpp
template<class T> class X; //X is a class template
template<> class X<int>; // OK: specialization for X<int>

X<int>* p; // OK: pointer to declared 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:

```cpp
template<class T> class Array { /* */ }; // explicit specialization for 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) { /* */ } // OK: inline

template<> inline void f<>(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;

Draft
This is a declaration regardless of whether X can be default initialized (8.5). — end note]

A member or a member template of a class template may be explicitly specialized for a given implicit instantiation of the class template, even if the member or member template is defined in the class template definition. An explicit specialization of a member or member template is specified using the syntax for explicit specialization. [Example:

```cpp
template<class T> struct A {
    void f(T);
    template<class X1> void g1(T, X1);
    template<class X2> void g2(T, X2);
    void h(T) { }
};

// specialization
template<> void A<int>::f(int);

// out of class member template definition
template<class T> template<class X1> void A<T>::g1(T, X1) { }

// member template specialization
template<> template<class X1> void A<int>::g1(int, X1);

// member template specialization
template<> template<class X1> void A<int>::g2(char, X1); // X1 deduced as char

// member specialization even if defined in class definition
template<> void A<int>::h(int) { }

— end example ]

A member or a member template may be nested within many enclosing class templates. In an explicit specialization for such a member, the member declaration shall be preceded by a template<> for each enclosing class template that is explicitly specialized. [Example:

```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) { }
template <class Y> template <>
void A<Y>::B<double>::mf2() { } // ill-formed; B<double> is specialized but
// its enclosing class template A is not
```

— end example ]

20 A specialization of a member function template or member class template of a non-specialized class template is itself a template.

21 An explicit specialization declaration shall not be a friend declaration.

22 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;

— 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 [temp.fct.spec]

1 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.

2 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;
}
```

Draft
14.8 Function template specializations

void g(int a, char* b) {
    f(&a);   // calls f<int>(int*)
    f(&b);   // calls f<char*>(char**)  
}

Here f<int>(int*) has a static variable s of type int and f<char*>(char**) has a static variable s of type char*.
— end example ]

14.8.1 Explicit template argument specification [temp.arg.explicit]

1 Template arguments can be specified when referring to a function template specialization by qualifying the function template name with the list of template-arguments in the same way as template-arguments are specified in uses of a class template specialization. [Example:  

```c
template<class T> void sort(Array<T>& v);
void f(Array<dcomplex>& cv, Array<int>& ci) {
    sort<dcomplex>(cv);   // sort(Array<dcomplex>&)
    sort<int>(ci);        // sort(Array<int>&)
}
```

and

```c
template<class U, class V> U convert(V v);
void g(double d) {
    int i = convert<int,double>(d);   // int convert(double)
    char c = convert<char,double>(d); // char convert(double)
}
```

— end example ]

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-arguments may be omitted
   from the list of explicit template-arguments. 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:  

Draft
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); // X 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); // X is deduced to be double, Z is deduced to an empty sequence
  f<void>(g<int, bool>); // Y for outer f 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<>(1); // uses #1
```

— end note ]

5 Template arguments that are present shall be specified in the declaration order of their corresponding template-parameters. The template argument list shall not specify more template-arguments than there are corresponding template-parameters unless one of the template-parameters is a template parameter pack. [ Example:

```cpp
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); // Z is deduced to be double
  f<int,char*>("aa",3.0); // error: X cannot be deduced
  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 ]

6 Implicit conversions (clause 4) will be performed on a function argument to convert it to the type of the corresponding function parameter if the parameter type contains no template-parameters that participate in template argument deduction. [ Note: template parameters do not participate in template argument deduction if they are explicitly specified. For example,

```cpp
template<class T> void f(T);
```

class Complex {

Draft
Complex(double);
};

void g() {
    f<Complex>(1);    // OK, means f<Complex>(Complex(1))
}

— end note ]

7 [ Note: because the explicit template argument list follows the function template name, and because conversion member
    function templates and constructor member function templates are called without using a function name, there is no way
    to provide an explicit template argument list for these function templates. — end note ]

8 [ Note: For simple function names, argument dependent lookup (3.4.2) applies even when the function name is not
    visible within the scope of the call. This is because the call still has the syntactic form of a function call (3.4.1). But
    when a function template with explicit template arguments is used, the call does not have the correct syntactic form
    unless there is a function template with that name visible at the point of the call. If no such name is visible, the call is
    not syntactically well-formed and argument-dependent lookup does not apply. If some such name is visible, argument
    dependent lookup applies and additional function templates may be found in other namespaces. [ Example:

    namespace A {
        struct B {};
        template<int X> void f(B);
    }
    namespace C {
        template<class T> void f(T t);
    }
    void g(A::B b) {
        f<3>(b);        // ill-formed: not a function call
        A::f<3>(b);     // well-formed
        C::f<3>(b);     // ill-formed; argument dependent lookup
                        // applies only to unqualified names
        using C::f;
        f<3>(b);        // well-formed because C::f is visible; then
                        // A::f is found by argument dependent lookup
    }

    — end example ] — end note ]

9 Template argument deduction can extend the sequence of template arguments corresponding to a template parameter
    pack, even when the sequence contains explicitly specified template arguments. [ Example:

    template<class ... Types> void f(Types ... values);

    void g() {
        f<int*, float*>(0, 0, 0);  // Types is deduced to the sequence int*, float*, int
    }

    — end example ]
14.8.2 Template argument deduction

When a function template specialization is referenced, all of the template arguments shall have values. The values can be explicitly specified or, in some cases, be deduced from the use or obtained from default template-arguments.  

```c
void f(Array<dcomplex>& cv, Array<int>& ci) {
    sort(cv);    // calls sort(Array<dcomplex>&)
    sort(ci);    // calls sort(Array<int>&)
}
```

and

```c
void g(double d) {
    int i = convert<int>(d);    // calls convert<int,double>(double)
    int c = convert<char>(d);   // calls convert<char,double>(double)
}
```

— end example ]

2 When an explicit template argument list is specified, the template arguments must be compatible with the template parameter list and must result in a valid function type as described below; otherwise type deduction fails. Specifically, the following steps are performed when evaluating an explicitly specified template argument list with respect to a given function template:

— The specified template arguments must match the template parameters in kind (i.e., type, non-type, template). There must not be more arguments than there are parameters unless at least one parameter is a template parameter pack, and there shall be an argument for each non-pack parameter. Otherwise, type deduction fails.

— 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 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 instantiate a pack expansion containing multiple parameter packs of differing lengths.

— Attempting to create an array with an element type that is void, a function type, a reference type, or an abstract class type, or attempting to create an array with a size that is zero or negative.  

```c
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.  

```c
template <class T> int f(typename T::B*);
int i = f<int>(0);
```
14.8 Function template specializations

— 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:

```cpp
template <int I> struct X { };
template <template <class T> class> struct Z { };
template <class T> void f(typename T::*Y){}
template <class T> void g(X<T::N>*){}
template <class T> void h(Z<T::template TT>*){}
struct A { }; struct B { int Y; }; struct C {
    typedef int N; }
struct D {
    typedef int TT; }
```]

```cpp
int main() {
    // Deduction fails in each of these cases:
    f<A>(0); // A does not contain a member Y
    f<B>(0); // The Y member of B is not a type
    g<C>(0); // The N member of C is not a non-type
    h<D>(0); // The TT member of D is not a template
}
— end example }

— Attempting to create a pointer to reference type.
— Attempting to create a reference to void.
— Attempting to create “pointer to member of T” when T is not a class type. [Example:

```cpp
template <class T> int f(int T::*); int i = f<int>(0);
— end example }

— Attempting to give an invalid type to a non-type template parameter. [Example:

```cpp
template <class T, T> struct S {}; template <class T> int f(S<T, T>()*);
struct X {}; int i0 = f<X>(0);
```]
— end example ]

— Attempting to perform an invalid conversion in either a template argument expression, or an expression used in the function declaration. [Example:

```
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.

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:

```
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')
```

Draft
14.8 Function template specializations

```cpp
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.

6 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) {
    const int z = x;
    f(x, y, z); // Types is deduced to int, float, const int
    g(x, y, z); // T1 is deduced to int, Types is deduced to float, int
}
```

— end example ]

2 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.

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:

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:

// 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]
// 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

Example:

// The overload set contains a template, causing the second function
// parameter to be a non-deduced context.
template <class T> int f(T, T (*p)(T));
char g(char);
template <class T> T g(T);
int i = f(1, g);    // calls f(int, int (*)(int))

— end example

14.8.2.2 Deducing template arguments taking the address of a function template  [temp.deduct.funcaddr]

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  [temp.deduct.conv]

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.

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.

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.

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.

5 These alternatives are considered only if type deduction would otherwise fail. If they yield more than one possible deduced `A`, the type deduction fails.

6 When the deduction process requires a qualification conversion for a pointer or pointer to member type as described above, the following process is used to determine the deduced template argument values:

   If `A` is a type
   
   \[
   cv_{1,0} \text{ “pointer to …” } cv_{1,n-1} \text{ “pointer to” } cv_{1,n} T1
   \]
   
   and `P` is a type
   
   \[
   cv_{2,0} \text{ “pointer to …” } cv_{2,n-1} \text{ “pointer to” } cv_{2,n} T2
   \]
   
   The `cv-unqualified T1` and `T2` are used as the types of `A` and `P` respectively for type deduction. [Example:

   ```
   struct A {
       template <class T> operator T***();
   };
   A a;
   const int * const * const * p1 = a; // T is deduced as int, not const int
   ```
   
   — end example ]

14.8.2.4 Deducing template arguments during partial ordering [temp.deduct.partial]

1 Template argument deduction is done by comparing certain types associated with the two function templates being compared.

2 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.

3 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.

4 Each type from the parameter template and the corresponding type from the argument template are used as the types of `P` and `A`.

5 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.

6 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.

7 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 $A$.

8 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.

9 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.

10 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.

11 In most cases, all template parameters must have values in order for deduction to succeed, but for partial ordering purposes a template parameter may remain without a value provided it is not used in the types being used for partial ordering. [Note: a template parameter used in a non-deduced context is considered used. — end note] [Example:

```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]

12 [Note: Partial ordering of function templates containing template parameter packs is independent of the number of deduced arguments for those template parameter packs. — end note] [Example:

```cpp
template<class ...> struct Tuple { };  // calls #1
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
```

```cpp
g(Tuple<>());  // calls #1
```

```cpp
g(Tuple<int, float>());  // calls #2
```

```cpp
g(Tuple<int, float&>());  // calls #3
```

```cpp
g(Tuple<int>());  // calls #3
```

— end example]
14.8.2.5 Deducing template arguments from a type

Template arguments can be deduced in several different contexts, but in each case a type that is specified in terms of template parameters (call it P) is compared with an actual type (call it A), and an attempt is made to find template argument values (a type for a type parameter, a value for a non-type parameter, or a template for a template parameter) that will make P, after substitution of the deduced values (call it the deduced A), compatible with A.

In some cases, the deduction is done using a single set of types P and A, in other cases, there will be a set of corresponding types P and A. Type deduction is done independently for each P/A pair, and the deduced template argument values are then combined. If type deduction cannot be done for any P/A pair, or if for any pair the deduction leads to more than one possible set of deduced values, or if different pairs yield different deduced values, or if any template argument remains neither deduced nor explicitly specified, template argument deduction fails.

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<int>) 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<T>::B<T2>, both T and T2 are non-deduced. Likewise, if a type is specified as A<int>::X<T>, I, J, and T are non-deduced. If a type is specified as void f(typename A<T>::B, A<T>), the T in A<T>::B is non-deduced but the T in A<T> is deduced. —end example]
7  [Example: Here is an example in which different parameter/argument pairs produce inconsistent template argument
deductions:

```cpp
template<class T> void f(T x, T y) { /* ... */ }
struct A { /* ... */ };
struct B : A { /* ... */ };
void g(A a, B b) {
  f(a, b); // error: T could be A or B
  f(b, a); // error: T could be A or B
  f(a, a); // OK: T is A
  f(b, b); // OK: T is B
}
```

Here is an example where two template arguments are deduced from a single function parameter/argument pair. This
can lead to conflicts that cause type deduction to fail:

```cpp
template <class T, class U> void f( T (*)( T, U, U ) );
int g1( int, float, float);
char g2( int, float, float);
int g3( int, char, float);
void r() {
  f(g1); // OK: T is int and U is float
  f(g2); // error: T could be char or int
  f(g3); // error: U could be char or float
}
```

Here is an example where a qualification conversion applies between the argument type on the function call and the
deduced template argument type:

```cpp
template<class T> void f(const T*) { }
int *p;
void s() {
  f(p); //f(const int*)
}
```

Here is an example where the template argument is used to instantiate a derived class type of the corresponding function
parameter type:

```cpp
template <class T> struct B {};
template <class T> struct D : public B<T> {};
struct D2 : public B<int> {};
template <class T> void f(B<T>&){}
void t() {
  D<int> d;
  D2 d2;
  f(d); // calls f(B<int>&)
  f(d2); // calls f(B<int>&)
}
```
A template type argument \(T\), a template template argument \(TT\) or a template non-type argument \(i\) can be deduced if \(P\) and \(A\) have one of the following forms:

\[
\begin{align*}
T & \quad \text{class template} \\
T \text{ cv-list} & \\
T* & \\
T & \\
T& & \\
T&& & \\
T[\text{integer-constant}] & \\
template-name<&T> & \text{(where template-name refers to a class template)} \\
type(T) & \\
T() & \\
T(T) & \\
T type::* & \\
type T::* & \\
T T::* & \\
T (type::*)() & \\
type (T::*)() & \\
type (type::*)(T) & \\
type (T::*)(T) & \\
T (type::*)(T) & \\
T (T::*)(T) & \\
T (T::*)(T) & \\
type[i] & \\
template-name<&i> & \text{(where template-name refers to a class template)} \\
TT<&T> & \\
TT<&i> & \\
TT<&> &
\end{align*}
\]

where \((T)\) represents a parameter-type-list where at least one parameter type contains a \(T\), and \(()\) represents a parameter-type-list where no parameter type contains a \(T\). Similarly, \(<T>\) represents template argument lists where at least one argument contains a \(T\), \(<i>\) represents template argument lists where at least one argument contains an \(i\) and \(<&>\) represents template argument lists where no argument contains a \(T\) or an \(i\).

If \(P\) has a form that contains \(<T>\) or \(<i>\), then each argument \(P_i\) of the respective template argument list \(P\) is compared with the corresponding argument \(A_i\) of the corresponding template argument list of \(A\). If the template argument list of \(P\) contains a pack expansion that is not the last template argument, the entire template argument list is a non-deduced context. If \(P_i\) is a pack expansion, then the pattern of \(P_i\) is compared with each remaining argument in the template argument list of \(A\). Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by \(P_i\).

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. \[\text{Note: A function parameter pack can only occur at the end of a parameter-declaration-list (8.3.5).} \]
14.8 Function template specializations

These forms can be used in the same way as \( T \) is for further composition of types. [Example:

\[
X\text{<int> (*)(char[6])}
\]

is of the form

\[
\text{template-name<T> (*)(type[i])}
\]

which is a variant of

\[
type (*)(T)
\]

where type is \( X\text{<int>} \) and \( T \) is \( \text{char[6]} \). —end example]  

Template arguments cannot be deduced from function arguments involving constructs other than the ones specified above.

A template type argument cannot be deduced from the type of a non-type template-argument. [Example:

\[
\text{template<class T, T i> void f(double a[10][i]);}
\]

\[
\text{int v[10][20];}
\]

\[
f(v); \quad \text{// error: argument for template-parameter } T \text{ cannot be deduced}
\]

—end example]

[Note: except for reference and pointer types, a major array bound is not part of a function parameter type and cannot be deduced from an argument:

\[
\text{template<int i> void f1(int a[10][i]);}
\]

\[
\text{template<int i> void f2(int a[i][20]);}
\]

\[
\text{template<int i> void f3(int (&a)[i][20]);}
\]

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

\[
\text{int v[10][20];}
\]

\[
f1(v); \quad \text{// OK: i deduced to be 20}
\]

\[
f1<20>(v); \quad \text{// OK}
\]

\[
f2(v); \quad \text{// error: cannot deduce template-argument i}
\]

\[
f2<10>(v); \quad \text{// OK}
\]

\[
f3(v); \quad \text{// OK: i deduced to be 10}
\]

}]

If, in the declaration of a function template with a non-type template parameter, the non-type template parameter is used in a subexpression in the function parameter list, the expression is a non-deduced context as specified above. [Example:

\[
\text{template <int i> class A { /* ... */ };}
\]

\[
\text{template <int i> void g(A<i+1>);}
\]

\[
\text{template <int i> void f(A<i>, A<i+1>);}
\]

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

\[
\text{A<1> a1;}
\]

\[
\text{A<2> a2;}
\]

Draft
g(a1);  // error: deduction fails for expression i+1
g<0>(a1); // OK
f(a1, a2); // OK
}

— end example — end note [ Note: template parameters do not participate in template argument deduction if they are used only in non-deduced contexts. For example,

```cpp
template<int i, typename T>
T deduce(typename A<T>::X x, // T is not deduced here
T t, // but T is deduced here
type B<i>::Y y);  // i is not deduced here
A<int> a;
B<77> b;

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.135) [ 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:

135) 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.

Draft
template<class T> void f(void(*)(T,int));
template<class T> void foo(T,int);
void g(int,int);
void g(char,int);
void h(int,int,int);
void h(char,int);

int m() {
    f(&g);        // error: ambiguous
    f(&h);        // OK: void h(char,int) is a unique match
    f(&foo);      // error: type deduction fails because foo is a template
}

— end example —

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 —

The template-argument corresponding to a template template-parameter is deduced from the type of the template-argument of a class template specialization used in the argument list of a function call. [Example:

    template <template <class T> class X> struct A { };  
    template <template <class T> class X> void f(A<X>) { }  
    template<class T> struct B { };  
    A<B> ab;                          
    f(ab);                            // calls f(A<B>)

— end example —

[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

Draft
Y<int&, float&, double&> y2; // uses partial specialization; T is int&, Types contains float, double
Y<int, float, double> y3; // uses primary template; Types contains int, float, double
int fv = f(g); // OK; Types contains int, float

— end example ]

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:

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-arguments (deduced and/or explicit) are used to synthesize the declaration of a single function template specialization which is added to the candidate functions set to be used in overload resolution. If, for a given function template, argument deduction fails, no such function is added to the set of candidate functions for that template. The complete set of candidate functions includes all the synthesized declarations and all of the non-template overloaded functions of the same name. The synthesized declarations are treated like any other functions in the remainder of overload resolution, except as explicitly noted in 13.3.3.136)

[ Example:

template<T> T max(T a, T b) { return a>b?a:b; }

void f(int a, int b, char c, char d) {
    int m1 = max(a,b); // max(int, int)
    char m2 = max(c,d); // max(char, char)
    int m3 = max(a,c); // error: cannot generate max(int,char)
}

136) 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.
2 Adding the non-template function

    int max(int,int);

    to the example above would resolve the third call, by providing a function that could be called for max(a,c)
    after using the standard conversion of char to int for c.

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 ]
Chapter 15  Exception handling

1 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.

   try-block:
      try compound-statement handler-seq
   function-try-block:
      try ctor-initializer_opt compound-statement handler-seq
   handler-seq:
      handler handler-seq_opt
   handler:
      catch ( exception-declaration ) compound-statement
   exception-declaration:
      type-specifier-seq declarator
      type-specifier-seq abstract-declarator
      type-specifier-seq
      ...
   throw-expression:
      throw assignment-expression_opt

2 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]

3 A *goto* or *switch* statement shall not be used to transfer control into a try block or into a handler. [Example:

   void f() {
      goto 11;    // Ill-formed
      goto 12;    // Ill-formed
      try {
         goto 11;  // OK
         goto 12;  // Ill-formed
         11: ;
      } catch (...) {
         12: ;
      goto 11;  // Ill-formed
      goto 12;  // OK
   }


15.1 Throwing an exception

Throwing an exception transfers control to a handler. An object is passed and the type of that object determines which handlers can catch it. [Example:

```cpp
throw "Help!";
```

can be caught by a handler of `const char*` type:

Draft
try {
    // ...
}
catch(const char* p) {
    // handle character string exceptions here
}

and

class Overflow {
public:
    Overflow(char,double,double);
};

void f(double x) {
    throw Overflow('+',x,3.45e107);
}

...
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
    // respond (partially) to exception
    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, base classes and non-variant members, 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.

3 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.
A handler of type “array of T” or “function returning T” is adjusted to be of type “pointer to T” or “pointer to function returning T”, respectively.

A handler is a match for an exception object of type E if

- 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
- the handler is a pointer or pointer to member type and E is std::nullptr_t.

Note: a throw-expression whose operand is an integral constant expression of integer type that evaluates to zero does not match a handler of pointer or pointer to member type. — end note

Example:

```cpp
class Matherr { /* ... */ virtual void vf(); };
class Overflow: public Matherr { /* ... */ };
class Underflow: public Matherr { /* ... */ };
class Zerodivide: public Matherr { /* ... */ };  

void f() {
try {
  g();
} catch (Overflow oo) {
    // ...
} catch (Matherr mm) {
    // ...
}
}
```

Here, the Overflow handler will catch exceptions of type Overflow and the Matherr handler will catch exceptions of type Matherr and of all types publicly derived from Matherr including exceptions of type Underflow and Zerodivide. — end example

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.

8 The exception with the most recently activated handler that is still active is called the **currently handled exception**.

9 If no matching handler is found, the function std::terminate() is called; whether or not the stack is unwound before this call to std::terminate() is implementation-defined (15.5.1).

10 Referring to any non-static member or base class of an object in the handler for a *function-try-block* of a constructor or destructor for that object results in undefined behavior.

11 The 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 base classes and non-variant members of an object shall be destroyed before entering the handler of a function-try-block of a destructor for that object** (12.4).

12 The scope and lifetime of the parameters of a function or constructor extend into the handlers of a *function-try-block*.

13 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().

14 If a return statement appears in a handler of the *function-try-block* of a constructor, the program is ill-formed.

15 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).

16 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.

17 The copy constructor and destructor associated with the object shall be accessible even if the copy operation is elided (12.8).

18 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 )
```

Draft
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. [Example:

```c
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-ids. 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-ids. 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-ids as other declarations of that function. A diagnostic is required only if the sets of type-ids 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. [Example:

```c
struct B {
    virtual void f() throw (int, double);
    virtual void g();
};

struct D: B {
    void f();
    // ill-formed
    void g() throw (int);
    // OK
};
```

The declaration of D::f is ill-formed because it allows all exceptions, whereas B::f allows only int and double.
— end example ] A similar restriction applies to assignment to and initialization of pointers to functions, pointers to member functions, and references to functions: the target entity shall allow at least the exceptions allowed by the source value in the assignment or initialization. [Example:

```c
class A { /* ... */
void (*pf1)(); // no exception specification
void (*pf2)() throw(A);

void f() {
    pf1 = pf2; // OK: pf1 is less restrictive
```
15.4 Exception specifications

```cpp
pf2 = pf1;       // error: pf2 is more restrictive
```

— end example ]

5 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.

6 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:
```cpp
class X { };  
class Y { };  
class Z: public X { };  
class W { };  

void f() throw (X, Y) {
   int n = 0;
   if (n) throw X();       // OK
   if (n) throw Z();       // also OK
   throw W();              // will call std::unexpected()
}
```
— end example ]

10 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().

11 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 ]

12 A function with no exception-specification allows all exceptions. A function with an empty exception-specification, throw(), does not allow any exceptions.

Draft
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 \( \text{A::~A()} \) or \( \text{B::~B()} \) were virtual, \( \text{D::~D()} \) would not be as restrictive as that of \( \text{A::~A} \), 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 [except.special]

The exception handling mechanism relies on two functions, `std::terminate()` and `std::unexpected()`, for coping with errors related to the exception handling mechanism itself (18.7).

15.5.1 The `std::terminate()` function [except.terminate]

In the following situations exception handling must be abandoned for less subtle error handling techniques:

1. 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,\(^{137}\)
2. when the exception handling mechanism cannot find a handler for a thrown exception (15.3), or
3. when the destruction of an object during stack unwinding (15.2) exits using an exception, or
4. when construction or destruction of a non-local object with static storage duration exits using an exception (3.6.2), or
5. when execution of a function registered with `std::atexit` exits using an exception (18.4), or

\(^{137}\) 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.
15.6 Exceptions and access

Exception handling

— 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 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.138) A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro.

138) 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-listopt ) replacement-list new-line
  # define   identifier lparen . . . ) replacement-list new-line
  # define   identifier lparen identifier-list, . . . ) replacement-list new-line
  # undef    identifier new-line
  # line     pp-tokens new-line
  # error    pp-tokensopt new-line
  # pragma   pp-tokensopt new-line
  # new-line

text-line:
  pp-tokensopt new-line

non-directive:
  pp-tokensopt new-line

lparen:
  a ( character not immediately preceded by white-space

identifier-list:
  identifier
  identifier-list , identifier

replacement-list:
  pp-tokensopt

pp-tokens:
  preprocessing-token
  pp-tokens preprocessing-token

new-line:
  the new-line character

---

2 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.

3 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.

4 The only white-space characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the introducing # preprocessing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other white-space characters in translation phase 3).

5 The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called preprocessing, because conceptually they occur before translation of the resulting translation unit.

6 The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

[Example: In:
  #define EMPTY
  EMPTY  #  include <file.h>
the sequence of preprocessing tokens on the second line is not a preprocessing directive, because it does not begin with a # at the start of translation phase 4, even though it will do so after the macro 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;\(^\text{139}\) and it may contain unary operator expressions of the form

\[
\text{defined identifier}
\]

or

\[
\text{defined ( identifier )}
\]

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

\[
\text{# if constant-expression new-line group,opt}
\]

\[
\text{# elif constant-expression new-line group,opt}
\]

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 defined unary operator), just as in normal text. If the token defined is generated as a result of this replacement process or use of the defined unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined. After all replacements due to macro expansion and the defined unary operator have been performed, all remaining identifiers and keywords, except for true and false, are replaced with the pp-number 0, and then each preprocessing token is converted into a token. The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 5.19 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, intmax_t or 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.\(^\text{141}\) Also, whether a single-character character literal may have a negative value is implementation-defined. Each subexpression with type bool is subjected to integral promotion before processing continues.

\(^\text{139}\) 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.

\(^\text{140}\) 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.

\(^\text{141}\) 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

```
#define identifier new-line group opt
#if defined identifier
#else
#endif
```

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent to `#if defined identifier` and `#if !defined identifier` respectively.

Each directive’s condition is checked in order. If it evaluates to false (zero), the group that it controls is skipped; directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; the rest of the directives’ preprocessing tokens are ignored, as are the other preprocessing tokens in the group. Only the first group whose control condition evaluates to true (nonzero) is processed. If none of the conditions evaluates to true, and there is a `#else` directive, the group controlled by the `#else` is processed; lacking a `#else` directive, all the groups until the `#endif` are skipped.

### 16.2 Source file inclusion

1. A `#include` directive shall identify a header or source file that can be processed by the implementation.
2. A preprocessing directive of the form

   ```
   # include <h-char-sequence> new-line
   ```

   searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the `<` and `>` delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.
3. A preprocessing directive of the form

   ```
   # include "q-char-sequence" new-line
   ```

   causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the `"` delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

   ```
   # include <h-char-sequence> new-line
   ```

   with the identical contained sequence (including `>` characters, if any) from the original directive.
4. A preprocessing directive of the form

   ```
   # include pp-tokens new-line
   ```

   (that does not match one of the two previous forms) is permitted. The preprocessing tokens after `include` in the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined. The method by which a sequence of preprocessing tokens between a `<` and a `>` preprocessing token pair or a pair of `"` characters is combined into a single header name preprocessing token is implementation-defined.

---

142) 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.

143) 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.
The implementation provides unique mappings for sequences consisting of one or more nondigits or digits (2.10) followed by a period (.) and a single nondigit. The first character shall not be a digit. The implementation may ignore 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

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

```cpp
#define identifier replacement-list new-line
```

defines an object-like macro that causes each subsequent instance of the macro name\(^{144}\) to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive.\(^{145}\) The replacement list is then rescanned for more macro names as specified below.

A preprocessing directive of the form

```cpp
#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 [cpp.subst]

After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place.

\(^{144}\) 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.

\(^{145}\) 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

1. Each # preprocessing token in the replacement list for a function-like macro shall be followed by a parameter as the next preprocessing token in the replacement list.

2. 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

1. A ## preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

2. If, in the replacement list of a function-like macro, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument’s preprocessing token sequence; however, if an argument consists of no preprocessing tokens, the parameter is replaced by a placemarker preprocessing token instead.

3. For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemarker preprocessing tokens are handled specially; concatenation of two 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
```

Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.
// char p[] = "x ## y";

The expansion produces, at various stages:

```c
join(x, y)
in_between(x hash_hash y)
in_between(x ## y)
mkstr(x ## y)
"x ## y"
```

In other words, expanding hash_hash produces a new token, consisting of two adjacent sharp signs, but this new token is not the ## operator. — end example]

16.3.4 Rescanning and further replacement [cpp.rescan]

1 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.

2 If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the

source file’s preprocessing tokens), it is not replaced. Furthermore, if any nested replacements encounter the name of the

macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available

for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token

would otherwise have been replaced.

3 The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive

even if it resembles one, but all pragma unary operator expressions within it are then processed as specified in 16.9

down.

16.3.5 Scope of macro definitions [cpp.scope]

1 A macro definition lasts (independent of block structure) until a corresponding #undef directive is encountered or (if none is encountered) until the end of the translation unit. Macro definitions have no significance after translation phase

4.

2 A preprocessing directive of the form

```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.

3 [ Note: The simplest use of this facility is to define a “manifest constant,” as in

```c
#define TABSIZE 100
int table[TABSIZE];
```

4 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
(f)^m(m);
p() i[q()] = { q(1), r(2,3), r(4,), r(5), r(,) }; char c[2][6] = { str(hello), str() };
```

results in

```c
f(2 * (y+1)) + f(2 * (f(2 * (z[0]))))) % f(2 * (0)) + t(1);
f(2 * (2+(3,4)-0,1)) | f(2 * (~5)) & f(2 * (0,1))~m(0,1);
int i[] = { 1, 23, 4, 5, }; char c[2][6] = { "hello", "" };
```

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"

debug(1, 2);

fputs(str(strncmp("abc\0d", "abc", '\4') // this goes away
   == 0) str(: \0\n), s);
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW)
```

results in
or, after concatenation of the character string literals,

```
printf("x1= %d, x2= %s", x1, x2);

fgets("strncmp("abc\0d", "abc", '\4') == 0" : @

"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,{,}), t(),(11,), t({,}12), t({,})
```

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 )(/\n\
   a /* other stuff on this line */

/* end note */
```

9 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 ]
```

10 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__))
```

```
directive
```

Draft
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

   ```c
   # line digit-sequence new-line
   ```

   causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer). If the digit sequence specifies zero or a number greater than 2147483647, the behavior is undefined.

4. A preprocessing directive of the form

   ```c
   # 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

   ```c
   # line pp-tokens new-line
   ```

   (that does not match one of the two previous forms) is permitted. The preprocessing tokens after `line` on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined; otherwise, the result is processed as appropriate.

### 16.5 Error directive

1. A preprocessing directive of the form

   ```c
   # error pp-tokens opt new-line
   ```

   causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens, and renders the program ill-formed.

### 16.6 Pragma directive

1. A preprocessing directive of the form

   ```c
   # pragma pp-tokens opt new-line
   ```

Draft
16.7 Null directive

A preprocessing directive of the form

```
# 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.\(^{147}\)
- `_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 `yyyymm`L (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.

\(^{147}\) 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]
Chapter 17   Library introduction [library]

1 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.

2 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).

3 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.

4 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.

5 The containers, iterators, and algorithms provide a C++ program with access to a subset of the most widely used algorithms and data structures.

6 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.

7 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.

8 The atomic components allow more fine-grained concurrent access to shared data than is possible with locks.

9 This library also makes available the facilities of the Standard C library, suitably adjusted to ensure static type safety.

10 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.

11 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 [definitions]

17.1.1 [defs.arbitrary.stream]
arbitrary-positional stream
a stream (described in clause 27) that can seek to any integral position within the length of the stream. Every arbitrary-positional stream is also a repositional stream.

17.1.2 blocked thread
[defns.blocked]
a thread that is waiting for some condition (other than the availability of a processor) to be satisfied before it can continue execution.\(^{148}\) As a verb, to block is to place a thread in the blocked state, and to unblock is to place a thread in the unblocked state.

17.1.3 character
[defns.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.

17.1.4 character container type
[defns.character.container]
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.

17.1.5 comparison function
[defns.comparison]
an operator function (13.5) for any of the equality (5.10) or relational (5.9) operators.

17.1.6 component
[defns.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.

17.1.7 deadlock
[defns.deadlock]
two or more threads are unable to continue execution because each is blocked waiting for one or more of the others to satisfy some condition.

17.1.8 default behavior
[defns.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.

\(^{148}\) This definition is taken from POSIX.
17.1.9 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).

17.1.10 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.

17.1.11 modifier function
A class member function (9.3), other than constructors, assignment, or destructor, that alters the state of an object of the class.

17.1.12 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.

17.1.13 NTCTS
A sequence of values that have character type, that precede the terminating null character type value charT().

17.1.14 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).

17.1.15 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).

17.1.16 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).
17.2 Additional definitions

17.1.17 repositional stream
a stream (described in clause 27) that can seek only to a position that was previously encountered.

17.1.18 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.

17.1.19 reserved function
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.

17.1.20 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.

17.1.21 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.

17.1.22 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 Draft
407 Library introduction

17.3 Method of description (Informative)

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.

<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>

Each Library clause contains the following elements, as applicable:149)

— Summary
— Requirements
— Detailed specifications
— References to the Standard C library

17.3.1.1 Summary

The Summary provides a synopsis of the category, and introduces the first-level subclauses. Each subclause also provides a summary, listing the headers specified in the subclause and the library entities provided in each header.

Paragraphs labelled “Note(s):” or “Example(s):” are informative, other paragraphs are normative.

The summary and the detailed specifications are presented in the order:

— Macros
— Values
— Types
— Classes

149) 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)  

— Functions
— Objects

17.3.1.2 Requirements

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

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.\(^{150}\)

17.3.1.3 Specifications

The detailed specifications each contain the following elements:\(^{151}\)

— 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):\(^{152}\)

— Constructor(s) and destructor

\(^{150}\) Although in some cases the code given is unambiguously the optimum implementation.

\(^{151}\) The form of these specifications was designed to follow the conventions established by existing C++ library vendors.

\(^{152}\) 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.
3 Descriptions of function semantics contain the following elements (as appropriate):\(^ {153}\)

- **Requires:** the preconditions for calling the function
- **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
- **Remarks:** additional semantic constraints on the function

4 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.

5 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

Paragaphs 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.\(^ {154}\) 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.

---

\(^ {153}\) 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.

\(^ {154}\) Examples from 20.1 include: EqualityComparable, LessThanComparable, CopyConstructable, etc. Examples from 24.1 include: InputIterator, ForwardIterator, Function, Predicate, etc.
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.\(^{(155)}\)

The enumerated type enumerated can be written:

```c
enum enumerated {
    V0, V1, V2, V3, .....;
}
```

```c
static const enumerated C0 (V0);
static const enumerated C1 (V1);
static const enumerated C2 (V2);
static const enumerated C3 (V3);
......
```

Here, the names C0, C1, 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:

```c
enum bitmask {
    V0 = 1 << 0, V1 = 1 << 1, V2 = 1 << 2, V3 = 1 << 3, .....;
};
```

```c
static const bitmask C0(V0);
static const bitmask C1(V1);
static const bitmask C2(V2);
static const bitmask C3(V3);
......
```

// For exposition only.
// int_type is an integral type capable of
// representing all values of bitmask
bitmask operator& (bitmask X, bitmask Y) {
    return static_cast<bitmask>(
        static_cast<int_type>(X) &
        static_cast<int_type>(Y));
}

bitmask operator| (bitmask X, bitmask Y) {
    return static_cast<bitmask>(
        static_cast<int_type>(X) |
        static_cast<int_type>(Y));
}
```

\(^{(155)}\) Such as an integer type, with constant integer values (3.9.1).
411 Library introduction 17.3 Method of description (Informative)

```c

bitmask operator~(bitmask X, bitmask Y)
{
    return static_cast<bitmask>(
        static_cast<int_type>(X) ^
        static_cast<int_type>(Y));
}

bitmask operator~(bitmask X){
    return static_cast<bitmask>(~static_cast<int_type>(X));
}

bitmask& operator&=(bitmask& X, bitmask Y){
    X = X & Y; return X;
}

bitmask& operator|=(bitmask& X, bitmask Y) {
    X = X | Y; return X;
}

bitmask& operator^=(bitmask& X, bitmask Y) {
    X = X ^ Y; return X;
}

```

3 Here, the names C0, C1, etc. represent bitmask elements for this particular bitmask type. All such elements have distinct values such that, for any pair Ci and Cj, Ci & Cj is nonzero and Ci & Cj is zero.

4 The following terms apply to objects and values of bitmask types:
   — To set a value Y in an object X is to evaluate the expression X |= Y.
   — To clear a value Y in an object X is to evaluate the expression X &|= ~Y.
   — The value Y is set in the object X if the expression X & Y is nonzero.

17.3.2.1.3 Character sequences [character.seq]

1 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.  

156) Note that this definition differs from the definition in ISO C subclause 7.1.1.

157) declared in <clocale> (22.4).
17.3 Method of description (Informative) Library introduction

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).\(^{158}\)

1 The length of an NTBS is the number of elements that precede the terminating null character. An empty NTBS has a length of zero.

2 The value of an NTBS is the sequence of values of the elements up to and including the terminating null character.

3 A static NTBS is an NTBS with static storage duration.\(^{159}\)

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.\(^{160}\)

1 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 \((8.3.4)\) \(T A [N]\), where \(T\) is type \(\text{char16}_t\) \((3.9.1)\), optionally qualified by any combination of \(\text{const}\) or \(\text{volatile}\). The initial elements of the array have defined contents up to and including an element determined by some predicate. A 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.\(^{161}\)

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.\(^{162}\)

17.3.2.1.3.4 char32_t sequences

A char32-character sequence is an array object \((8.3.4)\) \(T A [N]\), where \(T\) is type \(\text{char32}_t\) \((3.9.1)\), optionally qualified by any combination of \(\text{const}\) or \(\text{volatile}\). The initial elements of the array have defined contents up to and including an element determined by some predicate. A 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.\(^{163}\)

\(^{158}\) Many of the objects manipulated by function signatures declared in \(<\text{cstring}>\) \((21.5)\) are character sequences or NTBSs. The size of some of these character sequences is limited by a length value, maintained separately from the character sequence.

\(^{159}\) A string literal, such as "abc", is a static NTBS.

\(^{160}\) 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.

\(^{161}\) Many of the objects manipulated by function signatures declared in \(<\text{cuchar}>\) are char16-character sequences or NTC16s.

\(^{162}\) A char16_t string literal, such as u"abc", is a static NTC16s.

\(^{163}\) Many of the objects manipulated by function signatures declared in \(<\text{cuchar}>\) are char32-character sequences or NTC32s.
17.3 Method of description (Informative)

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.\(^{164}\)

### 17.3.2.1.3.5 Wide-character sequences

1 A *wide-character sequence* is an array object (8.3.4) \(T A[N]\), 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.\(^{165}\)

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.\(^{166}\)

### 17.3.2.2 Functions within classes

1 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).

2 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

1 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.

2 Objects of certain classes are sometimes required by the external specifications of their classes to store data, apparently in member objects. For the sake of exposition, some subclauses provide representative declarations, and semantic requirements, for private member objects of classes that meet the external specifications of the classes. The declarations for such member objects and the definitions of related member types are enclosed in a comment that ends with exposition only, as in:

```c
// streambuf* sb;
```

\(^{164}\) A char32_t string literal, such as U"abc", is a static NTC32S.

\(^{165}\) Many of the objects manipulated by function signatures declared in <cwchar> are wide-character sequences or NTWCSS.

\(^{166}\) A wide string literal, such as L"abc" is a static NTWCS.
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`.

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.\(^{167}\)

The C++ Standard Library provides 43 C++ headers, as shown in Table 13.

| <algorithm> | <iomanip> | <map> | <sstream> | <typeinfo> |
| <array> | <ios> | <memory> | <stack> | <type_traits> |
| <bitset> | <iosfwd> | <new> | <stdexcept> | <unordered_map> |
| <complex> | <iostream> | <numeric> | <streambuf> | <unordered_set> |
| <datetime> | <iostream> | <ostream> | <string> | <utility> |
| <deque> | <iterator> | <queue> | <strstream> | <valarray> |
| <exception> | <limits> | <random> | <system_error> | <vector> |
| <fstream> | <list> | <regex> | <threads> |
| <functional> | <locale> | <set> | <tuple> |

The facilities of the Standard C Library are provided in 25 additional headers, as shown in Table 14.

Except as noted in clauses 18 through 27 and Annex D the contents of each header `name.h` shall be the same as that of the corresponding header `name.h`, as specified in the Standard C99 Library (1.2) or the C Unicode TR, as appropriate, as

\(^{167}\) 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).

Draft
Table 14: C++ Headers for C Library Facilities

<table>
<thead>
<tr>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;cassert&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cfloat&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cmath&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cstddef&gt;</code></td>
</tr>
<tr>
<td><code>&lt;ctgmath&gt;</code></td>
</tr>
<tr>
<td><code>&lt;ccomplex&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cinttypes&gt;</code></td>
</tr>
<tr>
<td><code>&lt;csetjmp&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cstdio&gt;</code></td>
</tr>
<tr>
<td><code>&lt;ctime&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cctype&gt;</code></td>
</tr>
<tr>
<td><code>&lt;ciso646&gt;</code></td>
</tr>
<tr>
<td><code>&lt;csignal&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cstdint&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cuchar&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cerrno&gt;</code></td>
</tr>
<tr>
<td><code>&lt;climits&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cstdarg&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cwchar&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cfenv&gt;</code></td>
</tr>
<tr>
<td><code>&lt;clocale&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cstddef&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cstring&gt;</code></td>
</tr>
<tr>
<td><code>&lt;cwctype&gt;</code></td>
</tr>
</tbody>
</table>

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).

5 Names which are defined as macros in C shall be defined as macros in the C++ Standard Library, even if C grants license for implementation as functions. [Note: the names defined as macros in C include the following: assert, offsetof, setjmp, va_arg, va_end, and va_start. — end note]

6 Names that are defined as functions in C shall be defined as functions in the C++ Standard Library.\(^{168}\)

7 Identifiers that are keywords or operators in C++ shall not be defined as macros in C++ standard library headers.\(^{169}\)

8 D.5, Standard C library headers, describes the effects of using the name.h (C header) form in a C++ program.\(^{170}\)

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 15: C++ Headers for Freestanding Implementations

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.1 Types</td>
<td><code>&lt;cassert&gt;</code></td>
</tr>
<tr>
<td>18.2 Implementation properties</td>
<td><code>&lt;limits&gt;</code></td>
</tr>
<tr>
<td>18.4 Start and termination</td>
<td><code>&lt;stdlib&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;cstdlib&gt;</code></td>
</tr>
</tbody>
</table>

The supplied version of the header `<cstdlib>` shall declare at least the functions abort(), atexit(), and exit() (18.4).

\(^{168}\) 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.

\(^{169}\) In particular, including the standard header `<ciso646.h>` or `<ciso646>` has no effect.

\(^{170}\) 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.
17.4 Library-wide requirements

17.4.2 Using the library

1 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

1 The entities in the C++ Standard Library are defined in headers, whose contents are made available to a translation unit when it contains the appropriate #include preprocessing directive (16.2).

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.\(^{(171)}\)

3 A translation unit shall include a header only outside of any external declaration or definition, and shall include the header lexically before the first reference to any of the entities it declares in that translation unit.

17.4.2.2 Linkage

1 Entities in the C++ Standard Library have external linkage (3.5). Unless otherwise specified, objects and functions have the default extern "C++" linkage (7.5).

2 Whether a name from the 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.\(^{(172)}\)

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 (17.4.3.5), run-time changes (17.4.3.6).

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 use of the namespace std (17.4.3.1), namespace its use of various reserved names (17.4.3.2), its use of headers (17.4.3.3), classes derived from its use of standard library classes as base classes (17.4.3.4), its definitions of replacement functions (17.4.3.5), and its installation of handler functions during execution (17.4.3.6).

17.4.3.1 Namespace std

1 It is undefined for a C++ program to add The behavior of a C++ program is undefined if it adds declarations or definitions to namespace std or to a namespaces within namespace std unless otherwise specified. A program may add a 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 only if the declaration depends on a user-defined type of external linkage and unless the specialization meets the standard library requirements for the original template.\(^{(173)}\)

2 It is undefined for a C++ program to declare The behavior of a C++ program is undefined if it declares

\(^{(171)}\) This is the same as the Standard C library.

\(^{(172)}\) 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.

\(^{(173)}\) 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.
— 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 a template defined 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.

3 A translation unit shall not declare namespace std to be an inline namespace (7.3.1).

### 17.4.3.2 Reserved names

The C++ Standard Library reserves the following kinds of names:

— Macros
— Global names
— Names with external linkage

If the a program declares or defines a name in a context where it is reserved, other than as explicitly allowed by this clause, the its behavior is undefined.

#### 17.4.3.2.1 Macro names

A translation unit that includes a standard library header shall not #define or #undef names declared in any standard library header.

A translation unit shall not #define or #undef names lexically identical to keywords.

#### 17.4.3.2.2 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.\(^{174}\)

#### 17.4.3.2.3 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,\(^{175}\) both in namespace std and in the global namespace.

\(^{174}\) Such names are also reserved in namespace ::std (17.4.3.2).
\(^{175}\) The list of such reserved names includes errno, declared or defined in <cerrno>.
Each global function signature declared with external linkage in a header is reserved to the implementation to designate that function signature with external linkage.

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, or as a name of namespace scope in the global namespace.

For each type `T` from the Standard C library, the types `::T` and `std::T` are reserved to the implementation and, when defined, `::T` shall be identical to `std::T`.

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.

Virtual member function signatures defined for a base class in the C++ Standard library may be overridden in a derived class defined in the program (10.3).

Clauses 18 through 30 and Annex D describe the behavior of numerous functions defined by the C++ Standard Library. Under some circumstances, however, certain of these function descriptions also apply to replacement functions defined in the program (17.1).

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&)`

The list of such reserved function signatures with external linkage includes `setjmp(jmp_buf)`, declared or defined in `<setjmp>`, and `va_end(va_list)`, declared or defined in `<stdarg>`.

The function signatures declared in `<uchar>`, `<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.

These types are `clock_t`, `div_t`, `FILE`, `fpos_t`, `fenv_t`, `ldiv_t`, `mbstate_t`, `pthreads_t`, `sig_atomic_t`, `size_t`, `time_t`, `tm`, `va_list`, `wctrans_t`, `wctype_t`, and `wint_t`.  

Draft
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.

17.4.3.6 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.

17.4.3.7 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 Library-wide requirements

17.4.3.8 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.

17.4.3.9 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 (17.4.4.5), access specifiers (17.4.4.6), class derivation (17.4.4.7), and exceptions (17.4.4.8).

17.4.4.1 Headers

A C++ header may include other C++ headers. A C++ header may include other C++ headers. 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).

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

The names or global function signatures described in 17.4.1.1 are reserved to the implementation.

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

It is unspecified whether any global or non-member functions in the C++ Standard Library are defined as inline (7.1.2).

(79) C++ headers must include a C++ header that contains any needed definition (3.2).
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.\(^{180}\)

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:

\[
\text{*out\_stream << value;}
\text{if (delim != 0)}
\text{\hspace{1em} \text{*out\_stream << delim;}}
\text{\hspace{1em} \text{return (**this);} }
\]

— end note ]

17.4.4.4 Member functions

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;\(^{181}\) The same latitude does not extend to the implementation of virtual or global or non-member functions, however.

— 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.

3 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.\(^{182}\)

17.4.4.5 Reentrancy

Which of the functions in the C++ Standard Library are not reentrant subroutines is implementation-defined.

17.4.4.6 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

It is unspecified whether a class in the C++ Standard Library is itself derived from other classes (with names reserved to the implementation).

2 Certain classes defined in the C++ Standard Library are derived from other classes in the C++ Standard Library:

\(^{180}\) 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.

\(^{181}\) Hence, taking the address of a member function has an unspecified type.

\(^{182}\) 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.
— 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.\(^\text{183}\)

### 17.4.4.8 Restrictions on exception handling \[^\text{res.on.exception.handling}\]

1. 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.\(^\text{184}\)

2. None of the functions from the Standard C library shall report an error by throwing an exception,\(^\text{185}\) unless it calls a program-supplied function that throws an exception.\(^\text{186}\)

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.\(^\text{187}\) An implementation may strengthen this implicit exception-specification by adding an explicit one.\(^\text{188}\)

### 17.4.4.9 Value of error codes \[^\text{value.error.codes}\]

1. Certain functions in the C++ Standard Library report errors via a `std::error_code \(19.4.2.1\)` object. That object’s `category()` member shall return a reference to `std::system_category` for errors originating from the operating system, or a reference to an implementation-defined `error_category` object for errors originating elsewhere. The implementation shall define the possible values of `value()` for each of these error categories. \[Example: For a POSIX-based operating system, implementations are encouraged to define the `std::system_category` values as identical to the POSIX `errno` values, plus additional values as defined by the operating system’s documentation. Implementations on non-POSIX operating systems are encouraged to define values identical to the operating system’s values. For errors that do not originate from the operating system, the implementation may provide enums for the associated values. \[end example\] \(^\text{188}\)\]

\(^{183}\) An implicit exception to this rule are types described as synonyms for basic integral types, such as `size_t \(18.1\)` and `streamoff \(27.4.1\)`.

\(^{184}\) 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.

\(^{185}\) 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.

\(^{186}\) The functions `qsort()` and `bsearch()` \(25.4\) meet this condition.

\(^{187}\) In particular, they can report a failure to allocate storage by throwing an exception of type `bad_alloc`, or a class derived from `bad_alloc` \(18.5.2.1\). Library implementations are encouraged (but not required) to report errors by throwing exceptions from (or derived from) the standard exception classes \(18.5.2.1, 18.7, 19.1\).

\(^{188}\) 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`. Draft
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>&lt;cassert&gt;</td>
</tr>
<tr>
<td>18.2 Implementation properties</td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>18.3 Integer types</td>
<td>&lt;cstdint&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;cassert&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;csetjmp&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;ctime&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;csetjmp&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;ctime&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cstdbool&gt;</td>
</tr>
</tbody>
</table>

18.1 Types  [support.types]

1 Common definitions.

Header <cassert> (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).\(^{189}\)

The macro `offsetof(type, member-designator)` accepts a restricted set of `type` arguments in this International Standard. If `type` is not a standard-layout class (clause 9), the results are undefined.\(^{190}\) 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.

`nullptr_t` is defined as follows:

```cpp
namespace std {
    typedef decltype(nullptr) nullptr_t;
}
```

The type for which `nullptr_t` is a synonym has the characteristics described in 3.9.1 and 4.10. [Note: Although `nullptr`'s address cannot be taken, the address of another `nullptr_t` object that is an lvalue can be taken. — end note]

See also: subclause 5.3.3, Sizeof, subclause 5.7, Additive operators, subclause 12.5, Free store, and ISO C subclause 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 constant expressions.

Non-arithmetic standard types, such as `complex<T>` (26.3.2), shall not have specializations.

**Header `<limits>` synopsis**

\(^{189}\) Possible definitions include 0 and `0L`, but not `(void*)0`.

\(^{190}\) Note that `offsetof` is required to work as specified even if unary `operator&` is overloaded for any of the types involved.

---

Table 17: Header `<cstdlib>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td>NULL, offsetof</td>
</tr>
<tr>
<td>Types:</td>
<td>ptrdiff_t, size_t, max_align_t, nullptr_t</td>
</tr>
</tbody>
</table>
namespace std {
    template<class T> class numeric_limits;
    enum float_round_style;
    enum float_denorm_style;

template<> class numeric_limits<bool>;

template<> class numeric_limits<char>;
template<> class numeric_limits<signed char>;
template<> class numeric_limits<unsigned char>;
template<> class numeric_limits<char16_t>;
template<> class numeric_limits<char32_t>;
template<> class numeric_limits<wchar_t>;

template<> class numeric_limits<short>;
template<> class numeric_limits<int>;
template<> class numeric_limits<long>;
template<> class numeric_limits<long long>;
template<> class numeric_limits<unsigned short>;
template<> class numeric_limits<unsigned int>;
template<> class numeric_limits<unsigned long>;
template<> class numeric_limits<unsigned long long>;

template<> class numeric_limits<float>;
template<> class numeric_limits<double>;
template<> class numeric_limits<long double>;
}

18.2.1.1 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;
    }
}

Draft
18.2 Implementation properties

The default \( \text{numeric\_limits}\langle T\rangle \) template shall have all members, but with 0 or false values.

The value of each member of a specialization of \( \text{numeric\_limits} \) on a \( \text{cv}\)-qualified type \( \text{cv}\ T \) shall be equal to the value of the corresponding member of the specialization on the unqualified type \( T \).

18.2.1.2 \( \text{numeric\_limits} \) members

\[
\begin{align*}
\text{static constexpr int} & \quad \text{max\_exponent} = 0; \\
\text{static constexpr int} & \quad \text{max\_exponent10} = 0; \\
\text{static constexpr bool} & \quad \text{has\_infinity} = \text{false}; \\
\text{static constexpr bool} & \quad \text{has\_quiet\_NaN} = \text{false}; \\
\text{static constexpr bool} & \quad \text{has\_signaling\_NaN} = \text{false}; \\
\text{static constexpr float\_denorm\_style} & \quad \text{has\_denorm} = \text{denorm\_absent}; \\
\text{static constexpr bool} & \quad \text{has\_denorm\_loss} = \text{false}; \\
\text{static constexpr T} & \quad \text{infinity}\langle\rangle \text{ throw}\langle\rangle; \\
\text{static constexpr T} & \quad \text{quiet\_NaN}\langle\rangle \text{ throw}\langle\rangle; \\
\text{static constexpr T} & \quad \text{signaling\_NaN}\langle\rangle \text{ throw}\langle\rangle; \\
\text{static constexpr T} & \quad \text{denorm\_min}\langle\rangle \text{ throw}\langle\rangle; \\
\text{static constexpr bool} & \quad \text{is\_iec559} = \text{false}; \\
\text{static constexpr bool} & \quad \text{is\_bounded} = \text{false}; \\
\text{static constexpr bool} & \quad \text{is\_modulo} = \text{false}; \\
\text{static constexpr bool} & \quad \text{traps} = \text{false}; \\
\text{static constexpr bool} & \quad \text{tinyness\_before} = \text{false}; \\
& \quad \text{static constexpr float\_round\_style\_round\_style} = \text{round\_toward\_zero}; \\
\end{align*}
\]

\[
\begin{align*}
\text{template\langle class T\rangle} & \quad \text{class numeric\_limits\langle const T\rangle}; \\
\text{template\langle class T\rangle} & \quad \text{class numeric\_limits\langle volatile T\rangle}; \\
\text{template\langle class T\rangle} & \quad \text{class numeric\_limits\langle const volatile T\rangle}; \\
\end{align*}
\]

1 Minimum finite value.\(^{\text{191}}\)

For floating types with denormalization, returns the minimum positive normalized value.

Meaningful for all specializations in which \( \text{is\_bounded} \neq \text{false} \) or \( \text{is\_bounded} == \text{false} \&\& \text{is\_signed} == \text{false} \).

2 Maximum finite value.\(^{\text{192}}\)

Meaningful for all specializations in which \( \text{is\_bounded} \neq \text{false} \).

\[\text{static constexpr T} \quad \text{min}\langle\rangle \text{ throw}\langle\rangle;\]
\[\text{static constexpr T} \quad \text{max}\langle\rangle \text{ throw}\langle\rangle;\]
\[\text{static constexpr T} \quad \text{lowest}\langle\rangle \text{ throw}\langle\rangle;\]

\(^{\text{191}}\) Equivalent to CHAR\_MIN, SHRT\_MIN, FLT\_MIN, DBL\_MIN, etc.

\(^{\text{192}}\) Equivalent to CHAR\_MAX, SHRT\_MAX, FLT\_MAX, DBL\_MAX, etc.
A finite value \( x \) such that there is no other finite value \( y \) where \( y < x \).\(^{193}\)

Meaningful for all specializations in which \texttt{is\_bounded} \!\!\!\!\!\!= \texttt{false}.

\begin{verbatim}
static constexpr int digits;
\end{verbatim}

Number of \texttt{radix} digits that can be represented without change.

For integer types, the number of non-sign bits in the representation.

For floating point types, the number of \texttt{radix} digits in the mantissa.\(^{194}\)

\begin{verbatim}
static constexpr int digits10;
\end{verbatim}

Number of base 10 digits that can be represented without change.\(^{195}\)

Meaningful for all specializations in which \texttt{is\_bounded} \!\!\!\!\!\!= \texttt{false}.

\begin{verbatim}
static constexpr int max_digits10;
\end{verbatim}

Number of base 10 digits required to ensure that values which differ are always differentiated.

Meaningful for all floating point types.

\begin{verbatim}
static constexpr bool is_signed;
\end{verbatim}

True if the type is signed.

Meaningful for all specializations.

\begin{verbatim}
static constexpr bool is_integer;
\end{verbatim}

True if the type is integer.

Meaningful for all specializations.

\begin{verbatim}
static constexpr bool is_exact;
\end{verbatim}

True if the type uses an exact representation. All integer types are exact, but not all exact types are integer. For example, rational and fixed-exponent representations are exact but not integer.

Meaningful for all specializations.

\begin{verbatim}
static constexpr int radix;
\end{verbatim}

For floating types, specifies the base or radix of the exponent representation (often 2).\(^{196}\)

For integer types, specifies the base of the representation.\(^{197}\)

Meaningful for all specializations.

\(^{193}\) \texttt{lowest()} is necessary because not all floating-point representations have a smallest (most negative) value that is the negative of the largest (most positive) finite value.

\(^{194}\) Equivalent to \texttt{FLT\_MANT\_DIG}, \texttt{DBL\_MANT\_DIG}, \texttt{LDBL\_MANT\_DIG}.

\(^{195}\) Equivalent to \texttt{FLT\_DIG}, \texttt{DBL\_DIG}, \texttt{LDBL\_DIG}.

\(^{196}\) Equivalent to \texttt{FLT\_RADIX}.

\(^{197}\) Distinguishes types with bases other than 2 (e.g. BCD).
static constexpr T epsilon() throw();

Machine epsilon: the difference between 1 and the least value greater than 1 that is representable.\(^{198}\)

static constexpr T round_error() throw();

Measure of the maximum rounding error.\(^{199}\)

static constexpr int min_exponent;

Minimum negative integer such that \(\text{radix}\) raised to the power of one less than that integer is a normalized floating point number.\(^{200}\)

Meaningful for all floating point types.

static constexpr int min_exponent10;

Minimum negative integer such that 10 raised to that power is in the range of normalized floating point numbers.\(^{201}\)

Meaningful for all floating point types.

static constexpr int max_exponent;

Maximum positive integer such that \(\text{radix}\) raised to the power one less than that integer is a representable finite floating point number.\(^{202}\)

Meaningful for all floating point types.

static constexpr int max_exponent10;

Maximum positive integer such that 10 raised to that power is in the range of representable finite floating point numbers.\(^{203}\)

Meaningful for all floating point types.

static constexpr bool has_infinity;

True if the type has a representation for positive infinity.

Meaningful for all floating point types.

static constexpr bool has_quiet_NaN;

Shall be true for all specializations in which is_iec559 != false.

\(^{198}\) Equivalent to FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON.

\(^{199}\) 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.

\(^{200}\) Equivalent to FLT_MIN_EXP, DBL_MIN_EXP, LDBL_MIN_EXP.

\(^{201}\) Equivalent to FLT_MIN_10_EXP, DBL_MIN_10_EXP, LDBL_MIN_10_EXP.

\(^{202}\) Equivalent to FLT_MAX_EXP, DBL_MAX_EXP, LDBL_MAX_EXP.

\(^{203}\) Equivalent to FLT_MAX_10_EXP, DBL_MAX_10_EXP, LDBL_MAX_10_EXP.

Draft
True if the type has a representation for a quiet (non-signaling) “Not a Number.”\(^{204}\)
Meaningful for all floating point types.
Shall be true for all specializations in which `is_iec559` \(!=\) false.

```cpp
static constexpr bool has_signaling_NaN;
```
True if the type has a representation for a signaling “Not a Number.”\(^{205}\)
Meaningful for all floating point types.
Shall be true for all specializations in which `is_iec559` \(!=\) false.

```cpp
static constexpr float_denorm_style has_denorm;
```
denorm_present if the type allows denormalized values (variable number of exponent bits)\(^{206}\),
denorm_absent if the type does not allow denormalized values, and
denorm_indeterminate if it is indeterminate at compile time whether the type allows denormalized values.
Meaningful for all floating point types.

```cpp
static constexpr bool has_denorm_loss;
```
True if loss of accuracy is detected as a denormalization loss, rather than as an inexact result\(^{207}\).

```cpp
static constexpr float infinity() throw();
```
Representation of positive infinity, if available\(^{208}\).
Meaningful for all specializations for which `has_infinity` \(!=\) false. Required in specializations for which `is_iec559` \(!=\) false.

```cpp
static constexpr float quiet_NaN() throw();
```
Representation of a quiet “Not a Number,” if available\(^{209}\).
Meaningful for all specializations for which `has_quiet_NaN` \(!=\) false. Required in specializations for which `is_iec559` \(!=\) false.

```cpp
static constexpr float signaling_NaN() throw();
```
Representation of a signaling “Not a Number,” if available\(^{210}\).
Meaningful for all specializations for which `has_signaling_NaN` \(!=\) false. Required in specializations for which `is_iec559` \(!=\) false.

```cpp
static constexpr float denorm_min() throw();
```
Required by LIA-1.

---

\(^{204}\) Required by LIA-1.
\(^{205}\) Required by LIA-1.
\(^{206}\) Required by LIA-1.
\(^{207}\) See IEC 559.
\(^{208}\) Required by LIA-1.
\(^{209}\) Required by LIA-1.
\(^{210}\) Required by LIA-1.
18.2 Implementation properties

Minimum positive denormalized value.\(^{211}\)

Meaningful for all floating point types.

In specializations for which \texttt{has\_denorm == false}, returns the minimum positive normalized value.

```cpp
static constexpr bool is_iec559;
```

True if and only if the type adheres to IEC 559 standard.\(^{212}\)

Meaningful for all floating point types.

```cpp
static constexpr bool is_bounded;
```

True if the set of values representable by the type is finite.\(^{213}\) [Note: All built-in types are bounded. This member would be false for arbitrary precision types. —end note]

Meaningful for all specializations.

```cpp
static constexpr bool is_modulo;
```

True if the type is modulo.\(^{214}\) 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.

Generally, this is \texttt{false} for floating types, \texttt{true} for unsigned integers, and \texttt{true} for signed integers on most machines.

Meaningful for all specializations.

```cpp
static constexpr bool traps;
```

True if, at program startup, there exists a value of the type that would cause an arithmetic operation using that value to trap.\(^{215}\)

Meaningful for all specializations.

```cpp
static constexpr bool tinyness_before;
```

True if tinyness is detected before rounding.\(^{216}\)

Meaningful for all floating point types.

```cpp
static constexpr float round_style round_style;
```

The rounding style for the type.\(^{217}\)

Meaningful for all floating point types. Specializations for integer types shall return \texttt{round\_toward\_zero}.

\(^{211}\) Required by LIA-1.
\(^{212}\) International Electrotechnical Commission standard 559 is the same as IEEE 754.
\(^{213}\) Required by LIA-1.
\(^{214}\) Required by LIA-1.
\(^{215}\) Required by LIA-1.
\(^{216}\) Refer to IEC 559. Required by LIA-1.
\(^{217}\) Equivalent to FLT\_ROUNDS. Required by LIA-1.
18.2.1.3 Type float_round_style

namespace std {
enum float_round_style {
  round_indeterminate = -1,
  round_toward_zero = 0,
  round_to_nearest = 1,
  round_toward_infinity = 2,
  round_toward_neg_infinity = 3
};
}

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

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.

[Example:}

Draft
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;
         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 float_denorm_style 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 ]

3 The specialization for bool shall be provided as follows:
namespace std {

template<> class numeric_limits<bool> {
    public:
        static constexpr bool is_specialized = true;
        static constexpr bool min() 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;
        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_denorm = true;
        static constexpr float_denorm_style has_denorm = denorm_absent;
        static constexpr bool has_denorm_loss = false;
        static constexpr bool has_infinity = false;
        static constexpr bool has_quiet_NaN = false;
        static constexpr bool has_signaling_NaN = false;
        static constexpr float_round_style round_style = round_toward_zero;

    };
}

18.2.2 C Library [c.limits]

Header <climits> (Table 18)\footnote{The types of the constants defined by

\footnote{\texttt{LLONG\_MAX}, \texttt{LLONG\_MIN}, and \texttt{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
18.3 Integer types

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</td>
</tr>
<tr>
<td></td>
<td>LONG_MAX</td>
</tr>
<tr>
<td></td>
<td>SCHAR_MIN</td>
</tr>
<tr>
<td></td>
<td>SHRT_MIN</td>
</tr>
<tr>
<td></td>
<td>ULLONG_MAX</td>
</tr>
<tr>
<td>CHAR_MAX</td>
<td>LONGLONG_MAX</td>
</tr>
<tr>
<td></td>
<td>LONG_MIN</td>
</tr>
<tr>
<td></td>
<td>SCHAR_MAX</td>
</tr>
<tr>
<td></td>
<td>UCHAR_MAX</td>
</tr>
<tr>
<td></td>
<td>ULONG_MAX</td>
</tr>
<tr>
<td>CHAR_MIN</td>
<td>LONGLONG_MIN</td>
</tr>
<tr>
<td></td>
<td>MB_LEN_MAX</td>
</tr>
<tr>
<td></td>
<td>SHRT_MAX</td>
</tr>
<tr>
<td></td>
<td>UINT_MAX</td>
</tr>
<tr>
<td></td>
<td>USHRT_MAX</td>
</tr>
<tr>
<td>INT_MIN</td>
<td></td>
</tr>
</tbody>
</table>

.. note::

   Macros in `<climits>` are not required to match the types to which the macros refer. — end note

3 Header `<cfloat>` (Table 19)\(^{219}\)

Table 19: Header `<cfloat>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values:</td>
<td></td>
</tr>
<tr>
<td>DBL_DIG</td>
<td>DBL_MIN_EXP</td>
</tr>
<tr>
<td></td>
<td>FLT_MAX_EXP</td>
</tr>
<tr>
<td></td>
<td>LDBL_MANT_DIG</td>
</tr>
<tr>
<td>DBL_EPSILON</td>
<td>DECIMAL_DIG</td>
</tr>
<tr>
<td></td>
<td>FLT_MIN</td>
</tr>
<tr>
<td></td>
<td>LDBL_MAX_10_EXP</td>
</tr>
<tr>
<td>DBL_MANT_DIG</td>
<td>FLT_DIG</td>
</tr>
<tr>
<td></td>
<td>FLT_MIN_10_EXP</td>
</tr>
<tr>
<td></td>
<td>LDBL_MAX_EXP</td>
</tr>
<tr>
<td>DBL_MAX</td>
<td>FLT_EPSILON</td>
</tr>
<tr>
<td></td>
<td>FLT_MIN_EXP</td>
</tr>
<tr>
<td></td>
<td>LDBL_MAX</td>
</tr>
<tr>
<td>DBL_MAX_10_EXP</td>
<td>FLT_EVAL_METHOD</td>
</tr>
<tr>
<td></td>
<td>FLT_RADIX</td>
</tr>
<tr>
<td></td>
<td>LDBL_MIN</td>
</tr>
<tr>
<td>DBL_MAX_EXP</td>
<td>FLT_MANT_DIG</td>
</tr>
<tr>
<td></td>
<td>FLT_ROUND</td>
</tr>
<tr>
<td></td>
<td>LDBL_MIN_10_EXP</td>
</tr>
<tr>
<td>DBL_MIN</td>
<td>FLT_MAX</td>
</tr>
<tr>
<td></td>
<td>LDBL_DIG</td>
</tr>
<tr>
<td></td>
<td>LDBL_MIN_EXP</td>
</tr>
<tr>
<td>DBL_MIN_10_EXP</td>
<td>FLT_MAX_10_EXP</td>
</tr>
<tr>
<td></td>
<td>LDBL_EPSILON</td>
</tr>
</tbody>
</table>

\(^{219}\)DECIMAL_DIG and FLT_EVAL_METHOD added by TR1.

4 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;     // optional

typedef signed integer type intptr_t;     // optional

typedef unsigned integer type uint8_t;    // optional

typedef unsigned integer type uint16_t;   // optional

typedef unsigned integer type uint32_t;   // optional

typedef unsigned integer type uint64_t;   // optional

typedef unsigned integer type uint_fast8_t;
typedef unsigned integer type uint_fast16_t;
typedef unsigned integer type uint_fast32_t;
typedef unsigned integer type uint_fast64_t;

typedef unsigned integer type uint_least8_t;
typedef unsigned integer type uint_least16_t;
typedef unsigned integer type uint_least32_t;
typedef unsigned integer type uint_least64_t;

typedef unsigned integer type uintmax_t;
typedef unsigned integer type uintptr_t;  // optional

} // namespace std

The header also defines numerous macros of the form:

INT_[FAST LEAST]{8 16 32 64}_MIN
[U]INT_[FAST LEAST]{8 16 32 64}_MAX
INT{MAX PTR}_MIN
[U]INT{MAX PTR}_MAX
{PTRDIFF SIG_ATOMIC WCHAR WINT}{_MAX _MIN}
SIZE_MAX

plus function macros of the form:

[U]INT{8 16 32 64 MAX}_C

The header defines all functions, types, and macros the same as 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
Table 20: Header `<cstdlib>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros</td>
<td>EXIT_FAILURE EXIT_SUCCESS</td>
</tr>
<tr>
<td>Functions</td>
<td>abort atexit at_quick_exit exit</td>
</tr>
<tr>
<td></td>
<td>quick_exit</td>
</tr>
</tbody>
</table>

**18.4 Start and termination**

The contents are the same as the Standard C library header `<stdlib.h>`, with the following changes:

1. 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 (*f)(void))
   extern "C++" int atexit(void (*f)(void))
   ```

   **Effects:** The `atexit()` functions register the function pointed to by `f` to be called without arguments at normal program termination.

2. **Implementation limits:** The implementation shall support the registration of at least 32 functions.

3. **Returns:** The `atexit()` function returns zero if the registration succeeds, nozero if it fails.

4. 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.221) 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.

   220) 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()`.

   221) A function is called for every time it is registered.

Draft
If control leaves a registered function called by exit because the function does not provide a handler for a thrown exception, terminate() shall be called.

— 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.\(^{222}\)

— 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.\(^{223}\)

8 The function exit() never returns to its caller.

```c
extern "C" int at_quick_exit(void (*f)(void));
extern "C++" int at_quick_exit(void (*f)(void));
```

9 Effects: The at_quick_exit() functions register the function pointed to by f to be called without arguments when quick_exit is called. The at_quick_exit() functions shall be thread safe. \(\text{[Note: The at_quick_exit registrations are distinct from the atexit registrations, and applications may need to call both registration functions with the same argument. — end note]}\)

Implementation limits: The implementation shall support the registration of at least 32 functions.

11 Returns: zero if the registration succeeds, non-zero if it fails.

```c
void quick_exit(int status)
```

12 Effects: Functions registered by calls to at_quick_exit are called in the reverse order of their registration, except that a function shall be called after any previously registered functions that had already been called at the time it was registered. Objects shall not be destroyed as a result of calling quick_exit.

If control leaves a registered function called by quick_exit because the function does not provide a handler for a thrown exception, terminate() shall be called.

After calling registered functions, quick_exit shall call _Exit(status). \(\text{[Note: The standard file buffers are not flushed. See: ISO C subclause 7.20.4.4. — end note]}\)

13 The function quick_exit() never returns to its caller.

See also: subclauses 3.6, 3.6.3, ISO C subclause 7.10.4.

18.5 Dynamic memory management \[support.dynamic\]

The header `<new>` defines several functions that manage the allocation of dynamic storage in a program. It also defines components for reporting storage management errors.

Header `<new>` synopsis

\(^{222}\) 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>`.  

\(^{223}\) The macros EXIT_FAILURE and EXIT_SUCCESS are defined in `<cstdlib>`.

Draft
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

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.

— 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.

Draft
— The loop terminates when an attempt to allocate the requested storage is successful or when a called `new_handler` function does not return.

```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 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. Otherwise, returns a null pointer.

[Example:
```c
T* p1 = new T; // throws bad_alloc if it fails
T* p2 = new(nothrow) T; // returns 0 if it fails
```

— end example ]

```c
void operator delete(void* ptr) throw();
```

```c
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:** `ptr` shall be a null pointer or its value shall be a value returned by an earlier call to the (possibly replaced) `operator new(std::size_t)` or `operator new(std::size_t, const std::nothrow_t&)` which has not been invalidated by an intervening call to `operator delete(void*)`.

**Default behavior:** If `ptr` is null, does nothing. Otherwise, reclaims the storage allocated by the earlier call to `operator new`.

**Remarks:** It is unspecified under what conditions part or all of such reclaimed storage will be allocated by subsequent calls to `operator new` or any of `calloc`, `malloc`, or `realloc`, declared in `<cstdlib>`.

```c
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)`. 

Draft
18.5 Dynamic memory management

18.5.1.2 Array forms

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.\(^{224}\)

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).

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.

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: ptr shall be a null pointer or its value shall be the value returned by an earlier call to operator new[](std::size_t) or operator new[](std::size_t,const std::nothrow_t&) which has not been invalidated by an intervening call to operator delete[](void*).

Default behavior: Calls operator delete(ptr).

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 array 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.

\(^{224}\) 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
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.

```cpp
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:

```cpp
void* place = operator new(sizeof(Something));
Something* p = new (place) Something();
```

— end example]

```cpp
void* operator new[](std::size_t size, void* ptr) throw();
```

Returns: ptr.

Remarks: Intentionally performs no other action.

```cpp
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).

```cpp
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

```cpp
namespace std {
    class bad_alloc : public exception {
    public:
        bad_alloc() throw();
        bad_alloc(const bad_alloc&) throw();
        bad_alloc& operator=(const bad_alloc&) throw();
        virtual const char* what() const throw();
    };
}
```

Draft
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` [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` [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.

### 18.6 Type identification [support.rtti]

The header `<typeinfo>` defines a type associated with type information generated by the implementation. It also defines two types for reporting dynamic type identification errors.

**Header `<typeinfo> synopsis**
namespace std {
    class type_info;
    class bad_cast;
    class bad_typeid;
}

SEE ALSO: 5.2.7, 5.2.8.

18.6.1 Class type_info

namespace std {
    class type_info {
    public:
        virtual ~type_info();
        bool operator==(const type_info& rhs) const;
        bool operator!=(const type_info& rhs) const;
        bool before(const type_info& rhs) const;
        const char* name() const;
        
        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.

bool operator==(const type_info& rhs) const;

Effects: Compares the current object with rhs.

Returns: true if the two values describe the same type.

bool operator!=(const type_info& rhs) const;

Returns: !(this == rhs).

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.

const char* name() const;

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)
18.6 Type identification

18.6.2 Class 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();
    }
}

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();

Effects: Constructs an object of class bad_cast.

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();

Effects: Copies an object of class bad_cast.

virtual const char* what() const throw();

Returns: An implementation-defined NTBS.

Remarks: The message may be a null-terminated multibyte string (17.3.1.3.2), suitable for conversion and display as a wstring (21.2.22.1.4)

18.6.3 Class bad_typeid

namespace std {
    class bad_typeid : public exception {
    public:
        bad_typeid() throw();
        bad_typeid(const bad_typeid&) throw();
        bad_typeid& operator=(const bad_typeid&) throw();
        virtual const char* what() const throw();
    }
}

The class bad_typeid defines the type of objects thrown as exceptions by the implementation to report a null pointer in a typeid expression (5.2.8).

bad_typeid() throw();

Effects: Constructs an object of class bad_typeid.

Remarks: The result of calling what() on the newly constructed object is implementation-defined.
445 Language support library 18.7 Exception handling

bad_typeid(const bad_typeid&) throw();
bad_typeid& operator=(const bad_typeid&) throw();

Effects: Copies an object of class bad_typeid.

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 (21.2.22.1.4)

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

namespace std {
    class exception;
    class bad_exception;
    class nested_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);

    template <class T> void throw_with_nested(T&& t); // [[noretturn]]
    template <class E> void rethrow_if_nested(const E& e);
}

See also: 15.5.

18.7.1 Class exception [exception]

namespace std {
    class exception {
    public:
        exception() throw();
        exception(const exception&) throw();

Draft
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.

```cpp
exception() throw();
```

**Effects:** Constructs an object of class `exception`.

**Remarks:** Does not throw any exceptions.

```cpp
exception(const exception&) throw();
excep& operator=(const exception&) throw();
```

**Effects:** Copies an exception object.

**Remarks:** The effects of calling `what()` after assignment are implementation-defined.

```cpp
virtual ~exception() throw();
```

**Effects:** Destroys an object of class `exception`.

**Remarks:** Does not throw any exceptions.

```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). 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-specifications

#### 18.7.2.1 Class `bad_exception`

```cpp
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();
    }; // class bad_exception
}
```

The class `bad_exception` defines the type of objects thrown as described in (15.5.2).

```cpp
bad_exception() throw();
```
2    Effects: Constructs an object of class bad_exception.
3    Remarks: The result of calling what() on the newly constructed object is implementation-defined.

    bad_exception(const bad_exception&) throw();
    bad_exception& operator=(const bad_exception&) throw();
4    Effects: Copies an object of class bad_exception.

    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).

18.7.2.2 Type unexpected_handler

typedef void (unexpected_handler)();
1    The type of a handler function to be called by unexpected() when a function attempts to throw an exception not listed in its exception-specification.
2    Required behavior: An unexpected_handler shall not return. See also 15.5.2.
3    Default behavior: The implementation’s default unexpected_handler calls terminate().

18.7.2.3 set_unexpected

unexpected_handler set_unexpected(unexpected_handler f) throw();
1    Effects: Establishes the function designated by f as the current unexpected_handler.
2    Requires: f shall not be a null pointer.
3    Returns: The previous unexpected_handler.

18.7.2.4 unexpected

void unexpected();
1    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.
2    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 Exception handling

18.7.3 Abnormal termination

18.7.3.1 Type terminate_handler

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

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

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

bool uncaught_exception() throw();

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 a null pointer.

Throws: the exception object to which p refers.

template<class E> exception_ptr copy_exception(E e);

Effects: as if

try {
    throw e;
} catch(...) {
    return current_exception();
}

[Note: this function is provided for convenience and efficiency reasons. —end note]

18.7.6 nested_exception

namespace std {
    class nested_exception {

Draft
1. The class \texttt{nested\_exception} is designed for use as a mixin through multiple inheritance. It captures the currently handled exception and stores it for later use.

2. \textit{[Note: \texttt{nested\_exception} has a virtual destructor to make it a polymorphic class. Its presence can be tested for with \texttt{dynamic\_cast. — end note]}]

\texttt{nested\_exception() throw();}

\textit{Effects: The constructor calls \texttt{current\_exception()} and stores the returned value.}

\texttt{void rethrow\_nested() const; // [[noreturn]]}

\textit{Throws: the stored exception captured by this \texttt{nested\_exception} object.}

\texttt{exception\_ptr nested\_ptr() const;}

\textit{Returns: the stored exception captured by this \texttt{nested\_exception} object.}

\texttt{template<class T> void throw\_with\_nested(T& t); // [[noreturn]]}

\texttt{template <class E> void rethrow\_if\_nested(const E& e);}
### 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 `<setjmp>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</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>Macros:</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>

### 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  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>

### Table 26: Header `<stdbool>` 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>

Draft
The restrictions that ISO C places on the second parameter to the `va_start()` macro in header `<stdarg.h>` are different in this International Standard. The parameter `parmN` is the identifier of the rightmost parameter in the variable parameter list of the function definition (the one just before the `...`). If the parameter `parmN` 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.

**SEE ALSO:** ISO C subclause 4.8.1.1.

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.

The header `<stdbool.h>` and the header `<stdbool.h>` shall not define macros named `bool`, `true`, and `false`.

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, except that it may use functions defined in clause 29 that are not member functions. 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.

---

225) Note that `va_start` is required to work as specified even if unary `operator&` is overloaded for the type of `parmN`.

226) 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>
<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: 

1. **logic errors** and 
2. **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);
  };
}

1 The class logic_error defines the type of objects thrown as exceptions to report errors presumably detectable before the program executes, such as violations of logical preconditions or class invariants.

logic_error(const string& what_arg);

2 Effects: Constructs an object of class logic_error.

3 Postcondition: strcmp(what(), what_arg.c_str()) == 0.

logic_error(const char* what_arg);

4 Effects: Constructs an object of class logic_error.

5 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);
  };
}

1 The class domain_error defines the type of objects thrown as exceptions by the implementation to report domain errors.

domain_error(const string& what_arg);

2 Effects: Constructs an object of class domain_error.

3 Postcondition: strcmp(what(), what_arg.c_str()) == 0.

domain_error(const char* what_arg);

4 Effects: Constructs an object of class domain_error.

5 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`

Draft
The class `range_error` defines the type of objects thrown as exceptions to report range errors in internal computations.

```cpp
range_error(const string& what_arg);
```

Effects: Constructs an object of class `range_error`.

Postcondition: `strcmp(what(), what_arg.c_str()) == 0`.

```cpp
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` [overflow.error]

```cpp
namespace std {
    class overflow_error : public runtime_error {
        public:
            explicit overflow_error(const string& what_arg);
            explicit overflow_error(const char* what_arg);
    };
}
```

The class `overflow_error` defines the type of objects thrown as exceptions to report an arithmetic overflow error.

```cpp
overflow_error(const string& what_arg);
```

Effects: Constructs an object of class `overflow_error`.

Postcondition: `strcmp(what(), what_arg.c_str()) == 0`.

```cpp
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` [underflow.error]

```cpp
namespace std {
    class underflow_error : public runtime_error {
        public:
            explicit underflow_error(const string& what_arg);
    };
}
```

The class `underflow_error` defines the type of objects thrown as exceptions to report an arithmetic underflow error.

```cpp
underflow_error(const string& what_arg);
```

Effects: Constructs an object of class `underflow_error`.

Postcondition: `strcmp(what(), what_arg.c_str()) == 0`.

```cpp
underflow_error(const char* what_arg);
```

Effects: Constructs an object of class `underflow_error`.

Postcondition: `strcmp(what(), what_arg) == 0`.  

Draft


```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.

```cpp
underflow_error(const string& what_arg);
```

*Effects:* Constructs an object of class **underflow_error**.

*Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

```cpp
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>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Macro:</strong></td>
<td><code>assert</code></td>
</tr>
</tbody>
</table>

Table 28: Header `<cassert>` synopsis

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. [Note: The intent is to remain in close alignment with the POSIX standard. — end note]
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 \texttt{errno} (19.3). Implementations are encouraged but not required to leave unchanged the error states provided by other libraries.

**Header `<system_error>` synopsis**

```cpp
namespace std {
    class error_category;
    class error_code;
    class error_condition;
    class system_error;

    template <class T>
    struct is_error_code_enum : public false_type {};

    template <class T>
    struct is_error_condition_enum : public false_type {};

    namespace posix_error {
        enum posix_errno {
            address_family_not_supported, // EAFNOSUPPORT
            address_in_use, // EADDRINUSE
            address_not_available, // EADDRNOTAVAIL
            already_connected, // EISCONN
            argument_list_too_long, // E2BIG

            Draft
```
19.4 System error support

Diagnocts library

argument_out_of_domain,  // EDOM
bad_address,  // EFAULT
bad_file_descriptor,  // EBADF
bad_message,  // EBADMSG
broken_pipe,  // EPIPE
connection_aborted,  // ECONNABORTED
connection_already_in_progress,  // EALREADY
connection_refused,  // ECONNREFUSED
connection_reset,  // ECONNRESET
cross_device_link,  // EXDEV
destination_address_required,  // EDESTADDRREQ
device_or_resource_busy,  // EBUSY
directory_not_empty,  // ENOTEMPTY
executable_format_error,  // ENOEXEC
file_exists,  // EXIST
file_too_large,  // EFBIG
filename_too_long,  // ENAMETOOLONG
function_not_supported,  // ENOSYS
host_unreachable,  // EHOSTUNREACH
identifier_removed,  // EIDRM
illegal_byte_sequence,  // EISDIR
inappropriate_io_control_operation,  // ENOTTY
interrupted,  // EINTR
invalid_argument,  // EINVAL
invalid_seek,  // ESPIPE
io_error,  // EIO
is_a_directory,  // EISDIR
message_size,  // EMSGSIZE
network_down,  // ENETDOWN
network_reset,  // ENETRESET
network_unreachable,  // ENETUNREACH
no_buffer_space,  // ENOBUFS
no_child_process,  // ECHILD
no_link,  // ENOLINK
no_lock_available,  // ENOLCK
no_message_available,  // ENODATA
no_message,  // ENOMSG
no_protocol_option,  // ENOPROTOOPT
no_space_on_device,  // ENOSPC
no_stream_resources,  // ENOSR
no_such_device_or_address,  // ENXIO
no_such_device,  // ENODEV
no_such_file_or_directory,  // ENOENT
no_such_process,  // ESRCCH
not_a_directory,  // ENOTDIR
not_a_socket,  // ENOTSOCK
not_a_stream,  // ENOSTR
not_connected,  // ENOTCONN
not_enough_memory,  // ENOMEM
not_supported,  // ENOTSUP

Draft
namespace posix_error {

    error_code make_error_code(posix_errno e);
    error_condition make_error_condition(posix_errno e);
} // namespace posix_error

// 19.4.4 Comparison operators:
bool operator==(const error_code& lhs, const error_code& rhs);
bool operator==(const error_code& lhs, const error_condition& rhs);
bool operator==(const error_condition& lhs, const error_code& rhs);
bool operator==(const error_condition& lhs, const error_condition& rhs);
bool operator!=(const error_code& lhs, const error_code& rhs);
bool operator!=(const error_code& lhs, const error_condition& rhs);
bool operator!=(const error_condition& lhs, const error_code& rhs);
bool operator!=(const error_condition& lhs, const error_condition& rhs);
} // namespace std

3 The value of each enum posix_errno constant shall be the same as the value of the <cerrno> macro shown in the above synopsis. Whether or not the <system_error> implementation exposes the <cerrno> macros is unspecified.

4 The is_error_code_enum and is_error_condition_enum templates may be specialized for user-defined types to indicate that such a type is eligible for class error_code and class error_condition automatic conversions,
respectively.

### 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]

```cpp
namespace std {
    class error_category {
    public:
        virtual ~error_category();
        error_category(const error_category&) = delete;
        error_category& operator=(const error_category&) = delete;
        virtual const char* name() const = 0;
        virtual error_condition default_error_condition(int ev) const;
        virtual bool equivalent(int code, const error_condition& condition) const;
        virtual bool equivalent(const error_code& code, int condition) const;
        virtual string message(int ev) const = 0;
        bool operator==(const error_category& rhs) const;
        bool operator!=(const error_category& rhs) const;
        bool operator<(const error_category& rhs) const;
    }

    const error_category& get_posix_category();
    const error_category& get_system_category();

    static const error_category& posix_category = get_posix_category();
    static const error_category& native_category system_category = get_system_category();
} // namespace std
```

### 19.4.1.2 Class `error_category` virtual members

```cpp
virtual const char* name() const = 0;

Returns: A string naming the error category.

Throws: Nothing.
```

```cpp
virtual error_condition default_error_condition(int ev) const;

Returns: `error_condition(ev, *this)`.

Throws: Nothing.
```

```cpp
virtual bool equivalent(int code, const error_condition& condition) const;
```
virtual bool equivalent(const error_code& code, int condition) const;

Returns: *this == code.category() && code.value() == condition.

Throws: Nothing.

default_error_condition(code) == condition.

Throws: Nothing.

virtual string message(int ev) const = 0;

Returns: A string that describes the error condition denoted by ev.

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).

bool operator<(const error_category& rhs) const;

Returns: less<error_category*>(this, &rhs).

[Note: less (20.5.7) provides a total ordering for pointers. — end note]

Throws: Nothing.

19.4.1.4 Program defined classes derived from error_category

virtual const char *name() const = 0;

Returns: a string naming the error category.

Throws: Nothing.

virtual error_condition default_error_condition(int ev) const;

Returns: An object of type error_condition that corresponds to ev.

Throws: Nothing.

virtual bool equivalent(int code, const error_condition& condition) const;

Returns: true if, for the category of error represented by *this, code is considered equivalent to condition; otherwise, false.

Throws: Nothing.
virtual bool equivalent(const error_code& code, int condition) const;

Returns: true if, for the category of error represented by *this, code is considered equivalent to condition; otherwise, false.

Throws: Nothing.

19.4.1.5 Error category objects

const error_category& get_posix_category();

Returns: A reference to an object of a type derived from class error_category.

Remarks: The object’s default_error_condition and equivalent virtual functions shall behave as specified for the class error_category. The object’s name virtual function shall return a pointer to the string "POSIX".

const error_category& get_system_category();

Returns: A reference to an object of a type derived from class error_category.

Remarks: The object’s equivalent virtual functions shall behave as specified for class error_category. The object’s name virtual function shall return a pointer to the string "system". The object’s default_error_condition virtual function shall behave as follows:

If the argument ev corresponds to a POSIX errno value posv, the function shall return error_condition(posv, posix_category). Otherwise, the function shall return error_condition(ev, system_category). What constitutes correspondence for any given operating system is unspecified. [Note: The number of potential system error codes is large and unbounded, and some may not correspond to any POSIX errno value. Thus implementations are given latitude in determining correspondence. — end note]

19.4.2 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]

namespace std {
    class error_code {
        public:
        // 19.4.2.2 constructors:
        error_code();
        error_code(int val, const error_category& cat); // 19.4.2.2.1
        template <class ErrorCodeEnum>
        error_code(ErrorCodeEnum e, typename enable_if<is_error_code_enum<ErrorCodeEnum>::value>::type * = 0); // 19.4.2.2.2

        // 19.4.2.3 modifiers:
        void assign(int val, const error_category& cat);
        template <class ErrorCodeEnum>
        void assign(ErrorCodeEnum e);
    } // end class
} // end namespace std

Draft
typedef enable_if<is_error_code_enum<ErrorCodeEnum>::value>::type&
    operator=(ErrorCodeEnum e);

void clear();

// 19.4.2.4 observers:
int value() const;
const error_category& category() const;
error_condition default_error_condition() const;
string message() const;
explicit operator bool() const;

private:
    int val_;   // exposition only
    const error_category& cat_; // exposition only
};

// 19.4.2.5 non-member functions:
bool operator<(const error_code& lhs, const error_code& rhs);

template <class charT, class traits>
basic_ostream<charT,traits>&
    operator<<(basic_ostream<charT,traits>& os, const error_code& ec);

} // namespace std

19.4.2.2 Class error_code constructors

error_code();
Effects: Constructs an object of type error_code.
Postconditions: val_ == 0 and cat_ == system_category.
Throws: Nothing.

error_code(int val, const error_category& cat);
Effects: Constructs an object of type error_code.
Postconditions: val_ == val and cat_ == cat.
Throws: Nothing.

template <class ErrorCodeEnum>
error_code(ErrorCodeEnum e,
    typename enable_if<is_error_code_enum<ErrorCodeEnum>::value>::type * = 0);
Effects: Constructs an object of type error_code.
Postconditions: *this == make_error_code(e).
Throws: Nothing.
19.4 System error support

19.4.2.3 Class error_code modifiers

```cpp
void assign(int val, const error_category& cat);

Postconditions: val_ == val and cat_ == cat.
Throws: Nothing.
```

```cpp
template <class ErrorCodeEnum>
type<enable_if<is_error_code_enum<ErrorCodeEnum>::value>::type&
operator=(ErrorCodeEnum e);

Postconditions: *this == make_error_code(e).
Returns: *this.
Throws: Nothing.
```

```cpp
void clear();
Postconditions: value() == 0 and category() == system_category.
```

19.4.2.4 Class error_code observers

```cpp
int value() const;

Returns: val_.
Throws: Nothing.
```

```cpp
const error_category& category() const;

Returns: cat_.
Throws: Nothing.
```

```cpp
error_condition default_error_condition() const;

Returns: category().default_error_condition(value()).
Throws: Nothing.
```

```cpp
string message() const;

Returns: category().message(value()).
Throws: Nothing.
```

```cpp
explicit operator bool() const;

Returns: value() != 0.
Throws: Nothing.
```
19.4.2.5 Class error_code non-member functions

error_code make_error_code(posix_errno e);

Returns: error_code(e, posix_category).

bool operator<(const error_code& lhs, const error_code& rhs);

Returns: lhs.category() < rhs.category() || lhs.category() == rhs.category() && lhs.value() < rhs.value().

Throws: Nothing.

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().

19.4.3 Class error_condition

19.4.3.1 Class error_condition overview

The class error_condition describes an object used to hold values identifying error conditions. [Note: error_condition values are portable abstractions, while error_code values (19.4.2) are implementation specific. —end note]

namespace std {
  class error_condition {
    public:
      // 19.4.3.2 constructors:
      error_condition();
      error_condition(int val, const error_category& cat);
      template <class ErrorConditionEnum>
      error_condition(ErrorConditionEnum e,
      typename enable_if<is_error_condition_enum<ErrorConditionEnum> >::type* = 0);

      // 19.4.3.3 modifiers:
      void assign(int val, const error_category& cat);
      template<typename ErrorConditionEnum>
      typename enable_if<is_error_condition_enum<ErrorConditionEnum>, error_code>::type &
      operator=(ErrorConditionEnum e);
      void clear();

      // 19.4.3.4 observers:
      int value() const;
      const error_category& category() const;
      string message() const;
      explicit operator bool() const;
  
    private:
  }

Draft
19.4 System error support

```cpp
int val_; // exposition only
cons const error_category& cat_; // exposition only
};

// 19.4.3.5 non-member functions:
bool operator<(const error_condition& lhs, const error_condition& rhs);
} // namespace std

19.4.3.2 Class error_condition constructors

```cpp
error_condition();
   
Effects: Constructs an object of type error_condition.
Postconditions: val_ == 0 and cat_ == posix_category.
Throws: Nothing.
```

```cpp
error_condition(int val, const error_category& cat);
   
Effects: Constructs an object of type error_condition.
Postconditions: val_ == val and cat_ == cat.
Throws: Nothing.
```

```cpp
```template <class ErrorConditionEnum>
error_condition(ErrorConditionEnum e,
   typename enable_if<is_error_condition_enum<ErrorConditionEnum>::value>::type * = 0);
   
Effects: Constructs an object of type error_condition.
Postcondition: *this == make_error_condition(e).
Throws: Nothing.
```

19.4.3.3 Class error_condition modifiers

```cpp
void assign(int val, const error_category& cat);
   
Postconditions: val_ == val and cat_ == cat.
Throws: Nothing.
```

```cpp
```template <class ErrorConditionEnum>
   typename enable_if<is_error_condition_enum<ErrorConditionEnum>::value>::type&
   operator=(ErrorConditionEnum e);
   
Postcondition: *this == make_error_condition(e).
Throws: Nothing.
```

```cpp
void clear();
Postconditions: \( \text{value}() == 0 \) and \( \text{category}() == \text{posix\_category} \).

**19.4.3.4 Class error\_condition observers**

```cpp
int \text{value}() \text{const};
\begin{align*}
1 & \quad \text{Returns: val\_.} \\
2 & \quad \text{Throws: Nothing.}
\end{align*}
\begin{align*}
\text{const error\_category\&} \text{category}() \text{const};
3 & \quad \text{Returns: cat\_.} \\
4 & \quad \text{Throws: Nothing.}
\end{align*}
\begin{align*}
\text{string message()} \text{const};
5 & \quad \text{Returns: category().message(value()).} \\
6 & \quad \text{Throws: Nothing.}
\end{align*}
\begin{align*}
\text{explicit operator bool()} \text{const};
7 & \quad \text{Returns: value() != 0.} \\
8 & \quad \text{Throws: Nothing.}
\end{align*}
```

**19.4.3.5 Class error\_condition non-member functions**

```cpp
\text{error\_condition make\_error\_condition(posix\_errno e);} \\
\begin{align*}
\quad & \quad \text{Returns: error\_condition(e, posix\_category).}
\end{align*}
\begin{align*}
\text{bool operator<}(\text{const error\_condition\&} \text{lhs, const error\_condition\&} \text{rhs});
1 & \quad \text{Returns: lhs.category() < rhs.category() || lhs.category() == rhs.category() && lhs.value()} \\
2 & \quad \quad < \text{rhs.value().} \\
3 & \quad \text{Throws: Nothing.}
\end{align*}
```

**19.4.4 Comparison operators**

```cpp
\begin{align*}
\text{bool operator==}(\text{const error\_code\&} \text{lhs, const error\_code\&} \text{rhs});
1 & \quad \text{Returns: lhs.category() == rhs.category() && lhs.value()} \\
2 & \quad \quad == \text{rhs.value().} \\
3 & \quad \text{Throws: Nothing.}
\end{align*}
\begin{align*}
\text{bool operator==}(\text{const error\_code\&} \text{lhs, const error\_condition\&} \text{rhs});
\end{align*}
```
19.4 System error support

Returns: \( \text{lhs.category().equivalent(lhs.value(), rhs) || rhs.category.equivalent(lhs, rhs.value())} \).

Throws: Nothing.

bool operator==(const error_condition& lhs, const error_code& rhs);

Returns: \( \text{rhs.category().equivalent(rhs.value(), lhs) || lhs.category.equivalent(rhs, lhs.value())} \).

Throws: Nothing.

bool operator==(const error_condition& lhs, const error_condition& rhs);

Returns: \( \text{lhs.category() == rhs.category() && lhs.value() == rhs.value()} \).

Throws: Nothing.

bool operator!=(const error_code& lhs, const error_code& rhs);
bool operator!=(const error_code& lhs, const error_condition& rhs);
bool operator!=(const error_condition& lhs, const error_code& rhs);
bool operator!=(const error_condition& lhs, const error_condition& rhs);

Returns: \( ! (\text{lhs == rhs}) \).

Throws: Nothing.

19.4.5 Class system_error

19.4.5.1 Class system_error overview

The class \text{system_error} describes an exception object used to report error conditions that have an associated error code. Such error conditions typically originate from the operating system or other low-level application program interfaces.

[Note: If an error represents an out-of-memory condition, implementations are encouraged to throw an exception object of type \text{bad_alloc} 18.5.2.1 rather than \text{system_error}. — end note]

namespace std {
class system_error : public runtime_error {
public:
    system_error(error_code ec, const string& what_arg);
    system_error(error_code ec);
    system_error(int ev, const error_category& ecat, 
                  const string& what_arg);
    system_error(int ev, const error_category& ecat);
    const error_code& code() const throw();
    const char* what() const throw();
};
} // namespace std

19.4.5.2 Class system_error members

system_error(error_code ec, const string& what_arg);
Effect: Constructs an object of class `system_error`.

Postconditions: `code() == ec` and `strcmp(runtime_error::what(), what_arg.c_str()) == 0`.

`system_error(error_code ec);`

Effect: Constructs an object of class `system_error`.

Postconditions: `code() == ec` and `strcmp(runtime_error::what(), "") == 0`.

`system_error(int ev, const error_category& ecat, const string& what_arg);`

Effect: Constructs an object of class `system_error`.

Postconditions: `code() == error_code(ev, ecat)` and `strcmp(runtime_error::what(), what_arg.c_str()) == 0`.

`system_error(int ev, const error_category& ecat);`

Effect: Constructs an object of class `system_error`.

Postconditions: `code() == error_code(ev, ecat)` and `strcmp(runtime_error::what(), "") == 0`.

Returns: `ec` or `error_code(ev, ecat).from()`, from the constructor, as appropriate.

`const char *what() const throw();`

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

1 This clause describes components used by other elements of the C++ Standard Library. These components may also be used by C++ programs.

2 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>
<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

1 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

1 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.

2 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 \([\text{equalitycomparable}]\)

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>requirement</th>
</tr>
</thead>
</table>
| \(a == b\) | convertible to \(\text{bool}\) | \(==\) is an equivalence relation, that is, it has the following properties:  
  - For all \(a\), \(a == a\).  
  - If \(a == b\), then \(b == a\).  
  - If \(a == b\) and \(b == c\), then \(a == c\). |

### Table 32: LessThanComparable requirements \([\text{lessthанcomparable}]\)

<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 (\text{bool})</td>
<td>(&lt;) is a strict weak ordering relation ((25.3))</td>
</tr>
</tbody>
</table>

### Table 33: MoveConstructible requirements \([\text{moveconstructible}]\)

<table>
<thead>
<tr>
<th>expression</th>
<th>post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T \ t = rv)</td>
<td>(t) is equivalent to the value of (rv) before the construction. ([Note: \text{There is no requirement on the value of } rv \text{ after the construction. — end note}])</td>
</tr>
</tbody>
</table>

### Table 34: CopyConstructible requirements \([\text{copyconstructible}]\)

<table>
<thead>
<tr>
<th>expression</th>
<th>post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T \ t = u)</td>
<td>(\text{the value of } u) is unchanged and is equivalent to (t) ([Note: \text{A type that satisfies the CopyConstructible requirements also satisfies the MoveConstructible requirements. — end note}])</td>
</tr>
</tbody>
</table>

### Table 35: MoveAssignable requirements \([\text{moveassignable}]\)

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>return value</th>
<th>post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t = rv)</td>
<td>(T&amp;)</td>
<td>(t)</td>
<td>(t) is equivalent to the value of (rv) before the assignment. ([Note: \text{There is no requirement on the value of } rv \text{ 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 s and s 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>
<tr>
<td>Variable</td>
<td>Definition</td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
</tr>
<tr>
<td>t</td>
<td>a value of type <code>const T&amp;</code></td>
</tr>
<tr>
<td>a, a1, a2</td>
<td>values of type <code>X&amp;</code></td>
</tr>
<tr>
<td>b</td>
<td>a value of type <code>Y</code></td>
</tr>
<tr>
<td>p</td>
<td>a value of type <code>X::pointer</code>, obtained by calling <code>a1.allocate</code>, where <code>a1 == a</code></td>
</tr>
<tr>
<td>q</td>
<td>a value of type <code>X::const_pointer</code> obtained by conversion from a value <code>p</code></td>
</tr>
<tr>
<td>r</td>
<td>a value of type <code>X::reference</code> obtained by the expression <code>*p</code></td>
</tr>
<tr>
<td>s</td>
<td>a value of type <code>X::const_reference</code> obtained by the expression <code>*q</code> or by conversion from a value <code>r</code></td>
</tr>
<tr>
<td>u</td>
<td>a value of type <code>Y::const_pointer</code> obtained by calling <code>Y::allocate</code>, or else 0</td>
</tr>
<tr>
<td>v</td>
<td>a value of type <code>V</code></td>
</tr>
<tr>
<td>n</td>
<td>a value of type <code>X::size_type</code></td>
</tr>
<tr>
<td>Args</td>
<td>a template parameter pack</td>
</tr>
<tr>
<td>args</td>
<td>a function parameter pack with the pattern <code>Args&amp;&amp;</code></td>
</tr>
</tbody>
</table>
### Table 40: Allocator requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>assertion/note</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 type</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. (^{227}) [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. operator== shall be reflexive, symmetric, and transitive.</td>
</tr>
<tr>
<td>a1 != a2</td>
<td>bool</td>
<td>same as !(a1 == a2)</td>
</tr>
<tr>
<td>X()</td>
<td>creates a default instance. [Note: a destructor is assumed. — end note]</td>
<td></td>
</tr>
<tr>
<td>X a1(a);</td>
<td>post: a1 == a</td>
<td></td>
</tr>
<tr>
<td>X a(b);</td>
<td>post: Y(a) == b, a == X(b)</td>
<td></td>
</tr>
<tr>
<td>a.construct(p, args)</td>
<td>(not used)</td>
<td>Effect: Constructs an object of type T at p by invoking T(forward&lt;Args&gt;(args)...)</td>
</tr>
<tr>
<td>a.destroy(p)</td>
<td>(not used)</td>
<td>Effect: Destroys the object at p</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

\(^{227}\) 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
to SomeAllocator<T>, then Allocator::rebind<U>::other is the same type as SomeAllocator<U>.

4 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.

5 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.

6 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

1 This subclause contains some basic function and class templates that are used throughout the rest of the library.

Header <utility> synopsis

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>&);

    Draft
template <class T1, class T2>
  bool operator<=(const pair<T1,T2>&, const 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>
  void swap(pair<T1,T2>&, pair<T1,T2>&&);
template <class T1, class T2>
  pair<typename decay<T1>::type V1, typename decay<T2>::type V2> 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

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;
    const T& operator()(const T& x) const;
};
```

[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]

```cpp
const T& operator()(const T& x) const;
```

2 Returns: x

```cpp
template <class T> T&& forward(typename identity<T>::type&& t);
```

[Note: The use of identity forces users to explicitly specify the template parameter. This is necessary to get the correct forwarding semantics. — end note]

```cpp
Returns: t.
```

5 Return type: If T is an lvalue-reference type, an lvalue; otherwise, an rvalue.

6 [Example:

```cpp
template <class T, class A1, class A2>
shared_ptr<T> factory(A1&& a1, A2&& a2) {
    return shared_ptr<T>(new T(std::forward<A1>(a1), std::forward<A2>(a2)));
}
```

```cpp
struct A {
    A(int&, const double&);
};
```

```cpp
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
}
```

7 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);
```

8 Returns: t.

9 Return type: an rvalue.
Example:

```cpp
template <class T, class A1>
shared_ptr<T> factory(A1&& a1) {
    return shared_ptr<T>(new T(std::forward<A1>(a1)));
}

struct A {
    A();
    A(const A&); // copies from lvalues
    A(A&&); // moves from rvalues
};

void g() {
    A a;
    shared_ptr<A> sp1 = factory<A>(a); // "a" binds to A(const A&)
    shared_ptr<A> sp1 = factory<A>(std::move(a)); // "a" binds to A(A&&)
}
```

In the first call to `factory`, `A1` is deduced as `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 20.3.1.4 and 20.3.1.5).

```cpp
namespace std {
    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);

        // allocator-extended constructors
        template <class Alloc> pair(allocator_arg_t, const Alloc& a);
    }
} // namespace std
```
template <class Alloc>
    pair(allocation_arg_t, const Alloc& a, const T1& x, const T2& y);
template <class U, class V, class Alloc>
    pair(allocation_arg_t, const Alloc& a, U& x, V& y);
template <class Alloc>
    pair(allocation_arg_t, const Alloc&, pair&& p);
template <class U, class V, class Alloc>
    pair(allocation_arg_t, const Alloc& a, pair<U, V>& p);
template <class U, class V, class Alloc>
    pair(allocation_arg_t, const Alloc& a, pair<U, V>&& p);
template <class U, class... Args>
    pair(allocation_arg_t, const Alloc& a, Args... args);

    pair& operator=(pair&& p);
    template<class U, class V> pair& operator=(pair<U, V>&& p);

    void swap(pair&& p);
};

template <class T1, class T2, class Alloc>
struct uses_allocator<pair<T1, T2>, Alloc>;

template <class T1, class T2>
struct constructible_with_allocator_prefix<pair<T1, T2> >:
    true_type {
};

// Requires: Alloc shall be an Allocator (20.1.2).
[ Note: Specialization of this trait informs other library components that pair can be constructed
  with an allocator, even though it does not have a nested allocator_type. — end note ]

template <class T1, class T2>
struct constructible_with_allocator_prefix<pair<T1, T2> > : true_type {
    [ Note: Specialization of this trait informs other library components that pair can be constructed
      with an allocator prefix argument. — end note ]

    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).

```cpp
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)....

```cpp
pair(pair&& p);
```

Effects: The constructor initializes first with std::move(p.first) and second with std::move(p.second).

```cpp
template<class U, class V> pair(const pair<U, V>& p);
```

Effects: Initializes members from the corresponding members of the argument, performing implicit conversions as needed.

```cpp
template<class U, class V> pair(pair<U, V>&& p);
```

Effects: The constructor initializes first with std::move(p.first) and second with std::move(p.second).

```cpp
template<class Alloc> pair(allocator_arg_t, const Alloc& a);
template<class Alloc>
  pair(allocator_arg_t, const Alloc& a, const T1& x, const T2& y);
template<class U, class Alloc>
  pair(allocator_arg_t, const Alloc& a, U&& x, V&& y);
template<class Alloc>
  pair(allocator_arg_t, const Alloc, pair&& p);
template<class U, class Alloc>
  pair(allocator_arg_t, const Alloc& a, const pair<U, V>& p);
template<class U, class Alloc>
  pair(allocator_arg_t, const Alloc& a, pair<U, V>&& p);
template<class Alloc>
  pair(allocator_arg_t, const Alloc& a, U&& x, Args&&... args);
```

Requires: Alloc shall be an Allocator (20.1.2).

Effects: The members first and second are both allocator constructed (20.6.2) with a.

```cpp
pair& operator=(pair&& p);
```

Effects: Assigns to first with std::move(p.first) and to second with std::move(p.second).

Returns: *this.

```cpp
template<class U, class V> pair& operator=(pair<U, V>&& p);
```

Effects: Assigns to first with std::move(p.first) and to second with std::move(p.second).

Returns: *this.

```cpp
void swap(pair&& p);
```

Effects: Swaps first with p.first and second with p.second.

Requires: first_type and second_type shall be Swappable.

Draft
template <class T1, class T2>
bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y);

Returns: \(x.first == y.first && x.second == y.second\).

template <class T1, class T2>
bool operator<(const pair<T1, T2>& x, const pair<T1, T2>& y);

Returns: \(x.first < y.first || (!y.first < x.first) && x.second < y.second\).

template<class T1, class T2> void swap(pair<T1, T2>& x, pair<T1, T2>& y);

Effects: \(x.swap(y)\).

template <class T1, class T2>
pair<typename decay<T1>::type V1, typename decay<T2>::type V2> make_pair(T1&&, T2&&);

Returns:
\[\text{pair<typename decay<T1>::type V1, typename decay<T2>::type V2>(std::forward<T1>(x), std::forward<T2>(y))};\]

where \(V1\) and \(V2\) are determined as follows: Let \(U_i\) be \(\text{decay<Ti>::type}\) for each \(T_i\). Then each \(V_i\) is \(X\&\) if \(U_i\) equals \(\text{reference_wrapper<X>}\), otherwise \(V_i\) is \(U_i\).

[Example: In place of:
\[
\text{return pair<int, double>(5, 3.1415926);} \quad \text{// explicit types}
\]
a C++ program may contain:
\[
\text{return make_pair(5, 3.1415926);} \quad \text{// types are deduced}
\]

\] — end example \]

tuple_size<pair<T1, T2> >::value

Returns: integral constant expression.

Value: 2.

tuple_element<0, pair<T1, T2> >::type

Value: the type \(T_1\).

tuple_element<1, pair<T1, T2> >::type

Value: the type \(T_2\).

\]
28  Return type: If I == 0 then P is T1, if I == 1 then P is T2, and otherwise the program is ill-formed.

29  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

namespace std {
    // 20.3.1, class template tuple:
    template <class... Types> class tuple;

    // 20.3.1.3, tuple creation functions:
    const unspecified ignore;

    template <class... Types>
    tuple<VTypes...> 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...>&);

    // 20.3.1.4, 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...>>;

    // 20.3.1.5, 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...>&);

    // 20.3.1.6, relational operators:
    template <class... TTypes, class... UTypes>

Draft
20.3 Tuples

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...>&);

} // namespace std

20.3.1 Class template tuple

namespace std {

template <class... Types>
class tuple {
public:

tuple();
explicit tuple(const Types&...);
template <class... UTypes>
explicit tuple(UTypes&&...);

tuple(const tuple&);
tuple(tuple&&);

tuple(const tuple<TTypes...>&);
template <class... UTypes>
tuple(const 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

// allocator-extended constructors

template <class Alloc>
tuple(allocator_arg_t, const Alloc& a);
template <class Alloc>
tuple(allocator_arg_t, const Alloc& a, const Types&...);
template <class Alloc, class... UTypes>
tuple(allocator_arg_t, const Alloc& a, const UTypes&&...);
}

Draft
template <class Alloc>
tuple(allocator_arg_t, const Alloc& a, const tuple&);
template <class Alloc>
tuple(allocator_arg_t, const Alloc& a, tuple&&);
template <class Alloc, class... UTypes>
tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
template <class Alloc, class... UTypes>
tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);
template <class Alloc, class... UTypes>
tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&);
template <class Alloc, class U1, class U2>
tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&);
template <class Alloc, class U1, class U2>
tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);

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 Tuple traits

template <class... Types, class Alloc>
struct uses_allocator<tuple<Types...>, Alloc> : true_type { };  

Requires: Alloc shall be an Allocator (20.1.2).

[Note: Specialization of this trait informs other library components that tuple can be constructed with an allocator, even though it does not have a nested allocator_type. — end note]


template <class... Types>
struct constructible_with_allocator_prefix<tuple<Types...>> > : true_type { }

[Note: Specialization of this trait informs other library components that tuple can be constructed with an allocator prefix argument. — end note]

20.3.1.2 Construction

tuple();

Requires: Each type in Types shall be default constructible.
2  
\textit{Effects:} Default initializes each element.

\texttt{tuples(const Types&...)};

3  
\textit{Requires:} Each type in \texttt{Types} shall be copy constructible.

\textit{Effects:} Copy initializes each element with the value of the corresponding parameter.

\template<\texttt{class... UTypes}>
\texttt{explicit tuples(UTypes&&... u)};

4  
\textit{Requires:} Each type in \texttt{Types} shall be copy constructible.

\textit{Effects:} Copy initializes each element with the value of the corresponding parameter.

\template<\texttt{class... UTypes}>
\texttt{tuples(const tuples&& u)};

5  
\textit{Requires:} Each type in \texttt{Types} shall be copy constructible.

\textit{Effects:} Copy constructs each element of \texttt{*this} with the corresponding element of \texttt{u}.

\template<\texttt{class... UTypes}>
\texttt{tuples(tuples&& u)};

6  
\textit{Requires:} Each type in \texttt{Types} shall be copy constructible.

\textit{Effects:} Copy constructs each element of \texttt{*this} with the corresponding element of \texttt{u}.

\template<\texttt{class... UTypes}>
\texttt{tuples(const tuples<UTypes...>& u)};

7  
\textit{Requires:} Each type in \texttt{Types} shall be copy constructible.

\textit{Effects:} Constructs each element of \texttt{*this} with the corresponding element of \texttt{u}.

\template<\texttt{class... UTypes}>
\texttt{tuples(tuples<UTypes...>&& u)};

8  
\textit{Requires:} Each type in \texttt{Types} shall be copy constructible.

\textit{Effects:} Constructs each element of \texttt{*this} with the corresponding element of \texttt{u}.

\template<\texttt{class U1, class U2}>
\texttt{tuples(const pair<U1, U2>& u)};

9  
\textit{Requires:} Each type in \texttt{Types} shall be copy constructible.

\textit{Effects:} Constructs the first element with \texttt{u.first} and the second element with \texttt{u.second}.

\texttt{Draft}
template <class U1, class U2> tuple(pair<U1, U2>&& u);

Requires: The first type in Types shall shall satisfy the requirements of MoveConstructible (Table 33) from U1 and the second type in Types shall be move-constructible from U2. sizeof...(Types) == 2.

Effects: Constructs the first element with std::move(u.first) and the second element with std::move(u.second).

tuple& operator=(const tuple& u);

Requires: Each type in Types shall be CopyAssignable (Table 36).

Effects: Assigns each element of u to the corresponding element of *this.

Returns: *this

tuple& operator=(tuple&& u);

Requires: Each type in Types shall satisfy the requirements of MoveAssignable (Table 35).

Effects: Move-assigns each element of u to the corresponding element of *this.

Returns: *this.

template <class... UTypes>
tuple& operator=(const tuple<UTypes...>& u);

Requires: Each type in Types shall be Assignable from the corresponding type in UTypes.

Effects: Assigns each element of u to the corresponding element of *this.

Returns: *this

template <class... UTypes>
tuple& operator=(tuple<UTypes...>&& u);

Requires: Each type in Types shall satisfy the requirements of MoveAssignable (Table 35) from the corresponding type in UTypes. sizeof...(Types) == sizeof...(UTypes).

Effects: Move-assigns each element of u to the corresponding element of *this.

Returns: *this.

template <class U1, class U2> tuple& operator=(const pair<U1, U2>& u);

Requires: The first type in Types shall shall satisfy the requirements of MoveAssignabe (Table 35) from U1 and the second type in Types shall satisfy the requirements of MoveAssignabe (Table 35) from U2. sizeof...(Types) == 2.

Effects: Assigns u.first to the first element of *this and u.second to the second element of *this.

Returns: *this

[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 T and if the parameter passed to the constructor is not of type T, but rather a tuple type that is convertible to T. The effect of the converting copy construction is most...]

Draft
likely the same as the effect of the element-wise construction would have been. However, it is 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.  

```cpp
template <class U1, class U2> tuple& operator=(pair<U1, U2>&& u);

40 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.

41 Effects: Assigns `std::move(u.first)` to the first element of `*this` and `std::move(u.second)` to the second
element of `*this`.

42 Returns: `*this`.
```

```cpp
template <class Alloc>
tuple(allocator_arg_t, const Alloc& a);

tuple(allocator_arg_t, const Alloc& a, const Types&...);

tuple(allocator_arg_t, const Alloc& a, const UTypes...);

tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);

tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);
```

43 Requires: Alloc shall be an Allocator (20.1.2).

44 Effects: Equivalent to the preceding constructors except that the allocator argument is passed conditionally to the
constructor of each element. Each member is `allocator constructed` (20.6.2) with a.

### 20.3.1.3 Tuple creation functions

```cpp
template<class... Types>
tuple<VTypes...> make_tuple(Types&&... t);
```

1 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.

Let Ui be decay<Ti>::type for each Ti in Types. Then each Vi in VTypes is X if Ui equals reference_wrapper<X>,
otherwise Vi is U1.

2 Returns: `tuple<VTypes...>(std::forward<Types>(t)...).`
Example:

```cpp
int i; float j;
make_tuple(i, ref(i), cref(j))
```

creates a tuple of type

```cpp
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:

```cpp
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 (Table 34). All the types in UTypes shall be CopyConstructible (Table 34).

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.

```
template <class... TTypes, class... UTypes>
tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&& t, const tuple<UTypes...>& u);
```

Requires: All the types in TTypes shall be MoveConstructible (Table 33). All the types in UTypes shall be CopyConstructible (Table 34).

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.

```
template <class... TTypes, class... UTypes>
tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>& t, tuple<UTypes...>&& u);
```

Requires: All the types in TTypes shall be CopyConstructible (Table 34). All the types in UTypes shall be MoveConstructible (Table 33).
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.

template <class... TTypes, class... UTypes>
tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&& t, tuple<UTypes...>&& u);

Requires: All the types in TTypes shall be MoveConstructible (Table 33). All the types in UTypes shall be MoveConstructible (Table 33).

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.

20.3.1.4 Tuple helper classes

template <class... Types>
class tuple_size<tuple<Types...> >
: public integral_constant<size_t, sizeof...(Types)>{ };

template <int I, class... Types>
class tuple_element<I, tuple<Types...>> {
  public:
    typedef TI type;
};

Requires: 0 <= I and I < sizeof...(Types). The program is ill-formed if I is out of bounds.

Type: TI is the type of the Ith element of Types, where indexing is zero-based.

20.3.1.5 Element access

template <int I, class... types>
typename tuple_element<I, tuple<Types...>> ::type& get(tuple<Types...>&& t);

Requires: 0 <= I and I < sizeof...(Types). The program is ill-formed if I is out of bounds.

Returns: A reference to the Ith element of t, where indexing is zero-based.

template <int I, class... types>
typename tuple_element<I, tuple<Types...>> ::type const& get(const tuple<Types...>& t);

Requires: 0 <= I and I < sizeof...(Types). The program is ill-formed if I is out of bounds.

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]
[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]

20.3.1.6 Relational operators [tuple.rel]

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).

5 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 r_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.

template<class... TTypes, class... UTypes>
bool operator!=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);

5 Returns: !(t == u).

template<class... TTypes, class... UTypes>
bool operator>(const tuple<TTypes...>& t, const tuple<UTypes...>& u);

6 Returns: u < t.

template<class... TTypes, class... UTypes>
bool operator<=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);

7 Returns: !(u < t)

template<class... TTypes, class... UTypes>
bool operator>=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);

8 Returns: !(t < u)

[Note: The above definitions for comparison operators do not require t_tail (or u_tail) to be constructed. It may not even be possible, as t and u are not required to be copy constructible. Also, all comparison operators are short circuited; they do not perform element accesses beyond what is required to determine the result of the comparison. — end note]
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 `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:
    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:
```

Draft
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:

Draft
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 {
    template <class T, T v>
    struct integral_constant {
        static const T value = v;
        typedef T value_type;
        typedef integral_constant<T,v> type;
    };
    typedef integral_constant<bool,true> true_type;
    typedef integral_constant<bool,false> false_type;
}

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

namespace std {
    template <class T, T v>
    struct integral_constant {
        static const T value = v;
        typedef T value_type;
        typedef integral_constant<T,v> type;
    };
    typedef integral_constant<bool,true> true_type;
    typedef integral_constant<bool,false> false_type;
}

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 41: Primary Type Category Predicates

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt; struct is_void;</td>
<td>T is void</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_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>
<tr>
<td>template &lt;class T&gt; struct is_class;</td>
<td>T is a class type but not a union type (3.9.2)</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_function;</td>
<td>T is a function type (3.9.2)</td>
<td></td>
</tr>
</tbody>
</table>

20.4.4.2 Composite type traits

1. These templates provide convenient compositions of the primary type categories, corresponding to the descriptions given in section 3.9.

2. For any given type \( T \), the result of applying one of these templates to \( T \), and to \( cv\)-qualified \( T \) shall yield the same result.

Table 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; struct is_reference;</td>
<td>T is an lvalue reference or an rvalue reference</td>
<td></td>
</tr>
</tbody>
</table>
20.4 Metaprogramming and type traits

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;class T&gt; struct is_arithmetic;</code></td>
<td>T is an arithmetic type (3.9.1)</td>
<td></td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct is_fundamental;</code></td>
<td>T is a fundamental type (3.9.1)</td>
<td></td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct is_object;</code></td>
<td>T is an object type (3.9)</td>
<td></td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct is_scalar;</code></td>
<td>T is a scalar type (3.9)</td>
<td></td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct is_compound;</code></td>
<td>T is a compound type (3.9.2)</td>
<td></td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct is_member_pointer;</code></td>
<td>T is a pointer to non-static data member or non-static member function</td>
<td></td>
</tr>
</tbody>
</table>

20.4.4.3 Type properties

1 These templates provide access to some of the more important properties of types.

2 It is unspecified whether the library defines any full or partial 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.

3 For all of the class templates $X$ declared in this clause, instantiating that template with a template-argument that is a class template specialization may result in the implicit instantiation of the template argument if and only if the semantics of $X$ require that the argument must be a complete type.

Table 43: Type Property Predicates

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;class T&gt; struct is_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) void.</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) void.</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) void.</td>
</tr>
<tr>
<td>Template</td>
<td>Description</td>
<td>Requirements</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
</tbody>
</table>
| template <class T>  
struct is_empty; | 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 `is_empty<B>::value` is false. | T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`. |
| template <class T>  
struct is_polymorphic; | T is a polymorphic class (10.3) | T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`. |
| template <class T>  
struct is_abstract; | T is an abstract class (10.4) | T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`. |
| template <class T>  
struct has_trivial_default_constructor; | T is a trivial type (3.9) or a class type with a trivial default constructor (12.1) or an array of such a class type. | T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`. |
| template <class T>  
struct has_trivial_copy_constructor; | T is a trivial type (3.9) or a reference type or a class type with a trivial copy constructor (12.8). | T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`. |
| template <class T>  
struct has_trivial_assign; | T is neither `const` nor a reference type, and T is a trivial type (3.9) or a class type with a trivial copy assignment operator (12.8). | T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`. |
| template <class T>  
struct has_trivial_destructor; | T is a trivial 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`. |
| template <class T>  
struct has_nothrow_default_constructor; | `has_trivial_default_constructor<T>::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`. |
### 20.4 Metaprogramming and type traits

**General utilities library 500**

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;class T&gt; struct has_nothrow_copy_constructor;</code></td>
<td><code>has_trivial_copy_constructor&lt;T&gt;::value</code> is true or <code>T</code> is a class type with a copy constructor that is known not to throw any exceptions or <code>T</code> is an array of such a class type.</td>
<td><code>T</code> 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_nothrow_assign;</code></td>
<td><code>T</code> is neither <code>const</code> nor a reference type, and <code>has_trivial_assign&lt;T&gt;::value</code> is true or <code>T</code> is a class type with a copy assignment operator taking an lvalue of type <code>T</code> that is known not to throw any exceptions or <code>T</code> is an array of such a class type.</td>
<td><code>T</code> 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_virtualDestructor;</code></td>
<td><code>T</code> has a virtual destructor (12.4)</td>
<td><code>T</code> 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_signed;</code></td>
<td><code>is_arithmetic&lt;T&gt;::value &amp;&amp; T(-1) &lt; T(0)</code></td>
<td></td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct is_unsigned;</code></td>
<td><code>is_arithmetic&lt;T&gt;::value &amp;&amp; T(0) &lt; T(-1)</code></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
```

---

### Example:

```cpp
remove_const<const volatile int>::type // volatile int
remove_const<const int* const>::type // const int*
remove_const<const int&>::type // const int&
```

---

Draft
Table 44: Type Property Queries

<table>
<thead>
<tr>
<th>Template</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt;</td>
<td>alignof(T).</td>
</tr>
<tr>
<td>struct alignment_of;</td>
<td>Precondition: T shall be a complete type, a reference type, or an array of unknown bound, but shall not be a function type or (possibly cv-qualified) void.</td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>If T names an array type, an integer value representing the number of dimensions of T; otherwise, 0.</td>
</tr>
<tr>
<td>struct rank;</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T, template &lt;unsigned I = 0&gt;</td>
<td></td>
</tr>
<tr>
<td>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>

6 [Example:]

// the following assertions hold:
assert(rank<int>::value == 0);
assert(rank<int[2]>::value == 1);
assert(rank<int[1][4]>::value == 2);

— end example ]

7 [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[1][4]>::value == 0);
assert((extent<int[2][1]>::value == 0);
assert((extent<int[2][4], 1>::value == 4);
assert((extent<int[], 1>::value == 4);

— end example ]

20.4.5 Relationships between types

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>template &lt;class T, class U&gt;</td>
<td>T and U name the same type with the same cv-qualifications</td>
<td></td>
</tr>
</tbody>
</table>
20.4 Metaprogramming and type traits

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<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. 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 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: <code>template &lt;class T&gt; typename add_rvalue_reference&lt;T&gt;::type create();</code> <code>To test() { return create&lt;From&gt;(); }</code> [Note: This requirement gives well defined results for reference types, void types, array types, and function types. — end note]</td>
</tr>
</tbody>
</table>

3 [Example:]```c++
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]

1 This sub-clause contains templates that may be used to transform one type to another following some predefined rule.

2 Each of the templates in this subclause shall be a TransformationTrait (20.4.1).

20.4.6.1 Const-volatile modifications [meta.trans.cv]

Table 46: Const-volatile modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
</table>

Draft
template <class T>
struct remove_const;
The member typedef type shall name the same type as T except that any
top-level const-qualifier has been removed. [Example: remove_const<const
volatile int>::type evaluates to volatile int, whereas
remove_const<const int*> is const int*. — end example]

template <class T>
struct remove_volatile;
The member typedef type shall name the same type as T except that any
top-level volatile-qualifier has been removed. [Example:
remove_volatile<const volatile int>::type evaluates to const int,
whereas remove_volatile<volatile int*> is volatile int*. — end
example]

template <class T>
struct remove_cv;
The member typedef type shall be the same as T except that any top-level
cv-qualifier has been removed. [Example: remove_cv<const volatile
int>::type evaluates to int, whereas remove_cv<volatile int*> is const volatile int*.
— end example]

template <class T>
struct add_const;
If T is a reference, function, or top-level const-qualified type, then type shall
name the same type as T, otherwise T const.

template <class T>
struct add_volatile;
If T is a reference, function, or top-level volatile-qualified type, then type shall
name the same type as T, otherwise T volatile.

template <class T>
struct add_cv;
The member typedef type shall name the same type as add_const<typename
add_volatile<T>::type>::type.

20.4.6.2 Reference modifications
[meta.trans.ref]

Table 47: Reference modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
</table>
| template <class T>
struct remove_reference; | If T has type “reference to T1” then the member typedef type shall name T1;
otherwise, type shall name T. |
| template <class T>
struct add_lvalue_reference; | If T names an object or function type then the member typedef type shall name
T&; otherwise, if T names a type “rvalue reference to T1” then the member
typedef type shall name T1&; otherwise, type shall name T. |
| template <class T>
struct add_rvalue_reference | If T names an object or function type then the member typedef type shall name
T&&; otherwise, type shall name T. [Note: This rule reflects the semantics of
reference collapsing. For example, when a type T names a type T1&, the type
add_rvalue_reference<T>::type is not an rvalue reference. — end note] |

20.4.6.3 Sign modifications
[meta.trans.sign]

Table 48: Sign modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Draft
20.4 Metaprogramming and type traits

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 \( \text{sizeof}(T) == \text{sizeof}(\text{type}) \), 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 \( \text{sizeof}(T) == \text{sizeof}(\text{type}) \), 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

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));
assert((is_same<remove_all_extents<int[2][3]>::type, int>::value));
assert((is_same<remove_all_extents<int[][3]>::type, int>::value));

Draft
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 unitialized 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 remove_cv&lt;U&gt;::type.</td>
<td></td>
</tr>
<tr>
<td>template &lt;bool B, class T = void&gt; struct enable_if;</td>
<td>If B is true, the member typedef type shall equal T; otherwise, there shall be no member typedef type.</td>
<td></td>
</tr>
</tbody>
</table>
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 would define aligned_storage as:

```cpp
template <std::size_t Len, std::size_t Alignment>
struct aligned_storage {
    typedef struct {
        alignas(Alignment) unsigned char __data[Len];
    } type;
};
```
— 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

```cpp
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> class result_of; // undefined
    template <class F, class... Args> class result_of<F(Args...)>;

    // 20.5.5, reference_wrapper:
    template <class T> class reference_wrapper;
    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, bitwise operations:
template <class T> struct bit_and;
template <class T> struct bit_or;
template <class T> struct bit_xor;

// 20.5.10, negators:
template <class Predicate> class unary_negate;
template <class Predicate>
  unary_negate<Predicate> not1(const Predicate&);
template <class Predicate> class binary_negate;
template <class Predicate>
  binary_negate<Predicate> not2(const Predicate&);

// 20.5.11, 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;
  .
  .
  .
  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&);

// 20.5.12, 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, Arg2, Result>
ptr_fun(Result (*)(Arg1,Arg2));

// 20.5.13, 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);

// 20.5.14, member function adaptors:
template<class R, class T> unspecified mem_fn(R T::*);

// 20.5.15 polymorphic function wrappers:
class bad_function_call;

template<class R, class... ArgTypes> class function<R(ArgTypes...)>;

Draft
template<
class R, class... ArgTypes>
void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&);

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);

template<
class R, class... ArgTypes>
bool operator!=(unspecified-null-pointer-type, const function<R(ArgTypes...)>&);

// 20.5.16, 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 long>;
template <> struct hash<float>;
template <> struct hash<double>;
template <> struct hash<long double>;
template<
class T> struct hash<T*>;

template <> struct hash<std::string>;
template <> struct hash<std::u16string>;
template <> struct hash<std::u32string>;
template <> struct hash<std::wstring>;
template <> struct hash<std::error_code>;
template <> struct hash<std::thread::id>;

// 20.5.17, reference_closure:
template <class> class reference_closure; // undefined
template <class R, class... ArgTypes>

Draft
```c++
class reference_closure<R(ArgTypes...)>;
}
```

3 [ Example: ] If a C++ program wants to have a by-element addition of two vectors a and b containing double and put the result into a, it can do:

```cpp
transform(a.begin(), a.end(), b.begin(), a.begin(), plus<double>());
```

— end example ]

4 [ Example: ] To negate every element of a:

```cpp
transform(a.begin(), a.end(), a.begin(), negate<double>());
```

— 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 `argument_type` and `result_type` for function objects that take one argument and `first_argument_type`, `second_argument_type`, and `result_type` for function objects that take two arguments.

20.5.1 Definitions [func.def]

1 The following definitions apply to this clause:

2 A `call signature` is the name of a return type followed by a parenthesized comma-separated list of zero or more argument types.

3 A `call wrapper` is an object of a call wrapper type.

4 A `call wrapper type` is a type that holds a callable object and supports a call operation that forwards to that object.

5 A `callable object` is an object of a callable type.

6 A `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 `target object` is the callable object held by a call wrapper object.

20.5.2 Requirements [func.require]

1 Define `INVOKE(f, t1, t2, ..., tN)` as follows:

- `(t1.*f)(t2, ..., tN)` when f is a pointer to a member function of a class T and t1 is an object of type T or a reference to an object of type T or a reference to an object of a type derived from T;

- `((*t1).*f)(t2, ..., tN)` when f is a pointer to a member function of a class T and t1 is not one of the types described in the previous item;

- `t1.*f` when f is a pointer to member data of a class T and t1 is an object of type T or a reference to an object of type T or a reference to an object of a type derived from T;

- `(*t1).*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;
2 Define \texttt{INVOKEx(f, t1, t2, \ldots, tN)} as \texttt{INVOKEx(f, t1, t2, \ldots, tN)} implicitly converted to \texttt{R}.

3 If a call wrapper ([20.5.1]) has a \textit{weak result type} the type of its member type \texttt{result\_type} is based on the type \texttt{T} of the wrapper’s target object ([20.5.1]):

\begin{itemize}
  \item if \texttt{T} is a function, reference to function, or pointer to function type, \texttt{result\_type} shall be a synonym for the return type of \texttt{T};
  \item if \texttt{T} is a pointer to member function, \texttt{result\_type} shall be a synonym for the return type of \texttt{T};
  \item if \texttt{T} is a class type with a member type \texttt{result\_type}, then \texttt{result\_type} shall be a synonym for \texttt{T::result\_type};
  \item otherwise \texttt{result\_type} shall not be defined.
\end{itemize}

4 Every call wrapper [20.5.1] shall be CopyConstructible. A \textit{simple call wrapper} is a call wrapper that is CopyAssignable and whose copy constructor and assignment operator do not throw exceptions. A \textit{forwarding call wrapper} is a call wrapper that can be called with an argument list. [\textit{Note: in a typical implementation forwarding call wrappers have an overloaded function call operator of the form
\begin{verbatim}
  template<class... ArgTypes>
  R operator()(ArgTypes&&... args) cv-qual;
\end{verbatim}
—\textit{end note}]

20.5.3 \textbf{Base} \quad [base]

1 The following classes are provided to simplify the typedefs of the argument and result types:

\begin{verbatim}
namespace std {
  template <class Arg, class Result>
  struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
  };
}

namespace std {
  template <class Arg1, class Arg2, class Result>
  struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
  };
}
\end{verbatim}

20.5.4 \textbf{Function object return types} \quad [func.ret]

Draft
Given an rvalue \( fn \) of type \( Fn \) and values \( t_1, t_2, \ldots, t_N \) of types \( T_1, T_2, \ldots, T_N \) in \( \text{ArgTypes} \), respectively, the type member is the result type of the expression \( fn(t_1, t_2, \ldots, t_N) \). The values \( t_i \) are lvalues when the corresponding type \( T_i \) is a reference type, and rvalues otherwise.

### 20.5.5 Class template reference_wrapper

```cpp
class std { 
    template <class T> class reference_wrapper 
    : public unary_function<T1, R> // see below 
    : public binary_function<T1, T2, R> // see below 
    { 
        public : 
            // types 
            typedef T type; 
            typedef see below 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); 

            // assignment 
            reference_wrapper& operator=(const reference_wrapper<T>& x); 

            // access 
            operator T& () const; 
            T& get() const; 

            // invocation 
            template <class... ArgTypes> 
            typename result_of<T(ArgTypes...)>::type 
            operator() (ArgTypes&&...) const; 
    }; 
}
```

1. \( \text{reference}\_\text{wrapper}<T> \) is a CopyConstructible and Assignable wrapper around a reference to an object of type \( T \).
2. \( \text{reference}\_\text{wrapper} \) has a weak result type ([20.5.2]).
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>`

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

**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);`

*Postconditions:* `*this` stores a reference to `x.get()`.

*Throws:* nothing.

### 20.5.5.3 `reference_wrapper` access

`T& () const;`

*Returns:* The stored reference.

*Throws:* nothing.

`T& get() const;`
Returns: The stored reference.

Throws: nothing.

20.5.5.4 reference_wrapper invocation

```
template <class... ArgTypes>
  typename result_of<T(ArgTypes... )>::type
  operator()(ArgTypes&&... args) const;
```

Returns: \( \text{INVOKE} \) (get(), std::forward<ArgTypes>(args)\ldots). (20.5.2)

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);
```

Returns: reference_wrapper<T>(t)

Throws: nothing.

```
template <class T> reference_wrapper<T> ref(reference_wrapper<T>t);
```

Returns: ref(t.get())

Throws: nothing.

```
template <class T> reference_wrapper<const T> cref(const T& t);
```

Returns: reference_wrapper<const T>(t)

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 * 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.

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.

template <class T> struct greater : binary_function<T,T,bool> {
    bool operator()(const T& x, const T& y) const;
};
operator() returns x > y.

template <class T> struct less : binary_function<T,T,bool> {

20.5 Function objects

bool operator()(const T& x, const T& y) const;
};

operator() returns x < y.

template <class T> struct greater_equal : binary_function<T,T,bool> {
  bool operator()(const T& x, const T& y) const;
};

operator() returns x >= y.

template <class T> struct less_equal : binary_function<T,T,bool> {
  bool operator()(const T& x, const T& y) const;
};

operator() returns x <= 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.

20.5.8 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).

template <class T> struct logical_and : binary_function<T,T,bool> {
  bool operator()(const T& x, const T& y) const;
};

operator() returns x && y.

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 <class T> struct logical_not : unary_function<T,bool> {
  bool operator()(const T& x) const;
};

operator() returns !x.

20.5.9 Bitwise operations

The library provides basic function object classes for all of the bitwise operators in the language (5.11, 5.13, 5.12).

template <class T> struct bit_and : binary_function<T,T,T> {
  T operator()(const T& x, const T& y) const;
};

operator() returns x & y.
template <class T> struct bit_or : binary_function<T,T,T> {
    T operator()(const T& x, const T& y) const;
};

operator() returns \(x \mid y\).

template <class T> struct bit_xor : binary_function<T,T,T> {
    T operator()(const T& x, const T& y) const;
};

operator() returns \(x \oplus y\).

### 20.5.10 Negators

Negators `not1` and `not2` take a unary and a binary predicate, respectively, and return their complements (5.3.1).

```cpp
template <class 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;
};

operator() returns \(!\)pred(x).
```

```cpp
template <class Predicate>
unary_negate<Predicate> not1(const Predicate& pred);

Returns: unary_negate<Predicate>(pred).
```

```cpp
template <class 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;
};

operator() returns \(!\)pred(x,y).
```

```cpp
template <class Predicate>
binary_negate<Predicate> not2(const Predicate& pred);

Returns:

binary_negate<Predicate>(pred).
```
20.5 Function objects

20.5.11 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 \( f \) of two arguments and a value \( x \) and return a function object of one argument constructed out of \( f \) with the first or second argument correspondingly bound to \( x \).

20.5.11.1 Function object binders

This subclause describes a uniform mechanism for binding arguments of function objects.

20.5.11.1.1 Class template is_bind_expression

```cpp
namespace std {
    template<class T> struct is_bind_expression {
        static const bool value = see below;
    };
}
```

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.

```cpp
static const bool value;
```

true if \( T \) is a type returned from bind, false otherwise.

20.5.11.1.2 Class template is_placeholder

```cpp
namespace std {
    template<class T> struct is_placeholder {
        static const int value = see below;
    };
}
```

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.

```cpp
static const int value;
```

value is \( J \) if \( T \) is the type of \( \text{std::placeholders::}_J \), 0 otherwise.

20.5.11.1.3 Function template bind

```cpp
template<class F, class... BoundArgs>
unspecified bind(F f, BoundArgs... bound_args);
```

Requires: \( F \) and each \( T_i \) in \( \text{BoundArgs} \) shall be CopyConstructible. \( \text{INVOKE} \ (f, w_1, w_2, \ldots, w_N) \) shall be a valid expression for some values \( w_1, w_2, \ldots, w_N \), where \( N = \text{sizeof...}(\text{bound_args}) \).
Returns: A forwarding call wrapper g with a weak result type (20.5.2). The effect of \( g(u_1, u_2, \ldots, u_M) \) shall be \( \text{INVOK}(f, v_1, v_2, \ldots, v_N, \text{result_of}<F\ cv(V1, V2, \ldots, VN)>::\text{type}) \), where \( cv \) represents the \( cv \)-qualifiers of \( g \) and the values and types of the bound arguments \( v_1, v_2, \ldots, v_N \) are determined as specified below.

Throws: Nothing unless the copy constructor of \( F \) or of one of the types in the BoundArgs... pack expansion throws an exception.

\[
\text{template}<\text{class } R, \text{class } F, \text{class... } \text{BoundArgs}>
\text{unspecified } \text{bind}(F \ f, \text{BoundArgs... } \text{bound_args});
\]

Requires: \( F \) and each \( Ti \) in BoundArg shall be CopyConstructible. \( \text{INVOK}(f, w_1, w_2, \ldots, w_N) \) shall be a valid expression for some values \( w_1, w_2, \ldots, w_N \), where \( N = \text{sizeof...(bound_args)} \).

Returns: A forwarding call wrapper g with a nested type result_type defined as a synonym for \( R \). The effect of \( g(u_1, u_2, \ldots, u_M) \) shall be \( \text{INVOK}(f, v_1, v_2, \ldots, v_N, R) \), where the values and types of the bound arguments \( v_1, v_2, \ldots, v_N \) are determined as specified below.

Throws: Nothing unless the copy constructor of \( F \) or of one of the types in the BoundArgs... pack expansion throws an exception.

The values of the bound arguments \( v_1, v_2, \ldots, v_N \) and their corresponding types \( V_1, V_2, \ldots, V_N \) depend on the 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 \( ti\text{.get()} \) and its type \( Vi \) is \( T \& \);
- if the value of std::is_bind_expression<Ti>::value is true the argument is \( ti(u_1, u_2, \ldots, u_M) \) 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 &.

### 20.5.11.4 Placeholders

\[
\text{namespace std {}
\text{namespace placeholders {}
\text{// M is the implementation-defined number of placeholders}
\text{extern unspecified _1;}
\text{extern unspecified _2;}
\text{...}
\text{extern unspecified _M;}
\text{}}
\text{}}
\]

Draft
20.5 Function objects

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.

20.5.12 Adaptors for pointers to functions

To allow pointers to (unary and binary) functions to work with function adaptors the library provides:

```cpp
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 (*f)(Arg));
```

Returns: \( \text{pointer\_to\_unary\_function}<\text{Arg}, \text{Result}>(f) \).

```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 (*f)(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 (*f)(Arg1, Arg2));
```

Returns: \( \text{pointer\_to\_binary\_function}<\text{Arg1}, \text{Arg2}, \text{Result}>(f) \).

[Example:

```cpp
int compare(const char*, const char*);
replace_if(v.begin(), v.end(),
    not1(bind2nd(ptr_fun(compare), "abc")), "def");
```

replaces each \textit{abc} with \textit{def} in sequence \textit{v}. — end example ]

20.5.13 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 20.5.12.

```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.

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;
};

mem_fun1_t calls the member function it is initialized with given a pointer argument and an additional argument of the appropriate type.

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).

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.

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.

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).
20.5 Function objects

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.

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;
};

const_mem_fun1_t calls the member function it is initialized with given a pointer argument and an additional argument of the appropriate type.

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);

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).

template <class S, class T> class const_mem_fun_ref_t
  : public unary_function<T, S> {
public:
  explicit const_mem_fun_ref_t(S (T::*p)() const);
  S operator()(const T& p) const;
};

const_mem_fun_ref_t calls the member function it is initialized with given a reference argument.

template <class S, class T, class A> class const_mem_fun1_ref_t
  : public binary_function<T, A, S> {
public:
  explicit const_mem_fun1_ref_t(S (T::*p)(A) const);
  S operator()(const T& p, A x) const;
};

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.

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);

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).

20.5.14 Function template mem_fn

\[\text{template<}\text{class } R, \text{ class } T \text{> unspecified \ mem_fn(R T::* pm);}\]

1 \text{Returns: A simple call wrapper ([20.5.1]) } fn \text{ such that the expression } \text{fn(t, a2, ..., aN)} \text{ is equivalent to } INVOKE(pm, t, a2, ..., aN) \text{ ([20.5.2]). } fn \text{ shall have a nested type result_type that is a synonym for the return type of pm when pm is a pointer to member function.}

2 \text{The simple call wrapper shall be derived from std::unary_function<cv T*, Ret> when pm is a pointer to member function with cv-qualifier cv and taking no arguments, where Ret is pm's return type.}

3 \text{The simple call wrapper shall be derived from std::binary_function<cv T*, T1, Ret> when pm is a pointer to member function with cv-qualifier cv and taking one argument of type T1, where Ret is pm's return type.}

4 \text{Throws: Nothing.}

5 \text{Remarks: Implementations may implement mem_fn as a set of overloaded function templates.}

20.5.15 Polymorphic function wrappers

\[\text{This subclause describes a polymorphic wrapper class that encapsulates arbitrary function objects.}\]

20.5.15.1 Class bad_function_call

\[\text{namespace std \{}\]
\[\text{class bad_function_call : public std::exception } \{
\text{\text{\text{public:}}}
\text{\text{\text{// 20.5.15.1.1, constructor:}}}
\text{\text{bad_function_call();}}
\text{\text{\}};\]
\[\text{\text{// namespace std}}\]

20.5.15.1.1 bad_function_call constructor

\[\text{bad_function_call();}\]

1 \text{Effects: constructs a bad_function_call object.}

20.5.15.2 Class template function

\[\text{namespace std \{}\]
\[\text{template<class> class function; // undefined}\]
\[\text{template<class R, class... ArgTypes>}\]
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;

    // 20.5.15.2.1, construct/copy/destroy:
    explicit function();
    function(unspecified-null-pointer-type );
    function(const function&);
    function(function&&);
    template<class F> function(F);
    template<class F, class A> function(F, const A&);
    template<class A> function(allocator_arg_t, const A&);
    template<class A> function(allocator_arg_t, const A&,  
      unspecified-null-pointer-type);
    template<class A> function(allocator_arg_t, const A&,
      const function&);
    template<class A> function(allocator_arg_t, const A&,
      function&);
    template<class F, class A> function(allocator_arg_t, const A&,
      F);
    template<class F, class A> function(allocator_arg_t, const A&,
      function&);
    template<class F, class A> function(allocator_arg_t, const A&,
      F);
    template<class A> function(allocator_arg_t, const A&,
      allocator_arg_t);
    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();

    // 20.5.15.2.2, function modifiers:
    void swap(function&);
    template<class F, class A> void assign(F, const A&);

    // 20.5.15.2.3, function capacity:
    explicit operator bool() const;

    // deleted overloads 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;

    // 20.5.15.2.4, function invocation:
    R operator()(ArgTypes...) const;

Draft
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 values \( t_1, t_2, \ldots, t_N \) 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 \( \text{void} \), convertible to \( R \).

The function class template is a call wrapper (20.5.1) whose call signature ([20.5.1]) is \( R(\text{ArgTypes}...) \).

### 20.5.15.2.1 Function construct/copy/destroy

```cpp
explicit function();
```

**Postconditions:** !\*this.

**Throws:** nothing.
function(const function& f);

Postconditions: !*this if !f; otherwise, *this targets a copy of f.target().
Throws: shall not throw exceptions if f’s target is a function pointer or a function object passed via reference_wrapper. Otherwise, may throw 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 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 a 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.

function& operator=(const function& f);

Effects: function(f).swap(*this);
Returns: *this

function& operator=(function&& f);

Effects: Replaces the target of *this with the target of f, leaving f in a valid but unspecified state.
Returns: *this

function& operator=(unspecified-null-pointer-type);
Effects: If *this != NULL, destroys the target of this.
Postconditions: !(this).
Returns: *this

template<class F> function& operator=(F f);
Effects: function(f).swap(*this);
Returns: *this

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.

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.

20.5.15.2.2 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)

20.5.15.2.3 function capacity

explicit operator bool() const
Returns: true if *this has a target, otherwise false.
Throws: nothing.
20.5 Function objects

20.5.15.2.4 function invocation

R operator()(ArgTypes... args) const

1 Effects: \textit{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...

2 Returns: Nothing if R is void, otherwise the return value of \textit{INVOKE} (f, t1, t2, ..., tN, R).

3 Throws: bad_function_call if *this; otherwise, any exception thrown by the wrapped function object.

20.5.15.2.5 function target access

const std::type_info& target_type() const;

1 Returns: If *this has a target of type T, typeid(T); otherwise, typeid(void).

2 Throws: nothing.

template<typename T> T* target();
template<typename T> const T* target() const;

3 Requires: T shall be a function object type that is Callable (20.5.15.2) 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.

20.5.15.2.6 null pointer comparison operators

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.
20.5.15.2.7 specialized algorithms

template<class R, class... ArgTypes>
  void swap(function<R(ArgTypes...)> & f1, function<R(ArgTypes...)> & f2);

Effects: f1.swap(f2);

20.5.16 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, std::wstring, std::error_code, and std::thread::id.

namespace std {
  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.

20.5.17 Class template reference_closure

namespace std {
  template<class> class reference_closure; // undefined
}

template<class R, class... ArgTypes>
  class reference_closure<R (ArgTypes...)> {
  public:
    typedef R result_type;
    typedef T1 argument_type;       // iff sizeof...(ArgTypes) == 1 and ArgTypes contains T1
    typedef T1 first_argument_type; // iff sizeof...(ArgTypes) == 2 and ArgTypes contains T1, T2
    typedef T2 second_argument_type; // iff sizeof...(ArgTypes) == 2 and ArgTypes contains T1, T2

    // 20.5.17.1, construct/copy/destroy:
    reference_closure() = default;
    reference_closure(const reference_closure&) = default;
    constexpr reference_closure(nullptr_t);
    reference_closure& operator=(const reference_closure&) = delete;
    reference_closure& operator=(nullptr_t);
    ~reference_closure() = default;

    // 20.5.17.2, observer:
    explicit operator bool() const;

    // 20.5.17.3, invocation:
    R operator()(ArgTypes...) const;

    Draft
20.5 Function objects

General utilities library

};

// 20.5.17.4, comparisons:
template <class R, class... ArgTypes>
    bool operator==(const reference_closure<R (ArgTypes...)>&, nullptr_t);
template <class R, class... ArgTypes>
    bool operator==(nullptr_t, const reference_closure<R (ArgTypes...)>&);
template <class R, class... ArgTypes>
    bool operator!=(const reference_closure<R(ArgTypes...)>&, nullptr_t);
template <class R, class... ArgTypes>
    bool operator!=(nullptr_t, const reference_closure<R (ArgTypes...)>&);

The reference_closure class template represents reference-only closures (5.1.1).

A reference_closure object f of type F is Callable for argument types T1, T2, ..., TN in ArgTypes and a return
type R if, given lvalues t1, t2, ..., tN of types T1, T2, ..., TN, respectively, \textit{INVOKE}(f, t1, t2, ..., tN) is
well-formed and, if R is not void, convertible to R.

The instances of reference_closure class template are trivial and standard-layout classes (3.9).

Unless otherwise specified, the functions in this section do not throw exceptions.

20.5.17.1 Construct, copy, destroy [func.referenceclosure.cons]

    explicit reference_closure();

    \textit{Postcondition: None — the object state is undefined.}

    reference_closure(const reference_closure& f)

    \textit{Postcondition: *this is a copy of f.}

    reference_closure(nullptr_t)

    \textit{Postcondition: !*this}

    \textit{Returns: *this}

    ~reference_closure();

    \textit{Effects: destroys the object.}

20.5.17.2 Observer [func.referenceclosure.obs]

    explicit operator bool() const;

    \textit{Returns: true if *this was constructed or copied from a closure, false if *this was constructed or copied
from an object of type nullptr_t, undefined otherwise.}
20.5.17.3 Invocation

R operator()(ArgTypes... args) const;

Effects: Undefined if *this was default constructed, constructed from an object of type nullptr_t or copied from such. Otherwise, invokes the closure with the given arguments.

Returns: Nothing if R is void, otherwise the return value of the closure.

Throws: Any exception thrown by the wrapped function object.

20.5.17.4 Comparison

template <class R, class... ArgTypes>
bool operator==(const reference_closure<R(ArgTypes...)>& f, nullptr_t);

template <class R, class... ArgTypes>
bool operator==(nullptr_t, const reference_closure<R(ArgTypes...)>& f);

Returns: if

template <class R, class... ArgTypes>
bool operator!=(const reference_closure<R(ArgTypes...)>& f, nullptr_t);

template <class R, class... ArgTypes>
bool operator!=(nullptr_t, const reference_closure<R(ArgTypes...)>& f);

Returns: (bool)f

20.6 Memory

Header <memory> synopsis

namespace std {

    // 20.6.1, allocator argument tag
    struct allocator_tag_t;
    const allocator_arg_t allocator_arg = allocator_arg_t();

    // 20.6.2, allocator-related traits
    template <class T, class Alloc> struct uses_allocator;
    template <class Alloc> struct is_scoped_allocator;
    template <class T> struct constructible_with_allocator_suffix;
    template <class T> struct constructible_with_allocator_prefix;

    // 20.6.3, allocation propagation traits
    template <class Alloc> struct allocator_propagate_never;
    template <class Alloc> struct allocator_propagate_on_copy_construction;
    template <class Alloc> struct allocator_propagate_on_move_assignment;
    template <class Alloc> struct allocator_propagate_on_copy_assignment;
    template <class Alloc> struct allocator_propagation_map;

    // 20.6.5, the default allocator:
    template <class T> class allocator;
template <> class allocator<void>;
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();

// 20.6.6. scoped allocator adaptor
template <class OuterA, class InnerA = void>
class scoped_allocator_adaptor;
template <class Alloc>
class scoped_allocator_adaptor<Alloc, void>;
template <class OuterA, class InnerA>
struct is_scoped_allocator<scoped_allocator_adaptor<OuterA, InnerA> >
  : true_type { };  // 20.6.6.1

template <class OuterA1, class OuterA2, class InnerA>
struct allocator_propagate_never<scoped_allocator_adaptor<OuterA1, InnerA> >
  : true_type { 
struct allocator_propagate_never<scoped_allocator_adaptor<OuterA2, InnerA> >
  : true_type { };  // 20.6.6.2

// 20.6.7. 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);

// 20.6.9. construct element
template <class Alloc, class T, class... Args>
void construct_element(Alloc& alloc, T & r, Args&&... args);  // 20.6.9.1

// 20.6.10. 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);  // 20.6.10.2

// 20.6.11 Class unique_ptr:
template <class X> class unique_ptr;
template <class X> class default_delete;

// 20.6.12.1, Class bad_weak_ptr:
class bad_weak_ptr;

// 20.6.12.2, Class template shared_ptr:
template<class T> class shared_ptr;

// 20.6.12.2.7, 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);

// 20.6.12.2.9, shared_ptr specialized algorithms:
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b);

// 20.6.12.2.10, 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);

// 20.6.12.2.8, 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);

// 20.6.12.11, shared_ptr get_deleter:
template<class D, class T>
  D* get_deleter(shared_ptr<T> const& p);

// 20.6.12.3, Class template weak_ptr:
template<class T> class weak_ptr;

// 20.6.12.3.6, weak_ptr comparison:
template<class T, class U>
  bool operator<(weak_ptr<T> const& a, weak_ptr<U> const& b);

// 20.6.12.3.7, weak_ptr specialized algorithms:
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b);

// 20.6.12.4, Class enable_shared_from_this:
template<class T> class enable_shared_from_this;

// 20.6.13, 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 Allocator argument tag

namespace std {
    struct allocator_arg_t {
        const allocator_arg_t allocator_arg = allocator_arg_t();
    }
}

The allocator_arg_t struct is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, several types (see pair, 20.2.3) have constructors with allocator_arg_t as the first argument, immediately followed by an argument of a type that satisfies the Allocator requirements (20.1.2).

20.6.2 Allocator-related traits

template <class T, class Alloc> struct uses_allocator;

Remark: Automatically detects whether T has a nested allocator_type that is convertible from Alloc. Meets the BinaryTypeTrait requirements (20.4.1). A program may specialize this type to derive from true_type for a user-defined type T that does not have a nested allocator_type but is nonetheless constructible using the specified Alloc.

Remark: uses_allocator<T, Alloc> shall be derived from true_type if Convertible<Alloc, T::allocator_type>, otherwise derived from false_type.

The class templates is_scoped_allocator, constructible_with_allocator_suffix, and constructible_-with_allocator_prefix shall meet the UnaryTypeTrait requirements (20.4.1). Each of these templates shall be publicly derived, directly or indirectly, from true_type if the corresponding condition is true, otherwise from false_type. A program may specialize these traits for user-defined types, provided that each user-defined type meets the requirements of the trait. [Note: These templates are elective traits; they are not computed automatically by determining an intrinsic property of the type, but rather indicate a deliberate choice by the author of the type. A program is never required to specialize these traits. — end note]

template <class Alloc> struct is_scoped_allocator : false_type { }

[ Note: If a specialization is_scoped_allocator<Alloc> is derived from true_type, it indicates that Alloc is a scoped allocator. A scoped allocator specifies the memory resource to be used by a container (as all allocators do) and also specifies an inner allocator resource to be used by every element of the container. — end note]

Requires: If a specialization is_scoped_allocator<Alloc> is derived from true_type, Alloc shall have a nested type inner_allocator_type and a member function inner_allocator() which is callable with no arguments and which returns an object of a type that is convertible to inner_allocator_type.

template <class T> struct constructible_with_allocator_suffix : false_type { }

[ Note: If a specialization constructible_with_allocator_suffix<T> is derived from true_type, it indicates that T can be constructed with an allocator as its last constructor argument. Ideally, all constructors of T (including copy and move constructors) should have a variant that accepts a final argument of allocator_type. — end note]

Requires: If a specialization constructible_with_allocator_suffix<T> is derived from true_type, T shall have a nested type allocator_type and at least one constructor for which allocator_type is the last parameter. If not all constructors of T can be called with a final allocator_type argument and T is used in a context where a container must call such a constructor, the program is ill-formed. [Example:
template <class T, class A = allocator<T> > struct Z {
    typedef A allocator_type;

    // Default constructor with optional allocator suffix
    Z(const allocator_type& a = allocator_type());

    // Copy constructor and allocator-extended copy constructor
    Z(const Z& zz);
    Z(const Z& zz, const allocator_type& a);
};

// Specialize trait for class template Z
template <class T, class A = allocator<T> >
struct constructible_with_allocator_suffix<Z<T,A> > : true_type { };
A type T is \textit{prefix constructible} with an allocator type Alloc if \texttt{uses_allocator<T, Alloc>::value} is true and \texttt{constructible_with_allocator_prefix<T>::value} is true. A type T is \textit{suffix constructible} with an allocator type Alloc if \texttt{uses_allocator<T, Alloc>::value} is true and \texttt{constructible_with_allocator_suffix<T>::value} is true. When an object of type T is \textit{allocator constructed} with an allocator object a of type Alloc, its constructor is selected according to the following rules:

- if T is \textit{prefix constructible} with Alloc, the object is constructed with \texttt{allocator_arg} and a as the first two constructor arguments.
- if T is \textit{suffix constructible} with Alloc, the object is constructed with a as the last constructor argument.
- otherwise, the object is constructed without an allocator.

### 20.6.3 Allocator propagation traits

\begin{verbatim}
template <class Alloc> struct allocator_propagate_never :
    false_type {}

    Requires: Alloc shall be an Allocator (20.1.2).

    [Note: If specialized to derive from true_type for a specific allocator type, indicates that a container using the specified Alloc should not copy or move the allocator (when) the container is copy-constructed, move-constructed, copy-assigned, moved-assigned, or swapped. —end note]
\end{verbatim}

\begin{verbatim}
template <class Alloc> struct allocator_propagate_on_copyConstruction :
    false_type {}

    Requires: Alloc shall be an Allocator (20.1.2).

    [Note: If specialized to derive from true_type for a specific allocator type, indicates that a container using the specified Alloc should copy or move the allocator when the container is copy constructed or move constructed, but not when the container is copy assigned, moved assigned, or swapped. —end note]
\end{verbatim}

\begin{verbatim}
Default behavior: The unspecialized trait derives from true_type if none of allocator_propagate_never, allocator_propagate_on_move_assignment, or allocator_propagate_on_copy_assignment is derived from true_type for the given type Alloc. Otherwise, it derives from false_type.
\end{verbatim}

\begin{verbatim}
template <class Alloc> struct allocator_propagate_on_move_assignment :
    false_type {}

    Requires: Alloc shall be an Allocator (20.1.2).

    [Note: if specialized to derive from true_type for specific allocator type, indicates that a container using the specified Alloc should copy or move the allocator when the container is copy constructed, move constructed, move assigned, or swaped but not when the container is copy assigned. —end note]
\end{verbatim}

\begin{verbatim}
Template <class Alloc> struct allocator_propagate_on_copy_assignment :
    false_type {}
\end{verbatim}
---

537 General utilities library

20.6 Memory

8

Requires: Alloc shall be an Allocator (20.1.2).

9

[Note: If specialized to derive from true_type for a specific allocator type, indicates that a container using the
specified Alloc should copy or move the allocator when the container is copy constructed, move constructed,
move assigned, swapped or copy assigned. — end note]

20.6.4 Allocator propagation map

[allocator.propagation.map]

template <class Alloc> struct allocator_propagation_map {
  static Alloc select_for_copy_construction(const Alloc&);
  static void move_assign(Alloc& to, Alloc& from);
  static void copy_assign(Alloc& to, Alloc& from);
  static void swap(Alloc& a, Alloc& b);
}

Requires: Exactly one propagation trait shall derive from true_type for Alloc.

[Note: The allocator_propagation_map provides functions to be used by containers for manipulating allocators
during construction, assignment, and swap operations. The implementations of the functions above are
dependent on the allocator propagation traits of the specific Alloc. — end note]

static Alloc select_for_copy_construction(const Alloc&);

Returns: Alloc() if allocator_propagate_never<Alloc>::value is true, otherwise x.

static void move_assign(Alloc& to, Alloc& from);

Effects: If allocator_propagate_on_move_assignment<Alloc>::value is true or if allocator_propagate_on_copy_assignment<Alloc>::value is true, assigns to = forward(from); otherwise does nothing.

static void copy_assign(Alloc& to, Alloc& from);

Effects: If allocator_propagate_on_copy_assignment<Alloc>::value is true, assigns to = from; otherwise does nothing.

static void swap(Alloc& a, Alloc& b);

Effects: If allocator_propagate_on_move_assignment<Alloc>::value is true or if allocator_propagate_on_copy_assignment<Alloc>::value is true, exchanges the values of a and b; otherwise, if a == b, does nothing; otherwise the behavior is undefined.

20.6.5 The default allocator

[default分配]

declare std {
  template <class T> class allocator;

  // specialize for void:
  template <> class allocator<void> {
    public:
      typedef void* pointer;

Draft
typedef const void* const_pointer;
// reference-to-void members are impossible.
typedef void value_type;
typedef allocator<void>::const_pointer
    // reference-to-void members are impossible.
typedef allocator<void>::const_pointer
    // reference-to-void members are impossible.
typedef allocator<void>::const_pointer
    // reference-to-void members are impossible.
typedef allocator<void>::const_pointer
    // reference-to-void members are impossible.
typedef allocator<void>::const_pointer
    // reference-to-void members are impossible.
typedef allocator<void>::const_pointer
    // reference-to-void members are impossible.

allocator() throw();
allocator(const allocator&) throw();
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... Args> void construct(pointer p, Args&&... args);
void destroy(pointer p);

20.6.5.1 allocator members

pointer address(reference x) const;
const_pointer address(const_reference x) const;

pointers:
The actual address of the object referenced by x, even in the presence of an overloaded operator&.

const_pointer address(const_reference x) const;

Returns: The actual address of the object referenced by x, even in the presence of an overloaded operator&.

pointer allocate(size_type n, allocator<void>::const_pointer hint = 0);

[Note: In a container member function, the address of an adjacent element is often a good choice to pass for the hint argument. — end note]
Returns: a pointer to the initial element of an array of storage of size \( n \times \text{sizeof}(T) \), 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.

```cpp
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.

```cpp
size_type max_size() const throw();
```

Returns: the largest value \( N \) for which the call `allocate(N,0)` might succeed.

```cpp
void construct(pointer p, const_reference val);
```

Effects: `::new((void*)p) T(val)`

```cpp
template <class... Args> void construct(pointer p, Args&&... args);
```

Effects: `::new((void*)p) T(std::forward<Args>(args)...)`

```cpp
void destroy(pointer p);
```

Effects: \( p->\sim T() \)

### 20.6.5.2 allocator globals

```cpp
template <class T1, class T2>
bool operator==(const allocator<T1>&, const allocator<T2>&) throw();
```

Returns: `true`.

```cpp
template <class T1, class T2>
bool operator!=(const allocator<T1>&, const allocator<T2>&) throw();
```

Returns: `false`.

### 20.6.6 Scoped allocator adaptor

The `scoped_allocator_adaptor` class template is an allocator template that specifies the memory resource (the outer allocator) to be used by a container (as any other allocator does) and also specifies an inner allocator resource to be used by every element in the container. This adaptor is instantiated with outer and inner allocator types. If instantiated with only one allocator type (i.e., the second type is `void`), the same allocator type is used for both the outer and inner...
allocator types and the same allocator instance is used for both the outer and inner allocator instances. The interface is specialized for the single-allocator case such that it takes only one allocator instance argument in the constructor, versus two allocators for the general case. Otherwise, the interface to the specialized and general cases are the same. A scoped_allocator_adaptor that is instantiated with two identical parameters is different than an adaptor instantiated with only one parameter: the former may be constructed with different instances of outer and inner allocators whereas the second may be constructed only with one allocator instance. [Note: The scoped_allocator_adaptor is derived from the outer allocator type so it can be substituted for the outer allocator type in most expressions. —end note]

namespace std {
    template<typename OuterA, typename InnerA = void>
    class scoped_allocator_adaptor;
    template<typename OuterA>
    class scoped_allocator_adaptor<OuterA, void> : public OuterA {
        public:
            // outer and inner allocator types are the same:
            typedef OuterA outer_allocator_type;
            typedef OuterA inner_allocator_type;

            typedef typename outer_allocator_type::size_type size_type;
            typedef typename outer_allocator_type::difference_type difference_type;
            typedef typename outer_allocator_type::pointer pointer;
            typedef typename outer_allocator_type::const_pointer const_pointer;
            typedef typename outer_allocator_type::reference reference;
            typedef typename outer_allocator_type::const_reference const_reference;
            typedef typename outer_allocator_type::value_type value_type;

            template <typename _Tp>
            struct rebind {
                typedef scoped_allocator_adaptor<braced<OuterA::template rebind<_Tp>::other, void> other);
            };

            scoped_allocator_adaptor();
            scoped_allocator_adaptor(scoped_allocator_adaptor&");
            scoped_allocator_adaptor(const scoped_allocator_adaptor&");
            scoped_allocator_adaptor(OuterA&& outerAlloc);
            scoped_allocator_adaptor(const OuterA& outerAlloc);

            template <typename OuterA2>
            scoped_allocator_adaptor(scoped_allocator_adaptor<OuterA2, void>&");
            template <typename OuterA2>
            scoped_allocator_adaptor(const scoped_allocator_adaptor<OuterA2, void>&");

            ~scoped_allocator_adaptor();

            pointer address(reference x) const;
            const_pointer address(const_reference x) const;

            pointer allocate(size_type n);
    template <typename _HintP>
pointer allocate(size_type n, _HintP u);
void deallocate(pointer p, size_type n);
size_type max_size() const;
template<typename... Args>
  void construct(pointer p, Args&&... args);
void destroy(pointer p);

const outer_allocator_type& outer_allocator();
const inner_allocator_type& inner_allocator();
}

template<typename OuterA, typename InnerA>
class scoped_allocator_adaptor : public OuterA {
public:
typedef OuterA outer_allocator_type;
typedef InnerA inner_allocator_type;
typedef typename outer_allocator_type::size_type size_type;
typedef typename outer_allocator_type::difference_type difference_type;
typedef typename outer_allocator_type::pointer pointer;
typedef typename outer_allocator_type::const_pointer const_pointer;
typedef typename outer_allocator_type::reference reference;
typedef typename outer_allocator_type::const_reference const_reference;
typedef typename outer_allocator_type::value_type value_type;
template<typename _Tp>
struct rebind {
  typedef scoped_allocator_adaptor<OuterA::template rebind<_Tp>::other,
                                InnerA> other;
};

scoped_allocator_adaptor();
scoped_allocator_adaptor(outer_allocator_type&& outerAlloc,
                        inner_allocator_type&& innerAlloc);
scoped_allocator_adaptor(const outer_allocator_type& outerAlloc,
                          const inner_allocator_type& innerAlloc);
scoped_allocator_adaptor(scoped_allocator_adaptor&& other);
scoped_allocator_adaptor(const scoped_allocator_adaptor& other);

template<typename OuterAlloc2>
scoped_allocator_adaptor(
  scoped_allocator_adaptor<OuterAlloc2&,InnerA>&&);

template<typename OuterAlloc2>
scoped_allocator_adaptor(
  const scoped_allocator_adaptor<OuterAlloc2&,InnerA>&);
~scoped_allocator_adaptor();

pointer address(reference x) const;
const_pointer address(const_reference x) const;
pointer allocate(size_type n);
template<typename _HintP>
    pointer allocate(size_type n, _HintP u);
void deallocate(pointer p, size_type n);
size_type max_size() const;

template<class... Args>
    void construct(pointer p, Args&&... args);
void destroy(pointer p);

const outer_allocator_type& outer_allocator() const;
const inner_allocator_type& inner_allocator() const;

} // end of class scoped_allocator_adaptor

template<typename OuterA1, typename OuterA2, typename InnerA>
bool operator==(const scoped_allocator_adaptor<OuterA1,InnerA>& a,
      const scoped_allocator_adaptor<OuterA2,InnerA>& b);

template<typename OuterA1, typename OuterA2, typename InnerA>
bool operator!=(const scoped_allocator_adaptor<OuterA1,InnerA>& a,
      const scoped_allocator_adaptor<OuterA2,InnerA>& b);

20.6.6.1 scoped_allocator_adaptor constructors

scoped_allocator_adaptor();

Effects: Initializes the outer and inner allocator instances using their corresponding default constructors.

scoped_allocator_adaptor(scoped_allocator_adaptor&& other);
scoped_allocator_adaptor(const scoped_allocator_adaptor& other);

Effects: Initializes the outer and inner allocator instances from the corresponding parts of other.

scoped_allocator_adaptor(OuterA&& outer);
scoped_allocator_adaptor(const OuterA& outer);

Requires: scoped_allocator_adaptor was instantiated with only one parameter.

Effects: Initializes the base class (which is both the outer and inner allocator) from outer.

template<typename OuterA2>
    scoped_allocator_adaptor(    
        scoped_allocator_adaptor<OuterA2, InnerA>&& other);
template<typename OuterA2>
    scoped_allocator_adaptor(    
        const scoped_allocator_adaptor<OuterA2, InnerA>& other);

Requires: Same<OuterA2, OuterA::rebind<value_type>::other>.

Effects: Initializes the outer and inner allocator instances from the corresponding parts of other.

Draft
20.6.6.2 scoped_allocator_adaptor members

```cpp
pointer address(reference x) const;
const_pointer address(const_reference x) const;

Returns: outer_allocator().address(x)
```

```cpp
pointer allocate(size_type n);

Returns: outer_allocator().allocate(n)
```

```cpp
template <typename _HintP>
pointer allocate(size_type n, _HintP u);

Returns: outer_allocator().allocate(n, u)
```

```cpp
void deallocate(pointer p, size_type n);

Effects: outer_allocator().deallocate(p, n)
```

```cpp
size_type max_size() const;

Returns: outer_allocator().max_size()
```

```cpp
template <class... Args>
void construct(pointer p, Args&&... args);

Effects: outer_allocator().construct(p, args...)
```

```cpp
void destroy(pointer p)

Effects: outer_allocator().destroy(p)
```

```cpp
const outer_allocator_type& outer_allocator() const;

Returns: The outer allocator used to construct this object.
```

```cpp
const inner_allocator_type& inner_allocator() const;

Returns: The inner allocator used to construct this object. For the single-parameter instantiation, returns the same reference as outer_allocator().
```

20.6.6.3 scoped_allocator_adaptor globals

```cpp
template<typename OuterA1, typename OuterA2, typename InnerA>
bool operator==(const scoped_allocator_adaptor<OuterA1, InnerA>& a,
const scoped_allocator_adaptor<OuterA2, InnerA>& b);

Returns: a.outer_allocator() == b.outer_allocator()
&& a.inner_allocator() == b.inner_allocator()
```

```cpp
template<typename OuterA1, typename OuterA2, typename InnerA>
bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerA>& a,
```

Draft
20.6 Memory

const scoped_allocator_adaptor<OuterA2, InnerA>& b);  

Returns: !(a == b)

20.6.7 Raw 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*(void);
      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>;

Effects: Initializes the iterator to point to the same value to which x points.

raw_storage_iterator<OutputIterator,T>& operator*();

Returns: *this

raw_storage_iterator<OutputIterator,T>& operator=(const T& element);

Effects: Constructs a value from element at the location to which the iterator points.

Returns: A reference to the iterator.

raw_storage_iterator<OutputIterator,T>& operator++();

Effects: Pre-increment: advances the iterator and returns a reference to the updated iterator.

raw_storage_iterator<OutputIterator,T> operator++(int);

Effects: Post-increment: advances the iterator and returns the old value of the iterator.

20.6.8 Temporary buffers

template <class T>
  pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n);

Draft
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 \( \text{sizeof}(T) \)), 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`.

### 20.6.9 `construct_element` [construct.element]

```cpp
template <class Alloc, class T, class... Args>
void construct_element(Alloc& alloc, T& r, Args&&... args);
```

[Note: This function is called from within containers to construct elements during insertion operations and to move elements during reallocation operations. It automates the process of determining whether the scoped allocator model is in use and transmitting the inner allocator for scoped allocators. — end note]

Effects: The first of the following items that applies:

- if `is_scoped_allocator<Alloc>` is derived from `false_type` or `uses_allocator<T, A::inner_allocator_type>` is derived from `false_type`,
  ```cpp
  alloc.construct(alloc.address(r), args...)
  ```

- if `constructible_with_allocator_prefix<T, A::inner_allocator_type, Args...>` is derived from `true_type`,
  ```cpp
  alloc.construct(alloc.address(r), allocator_arg_t, alloc.inner_allocator(), args...)
  ```

- if `constructible_with_allocator_suffix<T, A::inner_allocator_type, Args...>` is derived from `true_type`,
  ```cpp
  alloc.construct(alloc.address(r), args..., alloc.inner_allocator())
  ```

- otherwise, the program is ill-formed.

### 20.6.10 Specialized algorithms [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.

### 20.6.10.1 `uninitialized_copy` [uninitialized.copy]

Draft
template <class InputIterator, class ForwardIterator>
ForwardIterator
uninitialized_copy(InputIterator first, InputIterator last,
ForwardIterator result);

Effects:
for (; first != last; ++result, ++first)
  new (static_cast<void*>(&*result))
    typename iterator_traits<ForwardIterator>::value_type(*first);

Returns: result

20.6.10.2 uninitialized_fill

template <class ForwardIterator, class T>
void uninitialized_fill(ForwardIterator first, ForwardIterator last,
const T& x);

Effects:
for (; first != last; ++first)
  new (static_cast<void*>(&*first))
    typename iterator_traits<ForwardIterator>::value_type(x);

20.6.10.3 uninitialized_fill_n

template <class ForwardIterator, class Size, class T>
void uninitialized_fill_n(ForwardIterator first, Size n, const T& x);

Effects:
for (; n--; ++first)
  new (static_cast<void*>(&*first))
    typename iterator_traits<ForwardIterator>::value_type(x);

20.6.11 Class template unique_ptr

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)).

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 CopyAssignabe, however it is MoveConstructible and MoveAssignabe. [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]
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);
}

20.6.11 Default deleters

20.6.11.1 default_delete

namespace std {

    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 Memory

20.6.11.1.2 default_delete<T[]>

namespace std {
    template <class T> struct default_delete<T[]>
    {
        void operator()(T*) const;
    }
}

void operator()(T* ptr) const;
operator() calls delete[] on ptr. A diagnostic is required if T is an incomplete type.

20.6.11.1.3 default_delete<T[N]>

namespace std {
    template <class T, size_t N> struct default_delete<T[N]>
    {
        void operator()(T*, size_t) const;
    }
}

void operator()(T* ptr, size_t) const;
operator() calls delete[] on ptr. A diagnostic is required if T is an incomplete type. The size_t parameter is ignored.

20.6.11.2 unique_ptr for single objects

namespace std {
    template <class T, class D = default_delete<T>> class unique_ptr {
        public:
            typedef T element_type;
            typedef D deleter_type;

            // 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=(unspecifiead-pointer-type);
    }
}
The default type for the template parameter \( D \) is default_delete. A client-supplied template argument \( D \) shall 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.

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 \texttt{MoveConstructible}, \texttt{MoveAssignble}, and \texttt{Swappable}, and need not be \texttt{CopyConstructible} (unless copied into the unique_ptr) nor \texttt{CopyAssignble}.

### 20.6.11.2.1 unique_ptr constructors

**unique_ptr();**

1. **Requires:** \( D \) shall be default constructible, and that construction must not throw an exception. \( D \) shall not be a reference type.
2. **Effects:** Constructs a unique_ptr which owns nothing.
3. **Postconditions:** \( \text{get()} == 0 \). \( \text{get_deleter()} \) returns a reference to a default constructed deleter \( D \).
4. **Throws:** nothing.

**unique_ptr(T* p);**

5. **Requires:** The expression \( D()(p) \) shall be well formed. The default constructor of \( D \) shall not throw an exception. \( D \) must not be a reference type.
6. **Effects:** Constructs a unique_ptr which owns \( p \).
7. **Postconditions:** \( \text{get()} == p \). \( \text{get_deleter()} \) returns a reference to a default constructed deleter \( D \).
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:

\[
\text{unique}_{-}\text{ptr}(T* \ p, \ \text{constant} A\& \ d); \\
\text{unique}_{-}\text{ptr}(T* \ p, \ A&& \ d);
\]

If \( D \) is an lvalue-reference type \( A\& \), then the signatures are:

\[
\text{unique}_{-}\text{ptr}(T* \ p, \ A\& \ d); \\
\text{unique}_{-}\text{ptr}(T* \ p, \ A&& \ d);
\]

If \( D \) is an lvalue-reference type \( \text{const} \ A\& \), then the signatures are:

\[
\text{unique}_{-}\text{ptr}(T* \ p, \ \text{constant} A\& \ d); \\
\text{unique}_{-}\text{ptr}(T* \ p, \ A&& \ d);
\]

Requires: The expression \( d(p) \) shall be well formed.

If \( D \) is not an lvalue-reference type then

- If \( d \) is an lvalue or \( \text{const} \) rvalue then the first constructor of this pair will be selected. \( D \) must be \text{CopyConstructible} (Table 34), and this \text{unique}_{-}\text{ptr} will hold a copy of \( d \). The copy constructor of \( D \) shall not throw an exception.

- Otherwise \( d \) is a non-\( \text{const} \) rvalue and the second constructor of this pair will be selected. \( D \) need only be \text{MoveConstructible} (Table 33), and this \text{unique}_{-}\text{ptr} will hold a value \text{move constructed} from \( d \). The move constructor of \( D \) shall not throw an exception.

Otherwise \( D \) is an lvalue-reference type. \( d \) shall be reference-compatible with one of the constructors. If \( d \) is an rvalue, it will bind to the second constructor of this pair. That constructor shall emit a diagnostic. \[ Note: The diagnostic could be implemented using a \text{static} \_\text{assert} which assures that \( D \) is not a reference type. \] Else \( d \) is an lvalue and will bind to the first constructor of this pair. The type which \( D \) references need not be \text{CopyConstructible} nor \text{MoveConstructible}. This \text{unique}_{-}\text{ptr} will hold a \( D \) which refers to the lvalue \( d \). \[ Note: \( D \) may not be an rvalue-reference type. \]

Postconditions: \( \text{get()} == p \). \( \text{get}\_\text{deleter()} \) returns a reference to the internally stored deleter. If \( D \) is a reference type then \( \text{get}\_\text{deleter()} \) returns a reference to the lvalue \( d \).

Throws: nothing.

[Example:

\[
\begin{align*}
\text{D} \ & d; \\
\text{unique}_{-}\text{ptr}<\text{int}, \ D&> \ & p1(\text{new} \ \text{int}, \ D()); & // \text{D must be MoveConstructible} \\
\text{unique}_{-}\text{ptr}<\text{int}, \ D&> \ & p2(\text{new} \ \text{int}, \ d); & // \text{D must be CopyConstructible} \\
\text{unique}_{-}\text{ptr}<\text{int}, \ D\&> \ & p3(\text{new} \ \text{int}, \ d); & // p3 \text{ holds a reference to} \ d \\
\text{unique}_{-}\text{ptr}<\text{int}, \ \text{const} \ D\&> \ & p4(\text{new} \ \text{int}, \ D()); & // \text{error: rvalue deleter object combined} \\
& & // \text{with reference deleter type}
\end{align*}
\]

Draft
551 General utilities library 20.6 Memory

— end example ]

unique_ptr(unique_ptr&& u);

17 

Requires: If the deleter is not a reference type, construction of the deleter D from an rvalue D shall not throw an exception.

18 

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 ]

19 

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.

20 

Throws: nothing.

template <class U, class E> unique_ptr(unique_ptr<U, E>&& u);

21 

Requires: If D is not a reference type, construction of the deleter D from an rvalue of type E shall be well formed and shall not throw an exception. If D is a reference type, then E shall be the same type as D (diagnostic required). U* shall be implicitly convertible to T*.

22 

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 ]

23 

postconditions get() == value u.get() had before the construction, modulo any required offset adjustments resulting from the cast from U* to T*. get_deleter() returns a reference to the internally stored deleter which was constructed from u.get_deleter().

24 

Throws: nothing.

20.6.11.2.2 unique_ptr destructor [unique.ptr.single.dtor]

~unique_ptr();

1 

Effects: If get() == 0 there are no effects. Otherwise get_deleter()(get()).

2 

Throws: nothing.

20.6.11.2.3 unique_ptr assignment [unique.ptr.single.asgn]

unique_ptr& operator=(unique_ptr&& u);

1 

Requires: Assignment of the deleter D from an rvalue D must shall not throw an exception.

2 

Effects: 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 deleters are move assigned. —end note]

Returns: *this.

Throws: nothing.

template <class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u);

Requires: Assignment of the deleter D from an rvalue D shall not throw an exception. U* shall be implicitly convertible to T*.

Effects: 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.

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.

20.6.11.2.4 unique_ptr observers

T& operator*() const;

Requires: get() != 0.

Returns: *get().

Throws: nothing.

T* operator->() const;

Requires: get() != 0.

Returns: get().

Throws: nothing.

T* get() const;
553 General utilities library

20.6 Memory

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]

20.6.11.2.5 unique_ptr 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 shall be Swappable and shall 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.

20.6.11.3 unique_ptr for array objects with a runtime length

namespace std {
    template <class T, class D> class unique_ptr<T[], D> {
        public:
            typedef T element_type;
            typedef D deleter_type;

            Draft
20.6 Memory

General utilities library

---

// constructors
unique_ptr();
explicit unique_ptr(T* p);
unique_ptr(T* p, implementation defined d);
unique_ptr(T* p, implementation defined d);
unique_ptr(uniquePtr&& u);

// destructor
~unique_ptr();

// assignment
unique_ptr& operator=(unique_ptr&& u);
unique_ptr& operator=(unspecifed-pointer-type);

// observers
T& operator[]( ) const;
T* get() const;
delete_type& get_deleter();
const delete_type& get_deleter() const;
operator unspecified-bool-type () const;

// modifiers
T* release();
void reset(T* p = 0);
void swap(unique_ptr&& u);

// disable copy from lvalue
unique_ptr(const unique_ptr&) = delete;
unique_ptr& operator=(const unique_ptr&) = delete;

---

1 A specialization for array types is provided with a slightly altered interface.
   — 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[].

2 Descriptions are provided below only for member functions that have behavior different from the primary template.

20.6.11.3.1 unique_ptr constructors

unique_ptr(T* p);
unique_ptr(T* p, implementation defined d);

Draft
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]

20.6.11.3.2 unique_ptr observers

T& operator[](size_t i) const;

1  Requires: i < the size of the array to which the stored pointer points.
2  Returns: get()[i].
3  Throws: nothing.

20.6.11.3.3 unique_ptr modifiers

void reset(T* p = 0);

1  Requires: Does not accept pointer types which are convertible to T* (diagnostic required). [Note: One imple-
2  mentation technique is to create a private templated overload. — end note]
3  Effects: If p == get() there are no effects. Otherwise get_deleter()(get()).
4  Postcondition: get() == p.
5  Throws: nothing.

20.6.11.4 unique_ptr for array objects with a compile time length

namespace std {
    template <class T, class D, size_t N> class unique_ptr<T[N], D> {
public:
    typedef T element_type;
    typedef D deleter_type;
    static const size_t size = N;

    // 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);

Draft
unique_ptr& operator=(unspecified-pointer-type);

// observers
T& operator[](size_t) 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);

// disable copy from lvalue
unique_ptr(const unique_ptr&) = delete;
unique_ptr& operator=(const unique_ptr&) = delete;
);
}

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.

— 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 \( \text{get_deleter()}(\text{get()}, N) \) instead of \( \text{get_deleter()}(\text{get}) \). Client-defined deleter may be able to make use of this extra information.

— The size of the array is available as a static const size_t named \text{size}

Descriptions are provided below only for member functions that have behavior different from the unique_ptr<T[]> specialization.

20.6.11.4.1 unique_ptr destructor [unique.ptr.compiletime.dtor]

~unique_ptr();
1 Effects: If \( \text{get()} == 0 \) there are no effects. Otherwise \( \text{get_deleter()}(\text{get()}, N) \).
2 Throws: nothing.

20.6.11.4.2 unique_ptr observers [unique.ptr.compiletime.observers]

T& operator[](size_t i) const;
1 If \( i \geq N \) the behavior is implementation defined.
2 Returns: \( \text{get()[i]} \).
3 Throws: nothing.
20.6.11.4.3 unique_ptr modifiers

```cpp
void reset(T *p = 0);
```

1. **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*]

2. **Effects:** If p == get() there are no effects. Otherwise get_deleter()(get(),N).

3. **Postcondition:** get() == p.

4. **Throws:** nothing.

20.6.11.5 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);
```

1. **Effects:** Calls x.swap(y).

```cpp
template <class T1, class D1, class T2, class D2>
bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

2. **Returns:** x.get() == y.get().

```cpp
template <class T1, class D1, class T2, class D2>
bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

3. **Returns:** x.get() != y.get().

```cpp
template <class T1, class D1, class T2, class D2>
bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

4. **Returns:** x.get() < y.get().

```cpp
template <class T1, class D1, class T2, class D2>
bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

5. **Returns:** x.get() <= y.get().

```cpp
template <class T1, class D1, class T2, class D2>
bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

6. **Returns:** x.get() > y.get().

```cpp
template <class T1, class D1, class T2, class D2>
bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

7. **Returns:** x.get() >= y.get.
20.6.12 Smart pointers

20.6.12.1 Class bad_weak_ptr

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.

20.6.12.2 Class template shared_ptr

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.

namespace std {
    template<class T> class shared_ptr {
    public:
        typedef T element_type;

        // 20.6.12.2.1, 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 A> shared_ptr(Y* p, A a);
        template<class Y> shared_ptr(const shared_ptr<Y>& r, T *p);
        shared_ptr(const shared_ptr& r);
        template<class Y> shared_ptr(const shared_ptr<Y>& r);
        shared_ptr(shared_ptr&& r);
        template<class Y> shared_ptr(shared_ptr<Y>&& r);
        template<class Y> explicit shared_ptr(const weak_ptr<Y>& r);
        template<class Y> explicit shared_ptr(auto_ptr<Y>&& r);
        template <class Y, class D> explicit shared_ptr(const unique_ptr<Y, D>&& r) = delete;
        template <class Y, class D> explicit shared_ptr(unique_ptr<Y, D>&& r);

        // 20.6.12.2.2, destructor:
        ~shared_ptr();

        // 20.6.12.2.3, assignment:
        shared_ptr& operator=(const shared_ptr& r);
        template<class Y> shared_ptr& operator=(const shared_ptr<Y>& r);
    } // template shared_ptr
} // namespace std

Draft
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);
    template <class Y, class D> shared_ptr& operator=(const unique_ptr<Y, D>&& r) = delete;
    template <class Y, class D> shared_ptr& operator=(unique_ptr<Y, D>&& r);

    // 20.6.12.2.4, 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);

    // 20.6.12.2.5, observers:
    T* get() const;
    T& operator*() const;
    T* operator->() const;
    long use_count() const;
    bool unique() const;
    operator unspecified-bool-type () const;
    
    // 20.6.12.2.6, shared_ptr creation
    template<class T, class... Args> shared_ptr<T> make_shared(Args&&... args);
    template<class T, class A, class... Args>
        shared_ptr<T> allocate_shared(const A& a, Args&&... args);

    // 20.6.12.2.7, shared_ptr comparisons:
    template<class T, class U>
        bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b);
    template<class T, class U>
        bool operator!=(const shared_ptr<T>& a, const shared_ptr<U>& b);
    template<class T, class U>
        bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b);

    // 20.6.12.2.8, shared_ptr I/O:
    template<class E, class T, class Y>
        basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, const shared_ptr<Y>& p);

    // 20.6.12.2.9, shared_ptr specialized algorithms:
    template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b);

    // 20.6.12.2.10, shared_ptr casts:
    template<class T, class U>
        shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r);
    template<class T, class U>
        shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r);
    template<class T, class U>
        shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r);

Draft
// 20.6.12.2.11, shared_ptr get_deleter:
  template<class D, class T> D* get_deleter(const shared_ptr<T>& p);
} // namespace std

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.

[ Example:
  if(shared_ptr<X> px = dynamic_pointer_cast<X>(py)) {
    // do something with px
  }
  — end example ]

20.6.12.2.1 shared_ptr constructors [util.smartptr.shared.const]

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.

```cpp
template<class Y> shared_ptr(const shared_ptr<Y>& r, T *p);
```

**Effects:** Constructs a \( \text{shared\_ptr} \) instance that stores \( p \) and \( \text{shares ownership} \) with \( r \).

**Postconditions:** \( \text{get() == p \&\& use\_count() == r.use\_count()} \)

**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 \text{empty} \( \text{shared\_ptr} \) instance with a non-NULL stored pointer. — end note]

```cpp
shared_ptr(const shared_ptr& r);
template<class Y> shared_ptr(const shared_ptr<Y>& r);
```

**Requires:** For the second constructor \( \text{Y*} \) shall be convertible to \( T* \).

**Effects:** If \( r \) is \text{empty}, constructs an \text{empty} \( \text{shared\_ptr} \) object; otherwise, constructs a \( \text{shared\_ptr} \) object that \( \text{shares ownership} \) with \( r \).

**Postconditions:** \( \text{get() == r.get() \&\& use\_count() == r.use\_count()} \).

**Throws:** nothing.

```cpp
shared_ptr(shared_ptr&& r);
template<class Y> shared_ptr(shared_ptr<Y>&& r);
```

**Requires:** For the second constructor \( \text{Y*} \) shall be convertible to \( T* \).

**Effects:** Move-constructs a \( \text{shared\_ptr} \) instance from \( r \).

**Postconditions:** \( \*\text{this} \) shall contain the old value of \( r \). \( r \) shall be \text{empty}.

**Throws:** nothing.

```cpp
template<class Y> explicit shared_ptr(const weak_ptr<Y>& r);
```

**Requires:** \( \text{Y*} \) shall be convertible to \( T* \).

**Effects:** Constructs a \( \text{shared\_ptr} \) object that \( \text{shares ownership} \) with \( r \) and stores a copy of the pointer stored in \( r \).

**Postconditions:** \( \text{use\_count()} == r.use\_count() \).

**Throws:** \text{bad\_weak\_ptr} when \( r\text{.expired()} \).

**Exception safety:** If an exception is thrown, the constructor has no effect.

```cpp
template<class Y> shared_ptr(auto_ptr<Y>&& r);
```

**Requires:** \( r\text{.release()} \) shall be convertible to \( T* \). \( \text{Y} \) shall be a complete type. The expression \( \text{delete } r\text{.release()} \) shall be well-formed, shall have well defined behavior, and shall not throw exceptions.
Effects: Constructs a shared_ptr object that stores and owns r.release().

Postconditions: use_count() == 1 && r.get() == 0.

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, the constructor has no effect.

```cpp
template <class Y, class D> shared_ptr(unique_ptr<Y, D>&& r);
```

Effects: Equivalent to shared_ptr(r.release(), r.get_deleter()) when D is not a reference type, otherwise shared_ptr(r.release(), ref(r.get_deleter())).

Exception safety: If an exception is thrown, the constructor has no effect.

20.6.12.2.2 shared_ptr destructor

```
~shared_ptr();
```

Effects:

- If *this is empty or shares ownership with another shared_ptr instance (use_count() > 1), there are no side effects.
- Otherwise, if *this owns a pointer p and a deleter d, d(p) is called.
- 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]

20.6.12.2.3 shared_ptr assignment

```
shared_ptr& operator=(const shared_ptr& r);
template<class Y> shared_ptr& operator=(const shared_ptr<Y>&& 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);

Effects: Equivalent to shared_ptr(std::move(r)).swap(*this).

Returns: *this.

template<class Y> shared_ptr& operator=(shared_ptr<Y>&& r);

Effects: Equivalent to shared_ptr(std::move(r)).swap(*this).

Returns: *this.

20.6.12.2.4 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).

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).

20.6.12.2.5 shared_ptr observers

T* get() const;

Returns: the stored pointer. Returns a null pointer if *this is empty.

Throws: nothing.

T& operator*() const;

Requires: get() != 0.

Returns: *get().

Throws: nothing.
6 Remarks: When T is void, it is unspecified whether this member function is declared. If it is declared, it is unspecified what its return type is, except that the declaration (although not necessarily the definition) of the function shall be well-formed.

T* operator->() const;
7 Requires: get() != 0.
8 Returns: get().
9 Throws: nothing.

long use_count() const;
10 Returns: the number of shared_ptr objects, *this included, that share ownership with *this, or 0 when *this is empty.
11 Throws: nothing.
12 [ Note: use_count() is not necessarily efficient. — end note ]

bool unique() const;
13 Returns: use_count() == 1.
14 Throws: nothing.
15 [ 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;
16 Returns: an unspecified value that, when used in boolean contexts, is equivalent to get() != 0.
17 Throws: nothing.
18 [ 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 ]

20.6.12.2.6 shared_ptr creation

template<class T, class... Args> shared_ptr<T> make_shared(Args&&... args);
1 template<class T, class A, class... Args>
2 shared_ptr<T> allocate_shared(const A& a, Args&&... args);

1 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.

2 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.
565 General utilities library 20.6 Memory

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]

20.6.12.2.7 `shared_ptr` comparison [util.smartptr.shared.cmp]

```cpp
template<class T, class U> bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b);
```

Returns: `a.get() == b.get()`.

Throws: nothing.

```cpp
template<class T, class U> bool operator!=(const shared_ptr<T>& a, const shared_ptr<U>& b);
```

Returns: `a.get() != b.get()`.

Throws: nothing.

```cpp
template<class T, class U> bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& 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 `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]

20.6.12.2.8 `shared_ptr` I/O [util.smartptr.shared.io]

```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`.

20.6.12.2.9 `shared_ptr` specialized algorithms [util.smartptr.shared.spec]

```cpp
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b);
```

Draft
Effects: Equivalent to `a.swap(b)`.

Throws: nothing.

**20.6.12.2.10** shared_ptr casts

```
template<class T, class U> shared_ptr<T> static_pointer_cast(const shared_ptr<U>& 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]

```
template<class T, class U> shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& 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.

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(const shared_ptr<U>& 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]

**20.6.12.2.11** get_deleter

```
template<class D, class T> D* get_deleter(const shared_ptr<T>& 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

Draft
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.

20.6.12.3 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);
            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.

20.6.12.3.1 weak_ptr constructors
weak_ptr();

1    Effects: Constructs an empty weak_ptr object.
2    Postconditions: use_count() == 0.
3    Throws: nothing.

weak_ptr(const weak_ptr& r);
template<class Y> weak_ptr(const weak_ptr<Y>& r);
template<class Y> weak_ptr(const shared_ptr<Y>& r);

4    Requires: For the first and third constructors, Y* shall be The second and third constructors shall not participate in the overload resolution unless Y* is implicitly convertible to T*.
5    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.
6    Postconditions: use_count() == r.use_count().
7    Throws: nothing.

20.6.12.3.2 weak_ptr destructor

~weak_ptr();

1    Effects: Destroys this weak_ptr object but has no effect on the object its stored pointer points to.
2    Throws: nothing.

20.6.12.3.3 weak_ptr assignment

weak_ptr& operator=(const weak_ptr& r);
template<class Y> weak_ptr& operator=(const weak_ptr<Y>& r);
template<class Y> weak_ptr& operator=(const shared_ptr<Y>& 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.

20.6.12.3.4 weak_ptr modifiers

void swap(weak_ptr& r);

1    Effects: Exchanges the contents of *this and r.
2    Throws: nothing.

void reset();
**Effects:** Equivalent to `weak_ptr().swap(*this)`.

### 20.6.12.3.5 weak_ptr observers [util.smartptr.weak.obs]

```cpp
long use_count() const;
```

*Returns:* 0 if `*this` is empty; otherwise, the number of `shared_ptr` instances that *share ownership* with `*this`.

*Throws:* nothing.

*Note:* `use_count()` is not necessarily efficient. — end note

```cpp
bool expired() const;
```

*Returns:* `use_count()` == 0.

*Throws:* nothing.

*Note:* `expired()` may be faster than `use_count()`. — end note

```cpp
shared_ptr<T> lock() const;
```

*Returns:* `expired()` ? `shared_ptr<T>()` : `shared_ptr<T>(*this)`.

*Throws:* nothing.

### 20.6.12.3.6 weak_ptr comparison [util.smartptr.weak.cmp]

```cpp
template<class T, class U> bool operator<(const weak_ptr<T>& a, const weak_ptr<U>& 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

### 20.6.12.3.7 weak_ptr specialized algorithms [util.smartptr.weak.spec]

```cpp
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b)
```

*Effects:* Equivalent to `a.swap(b)`.

*Throws:* nothing.
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`.

```
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
}
```

The template parameter `T` of `enable_shared_from_this` may be an incomplete type.

```
enable_shared_from_this();
enable_shared_from_this(const enable_shared_from_this<T>&);
```

**Effects:** Constructs an `enable_shared_from_this<T>` object.

**Throws:** nothing.

```
enable_shared_from_this<T>& operator=(const enable_shared_from_this<T>&);
```

**Returns:** `*this`.

**Throws:** nothing.

```
"enable_shared_from_this();
```

**Effects:** Destroys `*this`.

**Throws:** nothing.
shared_ptr<T> shared_from_this();
shared_ptr<T const> shared_from_this() const;

Requires: enable_shared_from_this<T> shall be an accessible base class of T. *this shall be a subobject of an object t of type T. There shall be at least one shared_ptr instance p that owns &t.

Returns: A shared_ptr<T> object r that shares ownership with p.

Postconditions: r.get() == this.

[Note: a possible implementation is shown below:

template<class T> class enable_shared_from_this {
private:
    weak_ptr<T> __weak_this;
protected:
    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]

20.6.13 Align [ptr.align]

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.

Draft
20.6.14 C Library

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</td>
</tr>
<tr>
<td></td>
<td>free realloc</td>
</tr>
</tbody>
</table>

2 The contents are the same as the Standard C library header <stdlib.h>, with the following changes:

3 The functions calloc(), malloc(), and realloc() do not attempt to allocate storage by calling ::operator new() (18.5).

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):

| Type: NULL |
| Type: size_t |
| Functions: memchr memcmp memcopy memmove memset |

6 The contents are the same as the Standard C library header <string.h>, with the change to memchr() specified in 21.5.

See also: ISO C clause 7.11.2.

20.7 Date and time

1 Header <ctime> (Table 54):

| Type: NULL |
| Type: size_t clock_t time_t |
| Struct: tm |
| Functions: asctime clock difftime localtime strftime ctime gmtime mktime strftime |

2 The contents are the same as the Standard C library header <time.h>.\(^{228}\)

\(^{228}\) 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.  

Draft
SEE ALSO: ISO C clause 7.12, Amendment 1 clause 4.6.4.
Chapter 21  Strings library

This clause describes components for manipulating sequences of any 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>
<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>21.5 Null-terminated sequence utilities</td>
<td>&lt;cstring&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cwctype&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

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

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></td>
<td>(described in 21.1.2)</td>
<td>compile-time</td>
</tr>
<tr>
<td>X::off_type</td>
<td></td>
<td>(described in 21.1.2)</td>
<td>compile-time</td>
</tr>
<tr>
<td>X::pos_type</td>
<td></td>
<td>(described in 21.1.2)</td>
<td>compile-time</td>
</tr>
<tr>
<td>X::state_type</td>
<td></td>
<td>(described in 21.1.2)</td>
<td>compile-time</td>
</tr>
<tr>
<td>X::eq(c,d)</td>
<td>bool</td>
<td>yields: whether c is to be treated as equal to d.</td>
<td>constant</td>
</tr>
<tr>
<td>X::lt(c,d)</td>
<td>bool</td>
<td>yields: whether c is to be treated as less than d.</td>
<td>constant</td>
</tr>
<tr>
<td>X::compare(p,q,n)</td>
<td>int</td>
<td>yields: 0 if for each i in [0,n), X::eq(p[i],q[i]) is true; else, a negative value if, for some j in [0,n), X::lt(p[j],q[j]) is true and for each i in [0,j) X::eq(p[i],q[i]) is true; else a positive value.</td>
<td>linear</td>
</tr>
<tr>
<td>X::length(p)</td>
<td>std::size_t</td>
<td>yields: the smallest i such that X::eq(p[i],charT()) is true.</td>
<td>linear</td>
</tr>
<tr>
<td>X::find(p,n,c)</td>
<td>const X::char_type*</td>
<td>yields: the smallest q in [p,p+n) such that X::eq(*q,c) is true, zero otherwise.</td>
<td>linear</td>
</tr>
<tr>
<td>X::move(s,p,n)</td>
<td>X::char_type*</td>
<td>for each i in [0,n), performs X::assign(s[i],p[i]). Copies correctly even where the ranges [p,p+n) and [s,s+n) overlap. yields: s.</td>
<td>linear</td>
</tr>
<tr>
<td>X::copy(s,p,n)</td>
<td>X::char_type*</td>
<td>pre: p not in [s,s+n). yields: s. for each i in [0,n), performs X::assign(s[i],p[i]).</td>
<td>linear</td>
</tr>
<tr>
<td>X::assign(r,d)</td>
<td>(not used)</td>
<td>assigns r=d.</td>
<td>constant</td>
</tr>
<tr>
<td>X::assign(s,n,c)</td>
<td>X::char_type*</td>
<td>for each i in [0,n), performs X::assign(s[i],c). yields: s.</td>
<td>linear</td>
</tr>
</tbody>
</table>
2 The struct template

    template<class charT> struct char_traits;

shall be provided in the header <string> as a basis for explicit specializations.

### 21.1.2 traits typedefs

typedef CHAR_T char_type;

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.

typedef INT_T int_type;

Requires: For a certain character container type char_type, a related container type INT_T shall be a type or class which can represent all of the valid characters converted from the corresponding char_type values, as well as an
end-of-file value, `eof()`. The type `int_type` represents a character container type which can hold end-of-file to be used as a return type of the iostream class member functions.

```cpp
typedef OFF_T off_type;
typedef POS_T pos_type;
```

Math: **Requires:** Requirements for `off_type` and `pos_type` are described in 27.1.2.

```cpp
typedef STATE_T state_type;
```

Math: **Requires:** `state_type` shall meet the requirements of `Assignable (23.1), CopyConstructible (34), and DefaultConstructible types.```

### 21.1.3 char_traits specializations

```cpp
namespace std {
    template<> struct char_traits<char> { 
        typedef char char_type;
        typedef int int_type;
        typedef streamoff off_type;
        typedef streampos pos_type;
        typedef mbstate_t state_type;

        static void assign(char_type& c1, const char_type& c2);
        static constexpr bool eq(char_type c1, char_type c2);
        static constexpr bool lt(char_type c1, 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);
    }
}
```

Math: 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>`.

Math: The requirements for the members of these specializations are given in clause 21.1.1.

#### 21.1.3.1 struct char_traits<char>

```cpp
namespace std {
    template<> struct char_traits<char> { 
        typedef char char_type;
        typedef int int_type;
        typedef streamoff off_type;
        typedef streampos pos_type;
        typedef mbstate_t state_type;

        static void assign(char_type& c1, const char_type& c2);
        static constexpr bool eq(char_type c1, char_type c2);
        static constexpr bool lt(char_type c1, 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);
    }
}
```

---

229 If `eof()` can be held in `char_type` then some iostreams operations may give surprising results.

Draft
static constexpr int_type not_eof(char_type c);
static constexpr char_type to_char_type(int_type c);
static constexpr int_type to_int_type(char_type c);
static constexpr bool eq_int_type(int_type c1, int_type c2);
static constexpr int_type eof();
}

The defined types for int_type, pos_type, off_type, and state_type shall be int, streampos, streamoff, and mbstate_t respectively.

2 The type streampos shall be an implementation-defined type that satisfies the requirements for POS_T in 21.1.2.

3 The type streamoff shall be an implementation-defined type that satisfies the requirements for OFF_T in 21.1.2.

4 The type mbstate_t is defined in <cwchar> and can represent any of the conversion states that can occur in an implementation-defined set of supported multibyte character encoding rules.

5 The two-argument member assign shall be defined identically to the built-in operator =. The two-argument members eq and lt shall be defined identically to the built-in operators == and < for type unsigned char.

6 The member eof() shall return EOF.

21.1.3.2 struct char_traits<char16_t> [char.traits.specializations.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 bool eq(char_type c1, char_type c2);
        static constexpr int_type to_int_type(char_type c);

        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(char_type c);
        static constexpr char_type to_char_type(int_type c);
        static constexpr int_type to_int_type(char_type c);
        static constexpr bool eq_int_type(int_type c1, int_type c2);
        static constexpr int_type eof();
    }
}
21.1 Character traits

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>`

```cpp
define 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(char_type c1, char_type c2);
        static constexpr bool lt(char_type c1, 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(char_type c);
        static constexpr char_type to_char_type(int_type c);
        static constexpr int_type to_int_type(char_type c);
        static constexpr bool eq_int_type(int_type c1, 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 Unicode code point.

### 21.1.3.4 `struct char_traits<wchar_t>`

```cpp
define std {
    template<> struct char_traits<wchar_t> {
        typedef wchar_t char_type;
        typedef wint_t int_type;
        typedef streamoff off_type;
        typedef wstreampos pos_type;
```

Draft
typedef mbstate_t state_type;

static void assign(char_type& c1, const char_type& c2);
static constexpr bool eq(char_type c1, char_type c2);
static constexpr bool lt(char_type c1, 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(char_type c);
static constexpr char_type to_char_type(int_type c);
static constexpr int_type to_int_type(char_type c);
static constexpr bool eq_int_type(int_type c1, 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

1. The header <string> defines the basic_string class template for manipulating varying-length sequences of char-like objects and four typedefs, string, u16string, u32string, and wstring, that name the specializations basic_string<char>, basic_string<char16_t>, basic_string<char32_t>, and basic_string<wchar_t>, respectively.

**Header <string> synopsis**

```cpp
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>;

```

Draft
// 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+(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,
        charT rhs);

template<class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>
    operator+(const basic_string<charT,traits,Allocator>& lhs,
        charT rhs);

template<class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>&&
    operator+(basic_string<charT,traits,Allocator>&& lhs,
        charT rhs);
template<class charT, class traits, class Allocator>
bool operator==(const basic_string<charT,traits,Allocator>& lhs,
               const basic_string<charT,traits,Allocator>& rhs);
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);
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);
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);
template<class charT, class traits, class Allocator>
bool operator>=(const charT* lhs,
              const basic_string<charT,traits,Allocator>& rhs);

Draft
21.2 String classes

Strings library

```cpp
// 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);

// 21.3.8.9: inserters and extractors
template<class charT, class traits, class Allocator>
basic_istream<charT, traits>&&
  getline(basic_istream<charT, traits>&& is,
          basic_string<charT, traits, Allocator>& str,
          charT delim);

// basic_string typedef names
typedef basic_string<char> string;
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
typedef basic_string<wchar_t> wstring;
```

// 21.4: numeric conversions
```plaintext
int stoi(const string& str, size_t *idx = 0, int base = 10);
long stol(const string str, size_t *idx = 0, int base = 10);
unsigned long stoul(const string str, size_t *idx = 0, int base = 10);
long long stoll(const string str, size_t *idx = 0, int base = 10);
unsigned long long stoull(const string str, size_t *idx = 0, int base = 10);
float stof(const string str, size_t *idx = 0);
```

Draft
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 Container, and for a Reversible Container, and for an Allocator-aware container. Thus, the iterators supported by `basic_string` are random access iterators.

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`;
- an `out-of-range` error is associated with exceptions of type `out_of_range`.

---

21.3 Class template `basic_string` [basic.string]

```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;

    } // basic_string
}
```

---

[Note: Allocator::value_type must name the same type as charT (21.3.1). — end note]
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 std::reverse_iterator<iterator> reverse_iterator;
typedef std::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(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(const basic_string&, const Allocator&);
basic_string(basic_string&&, const Allocator&);

~basic_string();
basic_string& operator=(const basic_string& str);
basic_string& operator=(basic_string&& str);
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;

Draft
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();

// 21.3.6 modifiers:
basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* str);
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(const charT* s);
basic_string& append(size_type n, 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,
21.3 Class template basic_string

Strings library

```cpp
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);
basic_string& insert(size_type pos, size_type n, charT c);
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);
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);
```

// 21.3.7 string operations:
const charT* c_str() const;       // explicit
const 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(charT c, 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 (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;
size_type find_last_not_of (const basic_string& str,
    size_type pos = npos) const;
size_type find_last_not_of (const charT* s, size_type pos,
    size_type n) const;
size_type find_last_not_of (const charT* s, size_type pos = npos) const;
size_type find_last_not_of (charT c, size_type pos = npos) const;

basic_string substr(size_type pos = 0, size_type n = npos) const;
int compare(const basic_string& str) const;
int compare(size_type pos1, size_type n1,
    const basic_string& str) const;
int compare(size_type pos1, size_type n1,
    const basic_string& str,
    size_type pos2, size_type n2) const;
int compare(const charT* s) const;
int compare(size_type pos1, size_type n1,
    const charT* s) const;
int compare(size_type pos1, size_type n1,
    const charT* s) const;

};

template <class charT, class traits, class Alloc
    struct constructible_with_allocator_suffix<
        basic_string<charT, traits, Alloc> > : true_type { };
21.3 Class template basic_string

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.

5. [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:

        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

explicit basic_string(const Allocator& a = Allocator());

Effects: Constructs an object of class basic_string. The postconditions of this function are indicated in Table 57.

basic_string(const basic_string<charT, traits, Allocator>& str);
basic_string(basic_string<charT, traits, Allocator>&& str);

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 57: basic_string(const Allocator&) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>a non-null pointer that is copyable and can have 0 added to it</td>
</tr>
<tr>
<td>size()</td>
<td>0</td>
</tr>
<tr>
<td>capacity()</td>
<td>an unspecified value</td>
</tr>
</tbody>
</table>

Throws: The second form throws nothing if the allocator's move constructor throws nothing.

Table 58: basic_string(const basic_string&) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>points at the first element of an allocated copy of the array whose first element is pointed at by str.data()</td>
</tr>
<tr>
<td>size()</td>
<td>str.size()</td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as size()</td>
</tr>
</tbody>
</table>

basic_string(const basic_string&lt;charT,traits,Allocator&gt;& str,  
  size_type pos, size_type n = npos,  
  const Allocator& a = Allocator());

Requires: \( pos \leq str.size() \)

Throws: out_of_range if \( pos > str.size() \).

Effects: Constructs an object of class basic_string and determines the effective length \( rlen \) of the initial string value as the smaller of \( n \) and \( str.size() - pos \), as indicated in Table 59.

Table 59: basic_string(const basic_string&amp, size_type, size_type, const Allocator&) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>points at the first element of an allocated copy of ( rlen ) consecutive elements of the string controlled by str beginning at position pos</td>
</tr>
<tr>
<td>size()</td>
<td>( rlen )</td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as size()</td>
</tr>
</tbody>
</table>

basic_string(const charT* s, size_type n,  
  const Allocator& a = Allocator());

Requires: \( s \) shall not be a null pointer and \( n < npos \).

Effects: Constructs an object of class basic_string and determines its initial string value from the array of charT of length \( n \) whose first element is designated by \( s \), as indicated in Table 60.
21.3 Class template basic_string

Strings library 592

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>

**Requires:** s shall not be a null pointer.

**Effects:** Constructs an object of class basic_string and determines its initial string value from the array of charT of length traits::length(s) whose first element is designated by s, as indicated in Table 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 62: basic_string(size_t, charT, const Allocator&) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>points at the first element of an allocated array of n elements, each storing the initial value c</td>
</tr>
<tr>
<td>size()</td>
<td>n</td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as size()</td>
</tr>
</tbody>
</table>

**template<class InputIterator>**

basic_string(InputIterator begin, InputIterator end, const Allocator& a = Allocator());

**Effects:** If InputIterator is an integral type, equivalent to

basic_string(static_cast<size_type>(begin), static_cast<value_type>(end), a)

Otherwise constructs a string from the values in the range [begin, end), as indicated in the Sequence Requirements table (see 23.1.1).

Draft
593  Strings library  21.3  Class template basic_string

basic_string(const basic_string&, const Allocator&);
basic_string(basic_string&, const Allocator&);

Effects: Constructs an object of class basic_string as indicated in Table 63. The stored allocator is constructed from alloc. In the second form, str is left in a valid state with an unspecified value.

Table 63: basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&, const Allocator&) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>points at the first element of an allocated copy of the array whose first element is pointed at by the original value of str.data()</td>
</tr>
<tr>
<td>size()</td>
<td>the original value of str.size()</td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as size()</td>
</tr>
<tr>
<td>get_allocator()</td>
<td>alloc</td>
</tr>
</tbody>
</table>

Throws: The second form throws nothing if alloc == str.get_allocator() unless the copy constructor for Allocator throws.

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. If *this and str are the same object, the member has no effect.

Returns: *this

Table 64: operator=(const basic_string<charT, traits, 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 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 65. The constructor 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.

Throws: Nothing.

Returns: *this

basic_string<charT,traits,Allocator>&
operator=(const charT* s);

Draft
21.3 Class template `basic_string` Strings library 594

Table 65: `operator=(const basic_string<charT, 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 <code>str.data()</code></td>
</tr>
<tr>
<td>size()</td>
<td>previous value of <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>

24 _Returns_: `*this = basic_string<charT, traits, Allocator>(s)`.

25 _Remarks_: Uses `traits::length()`.

`basic_string<charT, traits, Allocator>& operator=(charT c);`

26 _Returns_: `*this = basic_string<charT, traits, Allocator>(1, c)`.

21.3.3 `basic_string` iterator support

```
iterator begin();
const_iterator begin() const;

iterator end();
const_iterator end() const;
```

1 _Returns_: an iterator referring to the first character in the string.

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

2 _Returns_: an iterator which is the past-the-end value.

```
reverse_iterator rend();
const_reverse_iterator rend() const;
```

3 _Returns_: an iterator which is semantically equivalent to `reverse_iterator(end())`.

21.3.4 `basic_string` capacity

```
sizetype size() const;
```

1 _Returns_: a count of the number of char-like objects currently in the string.

```
sizetype length() const;
```

2 _Returns_: `size()`.

```
sizetype max_size() const;
```

Draft
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());

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);

reserve() uses Allocator::allocate() which may throw an appropriate exception.
21.3 Class template basic_string

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<
operator+=(const basic_string<
operator+==(const charT* s);

Returns: append(str).

Returns: *this += basic_string<charT,traits,Allocator>(s).

Remarks: Uses traits::length().

operator+=(charT c);

Returns: *this += basic_string<charT,traits,Allocator>(1,c).

21.3.6.2 basic_string::append

append(const basic_string<charT,traits>& str);

Returns: append(str, 0, npos).
basic_string<

// 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]
21.3 Class template basic_string

21.3.6.4 basic_string::insert

basic_string& insert(size_type pos1,
    const basic_string& str);

Returns: insert(pos1, str, 0, npos).
**21.3 Class template basic_string**

```cpp
basic_string<charT,traits,Allocator>&
insert(size_type pos, const charT* s, size_type n);
```

Requires: \(pos1 \leq size()\) and \(pos2 \leq str.size()\)

**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 \(size() \geq npos - rlen\).

Otherwise, the function replaces the string controlled by \(*\text{this}\) with a string of length \(size() + rlen\) whose first \(pos1\) elements are a copy of the initial elements of the original string controlled by \(*\text{this}\), 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 \(*\text{this}\).

Returns: \(*\text{this}\).

```cpp
basic_string<charT,traits,Allocator>&
insert(size_type pos, const charT* s);
```

Returns: insert(pos,basic_string<charT,traits,Allocator>(s)).

```cpp
basic_string<charT,traits,Allocator>&
insert(size_type pos, size_type n, charT c);
```

Returns: insert(pos,basic_string<charT,traits,Allocator>(n,c)).

```cpp
iterator insert(iterator p, charT c);
```

Requires: \(p\) is a valid iterator on \(*\text{this}\).

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.

```cpp
void insert(iterator p, size_type n, charT c);
```

Requires: \(p\) is a valid iterator on \(*\text{this}\).

Effects: inserts \(n\) copies of \(c\) before the character referred to by \(p\).

```cpp
template<class InputIterator>
void insert(iterator p, InputIterator first, InputIterator last);
```

Requires: \(p\) is a valid iterator on \(*\text{this}\). \([\text{first, last})\) 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);

  Requires: pos <= size()
  Throws: out_of_range if pos > size().
  Effects: Determines the effective length xlen of the string to be removed as the smaller of n and size() - pos.
  The function then replaces the string controlled by *this with a string of length size() - xlen whose first
  pos elements are a copy of the initial elements of the original string controlled by *this, and whose remaining
  elements are a copy of the elements of the original string controlled by *this beginning at position pos + xlen.

  Returns: *this.

iterator erase(iterator p);

  Effects: removes the character referred to by p.
  Returns: an iterator which points to the element immediately following p prior to the element being erased. If no
  such element exists, end() is returned.

iterator erase(iterator first, iterator last);

  Requires: first and last are valid iterators on *this, defining a range [first,last).
  Effects: removes the characters in the range [first,last).
  Returns: an iterator which points to the element pointed to by last prior to the other elements being erased. If no
  such element exists, end() is returned.

void pop_back();

  Requires: !empty()
  Effects: Equivalent to erase(size() - 1, 1).

21.3.6.6 basic_string::replace

basic_string<charT,traits,Allocator>&
replace(size_type pos1, size_type n1, const basic_string<charT,traits,Allocator>& str);

  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);

  Requires: pos1 <= size() && pos2 <= str.size().
  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).
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() - xlen >= max_size() - rlen, throws length_error. Otherwise, the function replaces the string controlled by *this with a string of length size() - xlen + 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 + xlen.

Returns: *this.

basic_string<
ccharT, traits, Allocator> k
replace(size_type pos, size_type n1, const charT* s, size_type n2);

Returns: replace(pos, n1, basic_string<
ccharT, traits, Allocator>(s, n2)).

basic_string<
ccharT, traits, Allocator> k
replace(size_type pos, size_type n1, const charT* s);

Returns: replace(pos, n1, basic_string<
ccharT, traits, Allocator>(s)).

Remarks: Uses traits::length().

basic_string<
ccharT, traits, Allocator> k
replace(size_type pos, size_type n1, size_type n2, charT c);

Returns: replace(pos, n1, basic_string<
ccharT, traits, Allocator>(n2, c)).

basic_string& replace(iterator i1, iterator i2, const basic_string& str);

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.

Returns: *this.

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);

Returns: replace(i1, i2, basic_string(s, n)).

Remarks: Length change: n - (i2 - i1).

basic_string& replace(iterator i1, iterator i2, const charT* s);

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);
18   Returns: replace(i1,i2,basic_string(n,c)).
19   Remarks: Length change: n - (i2 - i1).

template<class InputIterator>
basic_string& replace(iterator i1, iterator i2, InputIterator j1, InputIterator j2);
20   Returns: replace(i1,i2,basic_string(j1,j2)).
21   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;
1   Requires: pos <= size()
2   Throws: out_of_range if pos > size().
3   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.
4   Returns: rlen.

21.3.6.8 basic_string::swap

void swap(basic_string<charT,traits,Allocator>&& s);
1   Throws: Nothing.
2   Postcondition: *this contains the same sequence of characters that was in s, s contains the same sequence of characters that was in *this.
3   Complexity: constant time.
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().

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;

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.

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;

Returns: a copy of the Allocator object used to construct the string.

21.3.7.2 basic_string::find

size_type find(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 + 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 find(const charT* s, size_type pos, size_type n) const;

Returns: find(basic_string<charT,traits,Allocator>(s,n),pos).

size_type find(const charT* s, size_type pos = 0) const;

Returns: find(basic_string<charT,traits,Allocator>(s),pos).

Remarks: Uses traits::length().

size_type find(charT c, size_type pos = 0) const;

Returns: find(basic_string<charT,traits,Allocator>(1,c),pos).
21.3.7.3 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).

21.3.7.4 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().
size_type find_first_of(charT c, size_type pos = 0) const;

    Returns: find_first_of(basic_string<
charT,traits,Allocator>(1,c),pos).

21.3.7.5 basic_string::find_last_of

size_type
    find_last_of(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 < 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_last_of(const charT* s, size_type pos, size_type n) const;

    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;

    Returns: find_last_of(basic_string<charT,traits,Allocator>(s),pos).

    Remarks: Uses traits::length().

size_type find_last_of(charT c, size_type pos = npos) const;

    Returns: find_last_of(basic_string<charT,traits,Allocator>(1,c),pos).

21.3.7.6 basic_string::find_first_not_of

size_type
    find_first_not_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 no element I of the string controlled by str.

    Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.

    Remarks: Uses traits::eq().

size_type
    find_first_not_of(const charT* s, size_type pos, size_type n) const;

    Returns: find_first_not_of(basic_string<charT,traits,Allocator>(s,n),pos).
21.3 Class template basic_string

size_type find_first_not_of(const charT* s, size_type pos = 0) const;

Returns: find_first_not_of(basic_string<charT,traits,Allocator>(s),pos).
Remarks: Uses traits::length().

size_type find_first_not_of(charT c, size_type pos = 0) const;

Returns: find_first_not_of(basic_string<charT,traits,Allocator>(1,c),pos).

21.3.7.7 basic_string::find_last_not_of

size_type
find_last_not_of(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 < size();
— traits::eq(at(xpos), str.at(I)) for no element I of the string controlled by str.

Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.
Remarks: Uses traits::eq().

size_type find_last_not_of(const charT* s, size_type pos,
size_type n) const;

Returns: find_last_not_of(basic_string<charT,traits,Allocator>(s,n),pos).

size_type find_last_not_of(const charT* s, size_type pos = npos) const;

Returns: find_last_not_of(basic_string<charT,traits,Allocator>(s),pos).
Remarks: Uses traits::length().

size_type find_last_not_of(charT c, size_type pos = npos) const;

Returns: find_last_not_of(basic_string<charT,traits,Allocator>(1,c),pos).

21.3.7.8 basic_string::substr

basic_string<charT,traits,Allocator> substr(size_type pos = 0, size_type n = npos) const;

Requires: pos <= size()

Throws: out_of_range if pos > size().

Effects: Determines the effective length rlen of the string to copy as the smaller of n and size() - pos.

Returns: basic_string<charT,traits,Allocator>((data()+pos,rlen).
21.3.7.9 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 \( \text{size()} \) and \( \text{str.size()} \). The function then compares the two strings by calling \( \text{traits::compare(data(), str.data(), rlen)} \).

Returns: the nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table 66.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{size()} &lt; \text{str.size()} )</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>( \text{size()} == \text{str.size()} )</td>
<td>0</td>
</tr>
<tr>
<td>( \text{size()} &gt; \text{str.size()} )</td>
<td>&gt; 0</td>
</tr>
</tbody>
</table>

int compare(size_type pos1, size_type n1, const basic_string<charT,traits,Allocator>& str) const;

Returns: \( \text{basic_string}\langle\text{charT},\text{traits},\text{Allocator}\rangle(*\text{this},\text{pos1},\text{n1}).\text{compare}(\text{str}) \).

int compare(size_type pos1, size_type n1, size_type n2, const basic_string<charT,traits,Allocator>& str, size_type pos2, size_type n2) const;

Returns: \( \text{basic_string}\langle\text{charT},\text{traits},\text{Allocator}\rangle(*\text{this},\text{pos1},\text{n1}).\text{compare}(\text{basic_string}\langle\text{charT},\text{traits},\text{Allocator}\rangle(\text{str},\text{pos2},\text{n2})) \).

int compare(const charT *s) const;

Returns: \( \text{this->compare(basic_string}\langle\text{charT},\text{traits},\text{Allocator}\rangle(\text{s})) \).

int compare(size_type pos, size_type n1, const charT *s) const;

Returns: \( \text{basic_string}\langle\text{charT},\text{traits},\text{Allocator}\rangle(*\text{this},\text{pos},\text{n1}).\text{compare}(\text{basic_string}\langle\text{charT},\text{traits},\text{Allocator}\rangle(\text{s})) \).

int compare(size_type pos, size_type n1, const charT *s, size_type n2) const;

Returns: \( \text{basic_string}\langle\text{charT},\text{traits},\text{Allocator}\rangle(*\text{this},\text{pos},\text{n1}).\text{compare}(\text{basic_string}\langle\text{charT},\text{traits},\text{Allocator}\rangle(\text{s},\text{n2})) \).

Draft
21.3.8 basic_string non-member functions

21.3.8.1 operator+

```cpp
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>
operator+(const basic_string<charT, traits, Allocator>& lhs,
const basic_string<charT, traits, Allocator>& rhs);
```

1 **Returns:** basic_string<charT, traits, Allocator>(lhs).append(rhs)

```cpp
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>&&
operator+(basic_string<charT, traits, Allocator>&& lhs,
const basic_string<charT, traits, Allocator>& rhs);
```

2 **Returns:** lhs.append(rhs)

```cpp
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>&&
operator+(const basic_string<charT, traits, Allocator>& lhs,
basic_string<charT, traits, Allocator>&& rhs);
```

3 **Returns:** rhs.insert(0, lhs) [Note: Or equivalently rhs.insert(0, lhs) — end note]

```cpp
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>&
operator+(const charT* lhs,
const basic_string<charT, traits, Allocator>& rhs);
```

4 **Returns:** basic_string<charT, traits, Allocator>(lhs) + rhs.

5 **Remarks:** Uses traits::length().

```cpp
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>&
operator+(const charT* lhs,
const basic_string<charT, traits, Allocator>& rhs);
```

6 **Returns:** rhs.insert(0, lhs).

7 **Remarks:** Uses traits::length().

```cpp
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>&
operator+(charT lhs,
const basic_string<charT, traits, Allocator>& rhs);
```

Draft
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).

template<class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>&&
    operator+(const basic_string<charT,traits,Allocator>& lhs,
              const charT* rhs);

Returns: lhs + basic_string<charT,traits,Allocator>(rhs).

Remarks: Uses traits::length().

template<class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>&
    operator+(basic_string<charT,traits,Allocator>&& lhs,
              const charT* rhs);

Returns: lhs.append(rhs).

Remarks: Uses traits::length().

template<class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>&
    operator+(const basic_string<charT,traits,Allocator>& lhs,
              charT rhs);

Returns: lhs + basic_string<charT,traits,Allocator>(1,rhs).

template<class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>&
    operator+(basic_string<charT,traits,Allocator>&& lhs,
              charT rhs);

Returns: lhs.append(1, rhs).

21.3.8.2 operator==

template<class charT, class traits, class Allocator>
    bool operator==(const basic_string<charT,traits,Allocator>& lhs,
                    const basic_string<charT,traits,Allocator>& rhs);

Returns: lhs.compare(rhs) == 0.
21.3 Class template basic_string

Returns: basic_string<
2 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).
4 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 basic_string<charT,traits,Allocator>& rhs);
1 Returns: !(lhs == rhs).

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).
4 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);
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).
21.3.8.5 operator>

template<class charT, class traits, class Allocator>
bool operator> (const basic_string<charT,traits,Allocator>& lhs,  
    const basic_string<charT,traits,Allocator>& rhs);

\[ \text{Returns: } lhs.\text{compare}(rhs) > 0. \]

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);

\[ \text{Returns: } lhs.\text{compare}(rhs) <= 0. \]

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);

\[ \text{Returns: } lhs.\text{compare}(rhs) >= 0. \]
21.3 Class template basic_string

Returns: basic_string<charT, traits, Allocator>(lhs) >= rhs.

```cpp
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).

21.3.8.8 swap

```cpp
template<class charT, class traits, class Allocator>
void swap(basic_string<charT, traits, Allocator>& lhs,
          basic_string<charT, traits, Allocator>& rhs);
```

```cpp
template<class charT, class traits, class Allocator>
void swap(basic_string<charT, traits, Allocator>&& lhs,
          basic_string<charT, traits, Allocator>& rhs);
```

```cpp
template<class charT, class traits, class Allocator>
void swap(basic_string<charT, traits, Allocator>& lhs,
          basic_string<charT, traits, Allocator>&& rhs);
```

Effects: lhs.swap(rhs);

21.3.8.9 Inserters and extractors

```cpp
template<class charT, class traits, class Allocator>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>&& is,
         basic_string<charT, traits, Allocator>& str);
```

Effects: Behaves as a formatted input function (27.6.1.2.1). After constructing a sentry object, if the sentry converts to true, calls str.erase() and then extracts characters from is and appends them to str as if by calling str.append(1, c). If is.width() is greater than zero, the maximum number n of characters appended is is.width(); otherwise n is str.max_size(). Characters are extracted and appended until any of the following occurs:

- n characters are stored;
- end-of-file occurs on the input sequence;
- isspace(c, is.getloc()) is true for the next available input character c.

After the last character (if any) is extracted, is.width(0) is called and the sentry object k is destroyed.

If the function extracts no characters, it calls is.setstate(ios::failbit), which may throw ios_base::failure (27.4.4.3).

Returns: is

```cpp
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<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);

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 22.2.2.2.2, 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

template<class charT, class traits, class Allocator>
basic_istream<charT, traits>&
    getline(basic_istream<charT, traits>&& is,
        basic_string<charT, traits, Allocator>& str,
        charT delim);

Effects: Behaves as an unformatted input function (27.6.1.3), 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.

template<class charT, class traits, class Allocator>
basic_istream<charT, traits>&
    getline(basic_istream<charT, traits>&& is,
        basic_string<charT, traits, Allocator>& str)

Returns: getline(is,str,is.widen('
'))

21.4 Numeric Conversions

int stoi(const string& str, size_t *idx = 0, int base = 10);
long stol(const string& str, size_t *idx = 0, int base = 10);
unsigned long stoul(const string& str, size_t *idx = 0, int base = 10);
long long stoll(const string& str, size_t *idx = 0, int base = 10);
unsigned long long stoull(const string& str, size_t *idx = 0, int base = 10);

Effects: the first two functions call `strtol(str.c_str(), ptr, base)`, and the last three functions call `strtoul(str.c_str(), ptr, base)`, `strtoll(str.c_str(), ptr, base)`, and `strtoull(str.c_str(), ptr, base)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

Returns: the converted result.

Throws: invalid_argument if `strtol`, `strtoul`, `strtoll`, or `strtoull` reports that no conversion could be performed. Throws `out_of_range` if the converted value is outside the range of representable values for the return type.

float stof(const string& str, size_t *idx = 0);
double stod(const string& str, size_t *idx = 0);
long double stold(const string& str, size_t *idx = 0);

Effects: the first two functions call `strtod(str.c_str(), ptr)` and the third function calls `strtold(str.c_str(), ptr)`. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

Returns: the converted result.

Throws: invalid_argument if `strtod` or `strtold` reports that no conversion could be performed. Throws `out_of_range` if `strtod` or `strtold` sets `errno` to `ERANGE`.

string to_string(long long val);
string to_string(unsigned long long val);
string to_string(long double val);

Returns: each function returns a string object holding the character representation of the value of its argument that would be generated by calling `sprintf(buf, fmt, val)` with a format specifier of "%lld", "%llu", or "%Lf", respectively.

Throws: nothing

int stoi(const wstring& str, size_t *idx = 0, int base = 10);
long stol(const wstring& str, size_t *idx = 0, int base = 10);
unsigned long stoul(const wstring& str, size_t *idx = 0, int base = 10);
long long stoll(const wstring& str, size_t *idx = 0, int base = 10);
unsigned long long stoull(const wstring& str, size_t *idx = 0, int base = 10);

Effects: the first two functions call `wcstol(str.c_str(), ptr, base)`, and the last three functions call `wcstoul(str.c_str(), ptr, base)`, `wcstoll(str.c_str(), ptr, base)`, and `wcstoull(str.c_str(), ptr, base)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does
not throw an exception and \( idx \neq 0 \), the function stores in \*idx the index of the first unconverted element of 

**Returns:** the converted result.

**Throws:** invalid_argument if wcstol, wcstoul, wcstoll, or wcstoull reports that no conversion could be performed. Throws out_of_range if the converted value is outside the range of representable values for the return type.

```c
float stof(const wstring& str, size_t* idx = 0);
double stod(const wstring& str, size_t* idx = 0);
long double stold(const wstring& str, size_t* idx = 0);
```

**Effects:** the first two functions call wcstod(str.c_str(), ptr) and the third function calls wcstold(str.c_str(), ptr). Each function returns the converted result, if any. The argument ptr designates a pointer to an object internal to the function that is used to determine what to store at \*idx. If the function does not throw an exception and \( idx \neq 0 \), the function stores in \*idx the index of the first unconverted element of str.

**Returns:** the converted result.

**Throws:** invalid_argument if wcstod or wcstold reports that no conversion could be performed. Throws out_of_range if wcstod or wcstold sets errno to ERANGE.

```c
string to_wstring(long long val);
string to_wstring(unsigned long long val);
string to_wstring(long double val);
```

**Returns:** each function returns a wstring object holding the character representation of the value of its argument that would be generated by calling wprintf(buf, fmt, val) with a format specifier of L"%lld", L"%llu", or L"%Lf", respectively.

**Throws:** nothing

### 21.5 Null-terminated sequence utilities

Tables 67, 68, 69, 70, 71. and 72 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 `<ctype.h>`, `<wctype.h>`, `<string.h>`, `<wchar.h>`, and `<stdlib.h>` and the C Unicode TR header `<uchar.h>`, respectively, with the following modifications:

1. The headers shall not define the types `char16_t`, `char32_t`, and `wchar_t` (2.11).
2. The function signature `strchr(const char*, int)` shall be replaced by the two declarations:
   ```c
   const char* strchr(const char* s, int c);
   char* strchr(char* s, int c);
   ```
   both of which shall have the same behavior as the original declaration.
3. The function signature `strpbrk(const char*, const char*)` shall be replaced by the two declarations:
   ```c
   const char* strpbrk(const char* s1, const char* s2);
   char* strpbrk(char* s1, const char* s2);
   ```
Draft
both of which shall have the same behavior as the original declaration.

6 The function signature `strrchr(const char*, int)` shall be replaced by the two declarations:

```c
const char* strrchr(const char* s, int c);
char* strrchr(char* s, int c);
```

both of which shall have the same behavior as the original declaration.

7 The function signature `strstr(const char*, const char*)` shall be replaced by the two declarations:

```c
const char* strstr(const char* s1, const char* s2);
char* strstr(char* s1, const char* s2);
```

both of which shall have the same behavior as the original declaration.

8 The function signature `memchr(const void*, int, size_t)` shall be replaced by the two declarations:

```c
const void* memchr(const void* s, int c, size_t n);
void* memchr(void* s, int c, size_t n);
```

both of which shall have the same behavior as the original declaration.

9 The function signature `wcschr(const wchar_t*, wchar_t)` shall be replaced by the two declarations:

```c
const wchar_t* wcschr(const wchar_t* s, wchar_t c);
wchar_t* wcschr(wchar_t* s, wchar_t c);
```

both of which shall have the same behavior as the original declaration.

10 The function signature `wcspbrk(const wchar_t*, const wchar_t*)` shall be replaced by the two declarations:

```c
const wchar_t* wcspbrk(const wchar_t* s1, const wchar_t* s2);
wchar_t* wcspbrk(wchar_t* s1, const wchar_t* s2);
```

both of which shall have the same behavior as the original declaration.

11 The function signature `wcsrchr(const wchar_t*, wchar_t)` shall be replaced by the two declarations:

```c
const wchar_t* wcsrchr(const wchar_t* s, wchar_t c);
wchar_t* wcsrchr(wchar_t* s, wchar_t c);
```

both of which shall have the same behavior as the original declaration.

12 The function signature `wcsstr(const wchar_t*, const wchar_t*)` shall be replaced by the two declarations:

```c
const wchar_t* wcsstr(const wchar_t* s1, const wchar_t* s2);
wchar_t* wcsstr(wchar_t* s1, const wchar_t* s2);
```

both of which shall have the same behavior as the original declaration.

13 The function signature `wmemchr(const wchar_t*, int, size_t)` shall be replaced by the two declarations:

```c
const wchar_t* wmemchr(const wchar_t* s, wchar_t c, size_t n);
wchar_t* wmemchr(wchar_t* s, wchar_t c, size_t n);
```
both of which shall have the same behavior as the original declaration.

**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 67: Header `<cctype>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functions:</strong></td>
<td></td>
</tr>
<tr>
<td>isalnum</td>
<td>isblank</td>
</tr>
<tr>
<td>isdigit</td>
<td>isprint</td>
</tr>
<tr>
<td>isupper</td>
<td></td>
</tr>
<tr>
<td>tolower</td>
<td>isalpha</td>
</tr>
<tr>
<td>isgraph</td>
<td>ispunct</td>
</tr>
<tr>
<td>isxdigit</td>
<td></td>
</tr>
<tr>
<td>toupper</td>
<td>iscntrl</td>
</tr>
<tr>
<td>islower</td>
<td>isspace</td>
</tr>
</tbody>
</table>

### Table 68: Header `<cwctype>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Macro:</strong></td>
<td>WEOF <code>&lt;cwctype&gt;</code></td>
</tr>
<tr>
<td><strong>Types:</strong></td>
<td>wctrans_t wc_type_t wint_t <code>&lt;cwctype&gt;</code></td>
</tr>
<tr>
<td><strong>Functions:</strong></td>
<td></td>
</tr>
<tr>
<td>iswalnum</td>
<td>iswcntrl</td>
</tr>
<tr>
<td>iswgraph</td>
<td>iswpunct</td>
</tr>
<tr>
<td>iswxdigit</td>
<td>towupper</td>
</tr>
<tr>
<td>iswalpha</td>
<td>iswctype</td>
</tr>
<tr>
<td>iswlower</td>
<td>iswspace</td>
</tr>
<tr>
<td>towctrans</td>
<td>wctrans</td>
</tr>
<tr>
<td>iswblank</td>
<td>iswdigit</td>
</tr>
<tr>
<td>iswprint</td>
<td>iswupper</td>
</tr>
<tr>
<td>towlower</td>
<td>wctype</td>
</tr>
</tbody>
</table>

### Table 69: Header `<cstring>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Macro:</strong></td>
<td>NULL <code>&lt;cstring&gt;</code></td>
</tr>
<tr>
<td><strong>Type:</strong></td>
<td>size_t <code>&lt;cstring&gt;</code></td>
</tr>
<tr>
<td><strong>Functions:</strong></td>
<td></td>
</tr>
<tr>
<td>memchr</td>
<td>strcat</td>
</tr>
<tr>
<td>strcspn</td>
<td>strncmp</td>
</tr>
<tr>
<td>memcpy</td>
<td>strcmp</td>
</tr>
<tr>
<td>memcmp</td>
<td>strlen</td>
</tr>
<tr>
<td>strcmp</td>
<td>strstr</td>
</tr>
<tr>
<td>memmove</td>
<td>strcoll</td>
</tr>
<tr>
<td>strncat</td>
<td>strspn</td>
</tr>
<tr>
<td>memset</td>
<td>strcpy</td>
</tr>
<tr>
<td>strcpy</td>
<td>strncmp</td>
</tr>
<tr>
<td>strchr</td>
<td>strxfrm</td>
</tr>
<tr>
<td>strerror</td>
<td>strrrchr</td>
</tr>
<tr>
<td>strpbrk</td>
<td></td>
</tr>
</tbody>
</table>
### Table 70: Header `<cwchar>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td>NULL <code>&lt;cwchar&gt;</code> WCHAR_MAX WCHAR_MIN WEOF <code>&lt;cwchar&gt;</code></td>
</tr>
<tr>
<td>Types:</td>
<td>mbstate_t wint_t <code>&lt;cwchar&gt;</code> size_t tm</td>
</tr>
<tr>
<td>Functions:</td>
<td>btowc mbrlen vfwscanf wcsncpy wcspn wcsxfrm</td>
</tr>
<tr>
<td></td>
<td>fgetwc mbrtowc vswscanf wcs cpn wcsstr wctob</td>
</tr>
<tr>
<td></td>
<td>fgetws mbsinit vsprintf wcsftime wcstod wmemchr</td>
</tr>
<tr>
<td></td>
<td>fputwc mbstorwc vfprintf wcslen wcstof wmemcmp</td>
</tr>
<tr>
<td></td>
<td>fputws putwc vsprint wcsncat wcstok wmemcpy</td>
</tr>
<tr>
<td></td>
<td>fwide putwchar wcrtom wcsncmp wcstol wmemmove</td>
</tr>
<tr>
<td></td>
<td>fwprintf suprintf wcsprintf wcscpy wcsstol wmemset</td>
</tr>
<tr>
<td></td>
<td>fwscanf swscanf wcschr wcspbrk wcstoll wprintf</td>
</tr>
<tr>
<td></td>
<td>getwc ungetwc wcsncpy wcschr wcsstoul wmemmove</td>
</tr>
<tr>
<td></td>
<td>getwchar vfwprintf wscoll wcsrtombs wcstoull</td>
</tr>
</tbody>
</table>

### Table 71: Header `<cstdlib>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td>MB_CUR_MAX</td>
</tr>
<tr>
<td>Functions:</td>
<td>atof mblen strtof strtoul</td>
</tr>
<tr>
<td></td>
<td>atoi mbtowc strtol strtoull</td>
</tr>
<tr>
<td></td>
<td>atol mbstowcs strtold wctomb</td>
</tr>
<tr>
<td></td>
<td>atoll strtod strtoull wcstombs</td>
</tr>
</tbody>
</table>

### Table 72: Header `<cuchar>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td><strong>STDC_UTF_16</strong> <strong>STDC_UTF_32</strong></td>
</tr>
<tr>
<td>Functions:</td>
<td>mbtoci6 c16rtomb</td>
</tr>
<tr>
<td></td>
<td>mbtoci32 c32rtomb</td>
</tr>
</tbody>
</table>

Draft
Chapter 22  Localization library  [localization]

1  This clause describes components that C++ programs may use to encapsulate (and therefore be more portable when confronting) cultural differences. The locale facility includes internationalization support for character classification and string collation, numeric, monetary, and date/time formatting and parsing, and message retrieval.

2  The following subclauses describe components for locales themselves, the standard facets, and facilities from the ISO C library, as summarized in Table 73

Table 73: 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 Standard code conversion facets</td>
<td>&lt;codecvt&gt;</td>
</tr>
<tr>
<td>22.4 C library locales</td>
<td>&lt;cllocale&gt;</td>
</tr>
</tbody>
</table>

22.1  Locales  [locales]

1  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> 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<Elemshr{} 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.\footnote{In this subclause, the type name \texttt{struct \_\_tm} is an incomplete type that is defined in \texttt{<ctime>}.}

22.1.1 Class locale

```cpp
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();
    };
}
```

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:]  

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> & s, 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

typedef int category;

233) 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

\[(\text{collate | ctype | monetary | numeric | time | messages | all}) == \text{all}\]

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 74:

<table>
<thead>
<tr>
<th>Category</th>
<th>Includes Facets</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>collate</code></td>
<td><code>collate&lt;char&gt;</code>, <code>collate&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td><code>ctype</code></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><code>monetary</code></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_get&lt;char, true&gt;</code>, <code>money_get&lt;wchar_t, true&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><code>numeric</code></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><code>time</code></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><code>messages</code></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 74, `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 75:

The provided implementation of members of facets `num_get<charT>` and `num_put<charT>` calls `use_facet<F>(l)` (1) only for facet `F` of types `numpunct<charT>` and `ctype<charT>`, and for locale `l` the value obtained by calling member `getloc()` on the `ios_base&` argument to these functions.

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>collatebyname&lt;char&gt;, collatebyname&lt;wchar_t&gt;</td>
</tr>
<tr>
<td>ctype</td>
<td>ctypename&lt;char&gt;, ctypename&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvtbyname&lt;char, char, mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvtbyname&lt;char16_t, char, mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvtbyname&lt;char32_t, char, mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvtbyname&lt;wchar_t, char, mbstate_t&gt;</td>
</tr>
<tr>
<td>monetary</td>
<td>moneypunctbyname&lt;char, International&gt;</td>
</tr>
<tr>
<td></td>
<td>moneypunctbyname&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>numpunctbyname&lt;char&gt;, numpunctbyname&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>num_get&lt;C, InputIterator&gt;, num_put&lt;C, OutputIterator&gt;</td>
</tr>
<tr>
<td>time</td>
<td>time_get&lt;char, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_getbyname&lt;char, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_get&lt;wchar_t, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_getbyname&lt;wchar_t, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put&lt;char, OutputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_putbyname&lt;char, OutputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put&lt;wchar_t, OutputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_putbyname&lt;wchar_t, OutputIterator&gt;</td>
</tr>
<tr>
<td>messages</td>
<td>messagesbyname&lt;char&gt;, messagesbyname&lt;wchar_t&gt;</td>
</tr>
</tbody>
</table>
parameter that satisfies the requirements for a character on which any of the iostream components can be instantiated. 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();
        facet(const facet&) = delete;
        void operator=(const facet&) = delete;
    };
}
```

1 Template parameters in this clause which are required to be facets are those named Facet in declarations. A program that passes a type that is not a facet, or a type that refers to a volatile-qualified facet, as an (explicit or deduced) template parameter to a locale function expecting a facet, is ill-formed. A const-qualified facet is a valid template argument to any locale function that expects a Facet template parameter.

2 The refs argument to the constructor is used for lifetime management.
   — 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.

22.1.1.1.3 Class locale::id

```cpp
namespace std {
    class locale::id {
    public:
        id();
        void operator=(const id&) = delete;
        id(const id&) = delete;
    };
}
```
The class locale::id provides identification of a locale facet interface, used as an index for lookup and to encapsulate initialization.

[Note: Because facets are used by iostreams, potentially while static constructors are running, their initialization cannot depend on programmed static initialization. One initialization strategy is for locale to initialize each facet’s id member the first time an instance of the facet is installed into a locale. This depends only on static storage being zero before constructors run (3.6.2). — end note]

### 22.1.1.2 locale constructors and destructor

locale() throw();

**Effects:** Default constructor: a snapshot of the current global locale.

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).

template <class Facet> locale(const locale& other, Facet* f);

Draft
14  *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`.

15  *Remarks:* The resulting locale has no name.

\[
\text{locale}(\text{const locale}& \text{ other, const locale}& \text{ one, category cats});
\]

16  *Effects:* Constructs a locale incorporating all facets from the first argument except those that implement `cats`, which are instead incorporated from the second argument.

17  *Remarks:* The resulting locale has a name if and only if the first two arguments have names.

18  \(~\text{locale}()\) \text{ throw}();

A non-virtual destructor that throws no exceptions.

### 22.1.1.3 locale members

\[
\text{template <class Facet> locale combine(const locale}& \text{ other) const;}
\]

1  *Effects:* Constructs a locale incorporating all facets from `*this` except for that one facet of `other` that is identified by `Facet`.

2  *Returns:* The newly created locale.

3  *Throws:* runtime\_error if `has\_facet<Facet>(other)` is false.

4  *Remarks:* The resulting locale has no name.

\[
\text{basic\_string<char> name() const;}
\]

5  *Returns:* The name of `*this`, if it has one; otherwise, the string "]". If `*this` has a name, then `locale(name().\text{c\_str}())` is equivalent to `*this`. Details of the contents of the resulting string are otherwise implementation-defined.

### 22.1.4 locale operators

\[
\text{bool operator==(const locale}& \text{ other) const;}
\]

1  *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.

\[
\text{bool operator!=(const locale}& \text{ other) const;}
\]

2  *Returns:* The result of the expression: !(\text{*this == other}).

\[
\text{template <class charT, class Traits, class Allocator> bool operator()(const basic\_string<charT,Traits,Allocator>& s1, const basic\_string<charT,Traits,Allocator>& s2) const;}
\]

3  *Effects:* Compares two strings according to the `collate<charT>` facet.
22.1 Locales

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:

\[
\text{use\_facet< collate<}\text{charT} > (*this).\text{compare} \\
\quad (s1.\text{data}(), s1.\text{data}()+s1.\text{size}(), s2.\text{data}(), s2.\text{data}()+s2.\text{size}()) < 0;
\]

Example: A vector of strings \(v\) can be collated according to collation rules in locale \(\text{loc}\) simply by (25.3.1, 23.2.6):

\[
\text{std::sort}(v.\text{begin}(), v.\text{end}(), \text{loc});
\]

— 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

\[
\text{std::setlocale}(\text{LC\_ALL}, \text{loc.name}().\text{c\_str}());
\]

otherwise, the effect on the C locale, if any, is implementation-defined. No library function other than locale::

\[
\text{global}() \text{shall affect the value returned by locale().}
\]

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

\[
\text{template <class Facet> const Facet& use\_facet(const locale& loc);} \\
\]

Requires: Facet is a facet class whose definition contains the public static member \(\text{id}\) as defined in 22.1.1.2.

Returns: a reference to the corresponding facet of \(\text{loc}\), if present.

Throws: bad_cast if has_facet<Facet>(loc) is false.

Remarks: The reference returned remains valid at least as long as any copy of \(\text{loc}\) exists.

\[
\text{template <class Facet> bool has\_facet(const locale& loc) throw();} \\
\]

Returns: true if the facet requested is present in \(\text{loc}\); otherwise false.

Draft
22.1.3 Convenience interfaces

22.1.3.1 Character classification

```cpp
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:

```cpp
    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).\(^{234}\)

22.1.3.2 Conversions

22.1.3.2.1 Character conversions

```cpp
template <class charT> charT toupper(charT c, const locale& loc);
template <class charT> charT tolower(charT c, const locale& loc);
```

1 Returns: `use_facet<ctype<charT> >(loc).toupper(c)`.

2 Returns: `use_facet<ctype<charT> >(loc).tolower(c)`.

22.1.3.2.2 String conversions

1 Class template `wstring_convert` performs conversions between a wide string and a byte string. It lets you specify a code conversion facet (like class template `codecvt`) to perform the conversions, without affecting any streams or locales. [Example: 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:

```cpp
    wstring_convert<codecvt_utf8<wchar_t>> myconv;
    std::string mbstring = myconv.to_bytes(L"Hello\n");
    std::cout << mbstring;
```

— end example ]

2 Class template `wstring_convert synopsis`
namespace std {
    template<class Codecvt, class Elem = wchar_t> class wstring_convert {
        public:
            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
    }
}

3 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 object 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)

Draft
Localization library

— cvtstate — a conversion state object
— cvtcount — a conversion count

typedef std::basic_string<char> byte_string;

size_t converted() const;

Returns: cvtcount.

wide_string from_bytes(char byte);
wide_string from_bytes(const char *ptr);
wide_string from_bytes(const byte_string& str);
wide_string from_bytes(const char *first, const char *last);

Effects: The first member function shall convert the single-element sequence byte to a wide string. The second member function shall convert the nul-terminated sequence beginning at ptr to a wide string. The third member function shall convert the sequence stored in str to a wide string. The fourth member function shall convert the sequence defined by the range [first, last) to a wide string.

In all cases:

— 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 wide string. Otherwise, if the object was constructed with a wide-error string, the member function shall return the wide-error string. Otherwise, the member function throws an object of class std::range_error.

typedef typename wide_string::traits_type::int_type int_type;

The type shall be a synonym for wide_string::traits_type::int_type.

state_type state() const;

Returns cvtstate.

typedef typename Codecvt::state_type state_type;

The type shall be a synonym for Codecvt::state_type.

byte_string to_bytes(Elem wchar);
byte_string to_bytes(const Elem *wptr);
byte_string to_bytes(const wide_string& wstr);
byte_string to_bytes(const Elem *first, const Elem *last);

Effects: The first member function shall convert the single-element sequence wchar to a byte string. The second member function shall convert the 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.

typedef std::basic_string<Elem> wide_string;

The type shall be a synonym for std::basic_string<Elem>.

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());

ffects: 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.

\tilde{}wstring_convert();

ffects: The destructor shall delete cvtptr.

22.1.3.2.3 Buffer conversions

Class template wbuffer_convert looks like a wide stream buffer, but performs all its I/O through an underlying byte stream buffer that you specify when you construct it. Like class template wstring_convert, it lets you specify a code conversion facet to perform the conversions, without affecting any streams or locales.

Class template wbuffer_convert synopsis

namespace std {
  template<class Codecvt,
    class Elem = wchar_t,
    class Tr = std::char_traits<Elem> >
  class wbuffer_convert
    : public std::basic_streambuf<Elem, Tr> {
    public:
      typedef typename Tr::state_type state_type;

      wbuffer_convert(std::streambuf *bytebuf = 0,
                     Codecvt *pcvt = new Codecvt,
                     state_type state = state_type());

      std::streambuf *rdbuf() const;
  } // class wbuffer_convert
} // namespace std
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
- `cvtptr` — a pointer to the allocated conversion object (which is freed when the `wbuffer_convert` object is destroyed)
- `cvtstate` — a conversion state object

Returns: `cvtstate`.

Effects: stores `bytebuf` in `bufptr`.

Returns: the previous value of `bufptr`.

typedef typename Codecvt::state_type state_type;

The type shall be a synonym for `Codecvt::state_type`.

Effects: The constructor constructs a stream buffer object, initializes `bufptr` to `bytebuf`, initializes `cvtptr` to `pcvt`, and initializes `cvtstate` to `state`.

Effects: The destructor shall delete `cvtptr`. 
22.2 Standard locale categories

Each of the standard categories includes a family of facets. Some of these implement formatting or parsing of a datum, for use by standard or users’ iostream operators << and >>, as members put() and get(), respectively. Each such member function takes an ios_base& argument whose members flags(), precision(), and width(), specify the format of the corresponding datum. Those functions which need to use other facets call its member getloc() to retrieve the locale imbued there. Formatting facets use the character argument fill to fill out the specified width where necessary.

The put() members make no provision for error reporting. (Any failures of the OutputIterator argument must be extracted from the returned iterator.) The get() members take an ios_base::iostate& argument whose value they ignore, but set to ios_base::failbit in case of a parse error.

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

namespace std {
    template <class charT>
    class ctype : public locale::facet, public ctype_base {
public:
        typedef charT char_type;

        explicit ctype(size_t refs = 0);

        bool is(mask m, charT c) const;
        const charT* is(const charT* low, const charT* high, mask* vec) const;
        const charT* scan_is(mask m,
                                const charT* low, const charT* high) const;
    }
const charT* scan_not(mask m, const charT* low, const charT* high) const;
charT 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;
};

1 Class ctype encapsulates the C library <cctype> features. istream members are required to use ctype<> for character classing during input parsing.

2 The specializations required in Table 74 (22.1.1.1.1), namely ctype<char> and ctype<wchar_t>, implement character classing appropriate to the implementation’s native character set.

22.2.1.1 ctype members

bool is(mask m, charT c) const;
const charT* is(const charT* low, const charT* high, mask* vec) const;

Returns: do_is(m, c) or do_is(low, high, vec)
22.2 Standard locale categories

```
const charT* scan_is(mask m,
    const charT* low, const charT* high) const;

  Returns: do_scan_is(m,low,high)
```

```
const charT* scan_not(mask m,
    const charT* low, const charT* high) const;

  Returns: do_scan_not(m,low,high)
```

```
charT   toupper(charT) const;
const charT* toupper(charT* low, const charT* high) const;

  Returns: do_toupper(c) or do_toupper(low,high)
```

```
charT   tolower(charT c) const;
const charT* tolower(charT* low, const charT* high) const;

  Returns: do_tolower(c) or do_tolower(low,high)
```

```
charT   widen(char c) const;
const char* widen(const char* low, const char* high, charT* to) const;

  Returns: do_widen(c) or do_widen(low,high,to)
```

```
narrow(charT c, char dfault) const;
const charT* narrow(const charT* low, const charT* high, charT* to) const;

  Returns: do_narrow(c,dfault) or do_narrow(low,high,dfault,to)
```

22.2.1.1.2 ctype virtual functions

```
bool    do_is(mask m, charT c) const;
const charT* do_is(const charT* low, const charT* high,
    mask* vec) const;
```

**Effects:** Classifies a character or sequence of characters. For each argument character, identifies a value \( M \) of type `ctype_base::mask`. The second form identifies a value \( M \) of type `ctype_base::mask` for each \( *p \) where \( \text{low} \leq p < \text{high} \), 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`.

```
const charT* 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 \( \text{is}(m,*p) \) would return `true`; otherwise, returns `high`.

```
const charT* low, const charT* high) const;

Effects: Locates a character in a buffer that fails to conform to a classification m.

Returns: The smallest pointer p, if any, in the range [low,high) such that is(m,*p) would return false; otherwise, returns high.

do_toupper(charT c) const;
do_toupper(charT* low, const charT* high) const;

Effects: Converts a character or characters to upper case. The second form replaces each character *p in the range [low,high) for which a corresponding upper-case character exists, with that character.

Returns: The first form returns the corresponding upper-case character if it is known to exist, or its argument if not. The second form returns high.

do_tolower(charT c) const;
do_tolower(charT* low, const charT* high) const;

Effects: Converts a character or characters to lower case. The second form replaces each character *p in the range [low,high) and for which a corresponding lower-case character exists, with that character.

Returns: The first form returns the corresponding lower-case character if it is known to exist, or its argument if not. The second form returns high.

do_widen(char c) const;
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.\(^{235}\) 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, (ctc.is(M, c) || !is(M, do_widen(c))) is true.\(^{236}\)

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.

do_narrow(charT c, char dfault) const;
do_narrow(const charT* low, const charT* high, char dfault, charT* 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

\[
do_widen(do_narrow(c,0)) == c
\]

For any named ctype category with a ctype<char> facet ctc however, and ctype_base::mask value M,

\(^{235}\) The char argument of do_widen is intended to accept values derived from character literals for conversion to the locale’s encoding.

\(^{236}\) In other words, the transformed character is not a member of any character classification that c is not also a member of.
is true (unless do_narrow returns dfault). In addition, for any digit character c, the expression (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 ctype_byname

```cpp
namespace std {
    template <class charT>
    class ctype_byname : public ctype<charT> {
        public:
            typedef typename ctype<charT>::mask mask;
            explicit ctype_byname(const char*, size_t refs = 0);
            explicit ctype_byname(const string&, size_t refs = 0);
        protected:
            ~ctype_byname();
    };
}
```

### 22.2.1.3 ctype specializations

```cpp
namespace std {
    template <> class ctype<char> : public locale::facet, public ctype_base {
        public:
            char char_type;
            explicit ctype(const mask* tab = 0, bool del = false,
                            size_t refs = 0);

            bool is(mask m, char c) const;
            const char* is(const char* low, const char* high, mask* vec) const;
            const char* scan_is (mask m,
                                 const char* low, const char* high) const;
            const char* scan_not(mask m,
                                   const char* low, const char* high) const;

            char toupper(char c) const;
            const char* toupper(char* low, const char* high) const;
            char tolower(char c) const;
            const char* tolower(char* low, const char* high) const;

            char widen(char c) const;
            const char* widen(const char* low, const char* high, char* to) const;
    };
}
```
A specialization `ctype<char>` is provided so that the member functions on type `char` can be implemented inline.\(^2\) The implementation-defined value of member `table_size` is at least 256.

### 22.2.1.3.1 `ctype<char>` Destructor

`~ctype();`

**Effects:** If the constructor's first argument was nonzero, and its second argument was true, does delete [] `table()`.

### 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.

```cpp
explicit ctype(const mask* tbl = 0, bool del = false,
               size_t refs = 0);
```

\(^2\) 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
Precondition: tbl either 0 or an array of at least \texttt{table\_size} elements.

Effects: Passes its \texttt{refs} argument to its base class constructor.

\begin{verbatim}
bool is(mask m, char c) const;
const char* is(const char* low, const char* high,
               mask* vec) const;
\end{verbatim}

Effects: The second form, for all \texttt{*p} in the range \texttt{[low,high)}, assigns into \texttt{vec[p-low]} the value \texttt{table()[(unsigned char)*p]}.

Returns: The first form returns \texttt{table()[(unsigned char)c] & m}; the second form returns \texttt{high}.

\begin{verbatim}
const char* scan_is(mask m, const char* low, const char* high) const;
\end{verbatim}

Returns: The smallest \texttt{p} in the range \texttt{[low,high)} such that \texttt{table()[(unsigned char)*p] & m} is true.

\begin{verbatim}
const char* scan_not(mask m, const char* low, const char* high) const;
\end{verbatim}

Returns: The smallest \texttt{p} in the range \texttt{[low,high)} such that \texttt{table()[(unsigned char)*p] & m} is false.

\begin{verbatim}
char toupper(char c) const;
const char* toupper(char* low, const char* high) const;
\end{verbatim}

Returns: \texttt{do_toupper(c)} or \texttt{do_toupper(low,high)}, respectively.

\begin{verbatim}
char tolower(char c) const;
const char* tolower(char* low, const char* high) const;
\end{verbatim}

Returns: \texttt{do_tolower(c)} or \texttt{do_tolower(low,high)}, respectively.

\begin{verbatim}
char widen(char c) const;
const char* widen(const char* low, const char* high, char* to) const;
\end{verbatim}

Returns: \texttt{do_widen(c)} or \texttt{do_widen(low, high, to)}, respectively.

\begin{verbatim}
char narrow(char c, char dfault) const;
const char* narrow(const char* low, const char* high, char dfault, char* to) const;
\end{verbatim}

Returns: \texttt{do_narrow(c, dfault)} or \texttt{do_narrow(low, high, dfault, to)}, respectively.

\begin{verbatim}
const mask* table() const throw();
\end{verbatim}

Returns: The first constructor argument, if it was non-zero, otherwise \texttt{classic_table()}. 

Draft
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

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 {
    public:
        typedef internT intern_type;
        typedef externT extern_type;
        typedef stateT state_type;

        explicit codecvt(size_t refs = 0);

        result out(stateT& state,
            const internT* from, const internT* from_end, const internT*& from_next,
            externT* to, externT* to_limit, externT*& to_next) const;

        result unshift(stateT& state,
            externT* to, externT* to_limit, externT*& to_next) const;

        result in(stateT& state,
            const externT* from, const externT* from_end, const externT*& from_next,
            internT* to, internT* to_limit, internT*& to_next) const;

    };
22.2 Standard locale categories

int encoding() const throw();
bool always_noconv() const throw();
int length(stateT&, const externT* from, const externT* end,
size_t max) const;
int max_length() const throw();

static locale::id id;

protected:
~codecvt();

virtual result do_out(stateT& state,
    const internT* from, const internT* from_end, const internT*& from_next,
    externT* to, externT* to_limit, externT*& to_next) const;
virtual result do_in(stateT& state,
    const externT* from, const externT* from_end, const externT*& from_next,
    internT* to, internT* to_limit, internT*& to_next) const;
virtual result do_unshift(stateT& state,
    externT* to, externT* to_limit, externT*& to_next) const;
virtual int do_encoding() const throw();
virtual bool do_always_noconv() const throw();
virtual int do_length(stateT&, const externT* from,
    const externT* end, size_t max) const;
virtual int do_max_length() const throw();
};

The class \texttt{codecvt<internT,externT,stateT>} 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 \texttt{stateT} argument selects the pair of codesets being mapped between.

The specializations required in Table 74 (22.1.1.1.1) convert the implementation-defined native character set. \texttt{codecvt<char,char,mbstate_t>} implements a degenerate conversion; it does not convert at all. The specialization \texttt{codecvt<char16_t,char,mbstate_t>} converts between the UTF-16 and UTF-8 encodings schemes, and the specialization \texttt{codecvt<char32_t,char,mbstate_t>} converts between the UTF-32 and UTF-8 encodings schemes. \texttt{codecvt<wchar_t,char,mbstate_t>} converts between the native character sets for narrow and wide characters. Specializations on \texttt{mbstate_t} perform conversion between encodings known to the library implementor. Other encodings can be converted by specializing on a user-defined \texttt{stateT} type. The \texttt{stateT} object can contain any state that is useful to communicate to or from the specialized \texttt{do_in} or \texttt{do_out} members.

22.2.1.4.1 \texttt{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;

Returns: \texttt{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. 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.

2. Effects: Translates characters in the source range [from, from_end), placing the results in sequential positions starting at destination to. Converts no more than (from_end-from) source elements, and stores no more than (to_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 noconv, internT and externT are the same type and the converted sequence is identical to the input sequence [from, from_next). to_next is set equal to to, the value of state is unchanged, and there are no changes to the values in [to, to_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 ok, where from ! = from_end, then
do_out(state, from, from + 1, from_next, to, to_end, to_next)

shall also return ok, and that if

do_in(state, from, from_end, from_next, to, to_limit, to_next)

would return ok, where to != to_end, then

do_in(state, from, from_end, from_next, to, to + 1, to_next)

shall also return ok.\(^238\) [Note: As a result of operations on state, it can return ok or partial and set next == from and to_next != to. —end note]

Remarks: Its operations on state are unspecified. [Note: This argument can be used, for example, to maintain shift state, to specify conversion options (such as count only), or to identify a cache of seek offsets. —end note]

Returns: An enumeration value, as summarized in Table 76:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ok</td>
<td>completed the conversion</td>
</tr>
<tr>
<td>partial</td>
<td>not all source characters converted</td>
</tr>
<tr>
<td>error</td>
<td>encountered a character in [from,from_end) that it could not convert</td>
</tr>
<tr>
<td>noconv</td>
<td>internT and externT are the same type, and input sequence is identical to converted sequence</td>
</tr>
</tbody>
</table>

A return value of partial, if (from_next==from_end), indicates that either the destination sequence has not absorbed all the available destination elements, or that additional source elements are needed before another destination element can be produced.

result do_unshift(stateT& state,
externT* to, externT* to_limit, externT*& to_next) const;

Requires: (to <= to_end) well defined and true; state initialized, if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.

Effects: Places characters starting at to that should be appended to terminate a sequence when the current stateT is given by stateT.\(^239\) Stores no more than (to_limit-to) destination elements, and leaves the to_next pointer pointing one beyond the last element successfully stored.\(^\text{codecvt<char, char, mbstate_t> stores no characters.}\)

Returns: An enumeration value, as summarized in Table 77:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>codecvt&lt;char, char, mbstate_t&gt; returns noconv.</td>
<td></td>
</tr>
</tbody>
</table>

\(^{238}\)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.

\(^{239}\)Typically these will be characters to return the state to stateT().
Table 77: `do_unshift` result values

<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 <code>to_limit</code> to destination elements was needed to terminate a sequence given the value of state</td>
</tr>
<tr>
<td>error</td>
<td>an unspecified error has occurred</td>
</tr>
<tr>
<td>noconv</td>
<td>no termination is needed for this state_type</td>
</tr>
</tbody>
</table>

9  
Returns: -1 if the encoding of the externT sequence is state-dependent; else the constant number of externT characters needed to produce an internal character; or 0 if this number is not a constant.\(^{240}\)

```cpp
bool do_always_noconv() const throw();
```

10 
Returns: true if `do_in()` and `do_out()` return noconv for all valid argument values. `codecvt<char, char, mbstate_t>::do_in()` returns true.

```cpp
int do_length(stateT& state, const externT* from, const externT* from_end, size_t max) const;
```

Preconditions: (from<=from_end) well-defined and true; state initialized, if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.

Effects: The effect on the state argument is “as if” it called `do_in(state, from, from_end, from, to, to+max, to)` for to pointing to a buffer of at least max elements.

Returns: (from_next-from) where from_next is the largest value in the range [from,from_end] such that the sequence of values in the range [from,from_next) represents max or fewer valid complete characters of type internT. The specialization `codecvt<char, char, mbstate_t>::do_length()` returns the lesser of max and (from_-end-from).

```cpp
int do_max_length() const throw();
```

Returns: The maximum value that `do_length(state, from, from_end, 1)` can return for any valid range [from, from_end] and stateT value state. The specialization `codecvt<char, char, mbstate_t>::do_max_length()` returns 1.

22.2.1.5 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();
```

\(^{240}\) If `encoding()` yields -1, then more than `max_length()` externT elements may be consumed when producing a single internT character, and additional externT elements may appear at the end of a sequence after those that yield the final internT character.
22.2 Standard locale categories

### 22.2.2 The numeric category

The classes `num_get<>` and `num_put<>` handle numeric formatting and parsing. Virtual functions are provided for several numeric types. Implementations may (but are not required to) delegate extraction of smaller types to extractors for larger types.

All specifications of member functions for `num_put` and `num_get` in the subclauses of 22.2.2 only apply to the specializations required in Tables 74 and 75 (22.1.1.1.1), namely `num_get<char>`, `num_get<wchar_t>`, `num_get<C, InputIterator>`, `num_put<char>`, `num_put<wchar_t>`, and `num_put<C, OutputIterator>`. These specializations refer to the `ios_base&` argument for formatting specifications (22.2), 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 `num_get<>` and `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` [locale.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 in, iter_type end, ios_base&,
                       ios_base::iostate& err, bool& v) const;
        iter_type get(iter_type in, iter_type end, ios_base&,
                       ios_base::iostate& err, long& v) const;
        iter_type get(iter_type in, iter_type end, ios_base&,
                       ios_base::iostate& err, long long& v) const;
        iter_type get(iter_type in, iter_type end, ios_base&,
                       ios_base::iostate& err, unsigned short& v) const;
        iter_type get(iter_type in, iter_type end, ios_base&,
                       ios_base::iostate& err, unsigned int& v) const;
        iter_type get(iter_type in, iter_type end, ios_base&,
                       ios_base::iostate& err, unsigned long& v) const;
        iter_type get(iter_type in, iter_type end, ios_base&,
                       ios_base::iostate& err, float& v) const;
        iter_type get(iter_type in, iter_type end, ios_base&,
                       ios_base::iostate& err, double& v) const;
        iter_type get(iter_type in, iter_type end, ios_base&,
                       ios_base::iostate& err, long double& v) const;
    }
}
```

241) Parsing "-1" correctly into (e.g.) an unsigned short requires that the corresponding member `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

```
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, bool& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, long long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned short& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned int& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned long long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, float& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, double& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, long double& val) const;
```
22.2 Standard locale categories

iter_type get(iter_type in, iter_type end, ios_base& str,
ios_base::iostate& err, float& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
ios_base::iostate& err, double& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
ios_base::iostate& err, long double& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
ios_base::iostate& err, void*& val) const;

Returns: do_get(in, end, str, err, val).

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;

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

fmtflags flags = str.flags();

Draft
```cpp
define localization

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 78. The table is ordered. That is, the first line whose condition is true applies.

### Table 78: Integer Conversions

<table>
<thead>
<tr>
<th>State</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>basefield == oct</td>
<td>%o</td>
</tr>
<tr>
<td>basefield == hex</td>
<td>%X</td>
</tr>
<tr>
<td>basefield == 0</td>
<td>%i</td>
</tr>
<tr>
<td>signed integral type</td>
<td>%d</td>
</tr>
<tr>
<td>unsigned integral type</td>
<td>%u</td>
</tr>
</tbody>
</table>

For conversions to a floating 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 79.

### Table 79: Length Modifier

<table>
<thead>
<tr>
<th>type</th>
<th>length modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>h</td>
</tr>
<tr>
<td>unsigned short</td>
<td>h</td>
</tr>
<tr>
<td>long</td>
<td>l</td>
</tr>
<tr>
<td>unsigned long</td>
<td>l</td>
</tr>
<tr>
<td>long long</td>
<td>ll</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>ll</td>
</tr>
<tr>
<td>double</td>
<td>l</td>
</tr>
<tr>
<td>long double</td>
<td>L</td>
</tr>
</tbody>
</table>

**Stage 2:** If in==end then stage 2 terminates. Otherwise a charT is taken from in and local variables are initialized as if by

```cpp
char_type ct = *in;
char c = src[find(atoms, atoms + sizeof(src) - 1, ct) - atoms];
if (ct == use_facet<numpunct<charT>>(loc).decimal_point())
c = '.';
bool discard =
ct == use_facet<numpunct<charT>>(loc).thousands_sep()
& use_facet<numpunct<charT>>(loc).grouping().length() != 0;
```

where the values src and atoms are defined as if by:

```cpp
static const char src[] = "0123456789abcdefxABCDEFGHIJKLMNOPQRSTUVWXYZ-";
```
22.2 Standard locale categories

Localization library 650

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.

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.
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 iter_type do_put(iter_type s, ios_base& f, char_type fill, bool v) const;
    virtual iter_type do_put(iter_type s, ios_base& f, char_type fill, long v) const;
    virtual iter_type do_put(iter_type s, ios_base& f, char_type fill, long long v) const;
    virtual iter_type do_put(iter_type s, ios_base& f, char_type fill, unsigned long) const;
    virtual iter_type do_put(iter_type s, ios_base& f, char_type fill, unsigned long long) const;
    virtual iter_type do_put(iter_type s, ios_base& f, char_type fill, double v) const;
    virtual iter_type do_put(iter_type s, ios_base& f, char_type fill, long double v) const;
    virtual iter_type do_put(iter_type s, ios_base& f, char_type fill, const void* v) const;
};
}
22.2 Standard locale categories

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

```
    iter_type put(iter_type out, ios_base& str, char_type fill, 
                  bool val) const;
    iter_type put(iter_type out, ios_base& str, char_type fill, 
                  long val) const;
    iter_type put(iter_type out, ios_base& str, char_type fill, 
                  long long val) const;
    iter_type put(iter_type out, ios_base& str, char_type fill, 
                  unsigned long val) const;
    iter_type put(iter_type out, ios_base& str, char_type fill, 
                  unsigned long long val) const;
    iter_type put(iter_type out, ios_base& str, char_type fill, 
                  double val) const;
    iter_type put(iter_type out, ios_base& str, char_type fill, 
                  long double val) const;
    iter_type put(iter_type out, ios_base& str, char_type fill, 
                  const void* val) const;
```

Returns: do_put(out, str, fill, val).

22.2.2.2.2 num_put virtual functions

```
    iter_type do_put(iter_type out, ios_base& str, char_type fill, 
                     long val) const;
    iter_type do_put(iter_type out, ios_base& str, char_type fill, 
                     long long val) const;
    iter_type do_put(iter_type out, ios_base& str, char_type fill, 
                     unsigned long val) const;
    iter_type do_put(iter_type out, ios_base& str, char_type fill, 
                     unsigned long long val) const;
    iter_type do_put(iter_type out, ios_base& str, char_type fill, 
                     double val) const;
    iter_type do_put(iter_type out, ios_base& str, char_type fill, 
                     long double val) const;
    iter_type do_put(iter_type out, ios_base& str, char_type fill, 
                     const void* val) const;
```

Effects: Writes characters to the sequence out, formatting val as desired. In the following description, a local variable initialized with

```c
    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

```c
    printf(spec, val)
```

Draft
assuming that the current locale is the "C" locale.

— Stage 2: Adjust the representation by converting each char determined by stage 1 to a charT using a conversion and values returned by members of use_facet< numpunct<charT> >(str.getloc())

— Stage 3: Determine where padding is required.

— Stage 4: Insert the sequence into the out.

Detailed descriptions of each stage follow.

Returns: out.

Stage 1: The first action of stage 1 is to determine a conversion specifier. The tables that describe this determination use the following local variables

fmtflags flags = str.flags();
fmtflags basefield = (flags & (ios_base::basefield));
fmtflags uppercase = (flags & (ios_base::uppercase));
fmtflags floatfield = (flags & (ios_base::floatfield));
fmtflags showpos = (flags & (ios_base::showpos));
fmtflags showbase = (flags & (ios_base::showbase));

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 80.

<table>
<thead>
<tr>
<th>State</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>basefield == ios_base::oct</td>
<td>%o</td>
</tr>
<tr>
<td>(basefield == ios_base::hex) &amp;&amp; !uppercase</td>
<td>%x</td>
</tr>
<tr>
<td>(basefield == ios_base::hex)</td>
<td>%X</td>
</tr>
<tr>
<td>for a signed integral type</td>
<td>%d</td>
</tr>
<tr>
<td>for an unsigned integral type</td>
<td>%u</td>
</tr>
</tbody>
</table>

For conversion from a floating-point type, the function determines the floating-point conversion specifier as indicated in Table 81:

For conversions from an integral or floating type a length modifier is added to the conversion specifier as indicated in Table 82.

The conversion specifier has the following optional additional qualifiers prepended as indicated in Table 83.

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.
### 22.2 Standard locale categories

#### Table 81: 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>%a</td>
</tr>
<tr>
<td>!uppercase</td>
<td>%A</td>
</tr>
<tr>
<td>otherwise</td>
<td>%g</td>
</tr>
<tr>
<td>otherwise</td>
<td>%G</td>
</tr>
</tbody>
</table>

#### Table 82: 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>

#### Table 83: Numeric conversions

<table>
<thead>
<tr>
<th>Type(s)</th>
<th>State</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>an integral type</td>
<td>flags &amp; showpos</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>flags &amp; showbase</td>
<td>#</td>
</tr>
<tr>
<td>a floating-point type</td>
<td>flags &amp; showpos</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>flags &amp; showpoint</td>
<td>#</td>
</tr>
</tbody>
</table>
**Stage 2:** Any character 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

```cpp
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

```cpp
fmtflags adjustfield= (flags & (ios_base::adjustfield));
```

The location of any padding\(^{242}\) is determined according to Table 84

<table>
<thead>
<tr>
<th>State</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>adjustfield == ios_base::left</code></td>
<td>pad after</td>
</tr>
<tr>
<td><code>adjustfield == ios_base::right</code></td>
<td>pad before</td>
</tr>
<tr>
<td><code>adjustfield == internal</code></td>
<td></td>
</tr>
<tr>
<td>and a sign occurs in the representation</td>
<td>pad after the sign</td>
</tr>
<tr>
<td><code>adjustfield == internal</code></td>
<td></td>
</tr>
<tr>
<td>and representation after stage 1 began with 0x or 0X</td>
<td>pad after x or X</td>
</tr>
<tr>
<td><code>otherwise</code></td>
<td>pad before</td>
</tr>
</tbody>
</table>

If `str.width()` is nonzero and the number of `charT`'s in the sequence after stage 2 is less than `str.width()`, then enough fill characters are added to the sequence at the position indicated for padding to bring the length of the sequence to `str.width()`.

`str.width(0)` is called.

**Stage 4:** The sequence of `charT`'s at the end of stage 3 are output via

```cpp
*out++ = c
```

6. `iter_type do_put(iter_type out, ios_base& str, char_type fill, bool val) const;`

**Returns:** If `str.flags() & ios_base::boolalpha) == 0` returns `do_put(out, str, fill, (int)val)`. otherwise obtains a string `s` as if by

```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`.

\(^{242}\) The conversion specification `#o` generates a leading 0 which is not a padding character.

Draft
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
        }
    }
}
```

1 numpunct<> specifies numeric punctuation. The specializations required in Table 74 (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.

2 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;

Returns: do_decimal_point()
```

```
char_type thousands_sep() const;

Returns: do_thousands_sep()
```

```
string grouping() const;

Returns: do_grouping()
```

```
string_type truename() const;
string_type falsename() const;

Returns: do_truename() or do_falsename(), respectively.
```

### 22.2.3.1.2 numpunct virtual functions

```
char_type do_decimal_point() const;

Returns: A character for use as the decimal radix separator. The required specializations return '.' or L'.'.
```

```
char_type do_thousands_sep() const;

Returns: A character for use as the digit group separator. The required specializations return ', ' or L','.
```

```
string do_grouping() const;

Returns: A basic_string<char> vec used as a vector of integer values, in which each element vec[i] represents the number of digits in the group at position i, starting with position 0 as the rightmost group. If vec.size() <= i, the number is the same as group (i-1); if (i<0 || vec[i]<0 || vec[i]==CHAR_MAX), the size of the digit group is unlimited.
```

```
string_type do_truename() const;
string_type do_falsename() const;

Returns: A string representing the name of the boolean value true or false, respectively.
```

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".

---

Draft
22.2.3.2 Class template numpunctbyname

```cpp
namespace std {
    template <class charT>
    class numpunctbyname : public numpunct<charT> {
    // this class is specialized for char and wchar_t.
    public:
        typedef charT char_type;
        typedef basic_string<charT> string_type;
        explicit numpunctbyname(const char*, size_t refs = 0);
        explicit numpunctbyname(const string&, size_t refs = 0);
    protected:
        ~numpunctbyname();
    }
}
```

22.2.4 The collate category

22.2.4.1 Class template collate

```cpp
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;
        static locale::id id;
    protected:
        ~collate();
        virtual int do_compare(const charT* low1, const charT* high1,
                               const charT* low2, const charT* high2) const;
        virtual string_type do_transform(const charT* low, const charT* high) const;
        virtual long do_hash (const charT* low, const charT* high) const;
    }
}
```

The class `collate<charT>` provides features for use in the collation (comparison) and hashing of strings. A locale member function template, `operator()`, uses the collate facet to allow a locale to act directly as the predicate argument for standard algorithms (clause 25) and containers operating on strings. The specializations required in Table 74 (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:* do_transform(low, high)

```cpp
long hash(const charT* low, const charT* high) const;
```

*Returns:* do_hash(low, high)

### 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 74 (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.

```cpp
long do_hash(const charT* low, const charT* high) const;
```

*Returns:* An integer value equal to the result of calling hash() on any other string for which do_compare() returns 0 (equal) when passed the two strings. *Note:* The probability that the result equals that for another string which does not compare equal should be very small, approaching \((1.0/\text{numeric\_limits\_unsigned\_long\_max})\). — end note

### 22.2.4.2 Class template collate_byname

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();
```

244 This function is useful when one string is being compared to many other strings.
22.2 Standard locale categories

### 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 74 and 75 (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`

```c++
namespace std {
    class time_base {
    public:
        enum dateorder { no_order, dmy, mdy, ymd, ydm };
    };

    template <class charT, class InputIterator = istreambuf_iterator<charT> >
    class time_get : public locale::facet, public time_base {
    public:
        typedef charT char_type;
        typedef InputIterator iter_type;

        explicit time_get(size_t refs = 0);

        dateorder date_order() const { return do_date_order(); }
        iter_type get_time(iter_type s, iter_type end, ios_base& f,
                           ios_base::iostate& err, tm* t) const;
        iter_type get_date(iter_type s, iter_type end, ios_base& f,
                           ios_base::iostate& err, tm* t) const;
        iter_type get_weekday(iter_type s, iter_type end, ios_base& f,
                               ios_base::iostate& err, tm* t) const;
        iter_type get_monthname(iter_type s, iter_type end, ios_base& f,
                                ios_base::iostate& err, tm* t) const;
        iter_type get_year(iter_type s, iter_type end, ios_base& f,
                            ios_base::iostate& err, tm* t) const;
        iter_type get(iter_type s, iter_type end, ios_base& f,
                      char format, char modifier = 0) const;
        iter_type get(iter_type s, iter_type end, ios_base& f,
                      const char_type *fmt, const char_type *fmtend) const;

        static locale::id id;
    };
}
```

protected:
```
"time_get();"
```
```
virtual dateorder do_date_order() const;
virtual iter_type do_get_time(iter_type s, iter_type end, ios_base&,
                              ios_base::iostate& err, tm* t) const;
virtual iter_type do_get_date(iter_type s, iter_type end, ios_base&,
                              ios_base::iostate& err, tm* t) const;
```
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.\(^{245}\)

If the end iterator is reached during parsing by any of the `get()` member functions, the member sets `ios_base::eof-bit` in `err`.

### 22.2.5.1.1 time_get members

#### dateorder date_order() const;

*Returns: do_date_order()*

#### iter_type get_time(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm* t) const;

*Returns: do_get_time(s, end, str, err, t)*

#### iter_type get_date(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm* t) const;

*Returns: do_get_date(s, end, str, err, t)*

#### iter_type get_weekday(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm* t) const;

#### iter_type get_monthname(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm* t) const;

#### iter_type get_year(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm* t) const;

#### iter_type get(iter_type s, iter_type end, ios_base& f, ios_base::iostate& err, tm* t, char format, char modifier) const;

\(^{245}\) 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.
22.2 Standard locale categories

Returns: do_get(s, end, f, err, t, format, modifier)

iter_type get(iter_type s, iter_type end, ios_base& f,
ios_base::iostate& err, tm *t, const char_type *fmt, const char_type *fmtend) const;

Requires: [fmt,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, f.getloc()) is true, and finally resumes looping.
— The next character read from s matches the element pointed to by fmt in a case-insensitive comparison, in which case the function evaluates ++fmt, ++s and continues looping. Otherwise, the function evaluates err = ios_base::failbit.

[Note: The function uses the ctype<charT> facet installed in f’s locale to determine valid whitespace characters. It is unspecified by what means the function performs case-insensitive comparison or whether multi-character sequences are considered while doing so.

22.2.5.1.2 time_get virtual functions

[locale.time.get.virtuals]

dateorder do_date_order() const;

Returns: An enumeration value indicating the preferred order of components for those date formats that are composed of day, month, and year.\(^{246}\) Returns no_order if the date format specified by 'x' contains other variable components (e.g. Julian day, week number, week day).

\(^{246}\) This function is intended as a convenience only, for common formats, and may return no_order in valid locales.
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 85:

<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.

iter_type do_get_weekday(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm* t) const;

Effects: Reads characters starting at s until it has extracted the (perhaps abbreviated) name of a weekday or month. If it finds an abbreviation that is followed by characters that could match a full name, it continues reading until it matches the full name or fails. It sets the appropriate struct tm member accordingly.

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid name.

iter_type do_get_monthname(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm* t) const;

Effects: Reads characters starting at s until it has extracted an unambiguous year identifier. It is implementation-defined whether two-digit year numbers are accepted, and (if so) what century they are assumed to lie in. Sets the t->tm_year member accordingly.

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid year identifier.

iter_type do_get_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.


Requires: [fmt, end) shall be a valid range and t shall be dereferenceable.

Effects: The function starts by evaluating \( \text{err} = \text{ios\_base\::\:goodbit} \). 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 \text{strptime}, formed by concatenating '%', the modifier character, when non-NUL, and the format character. When the concatenation fails to yield a complete valid directive the function leaves the object pointed to by t unchanged and evaluates \( \text{err} \|= \text{ios\_base\::\:failbit} \). When s == end evaluates to true after reading a character the function evaluates \( \text{err} \|= \text{ios\_base\::\:eofbit} \).

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 \( \text{err} \|= \text{ios\_base\::\:eofbit} \). 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 \text{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 \text{time\_get\_byname} [locale.time.getbyname]

namespace std {
    template <class charT, class InputIterator = istreambuf_iterator<charT> >
    class time_get_byname : public time_get<charT, InputIterator> {
        public:
            typedef time_base::dateorder dateorder;
            typedef InputIterator iter_type;

            explicit time_get_byname(const char*, size_t refs = 0);
            explicit time_get_byname(const string&, size_t refs = 0);
        protected:
            ~time_get_byname();
            
    }
}

22.2.5.3 Class template \text{time\_put} [locale.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.

    }
}
665 Localization library 22.2 Standard locale categories

iter_type put(iter_type s, ios_base& f, char_type fill, const tm* tmb,
const charT* pattern, const charT* pat_end) const;
iter_type put(iter_type s, ios_base& f, char_type fill,
const tm* tmb, char format, char modifier = 0) const;

static locale::id id;

protected:
 ~time_put();
 virtual iter_type do_put(iter_type s, ios_base&, char_type, const tm* t,
 char format, char modifier) const;
};

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).

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& str, char_type fill, const tm* t,
 char format, char modifier) const;

Effects: Formats the contents of the parameter t into characters placed on the output sequence s. Formatting is controlled by the parameters format and modifier, interpreted identically as the format specifiers in the string argument to the standard library function strftime().248 except that the sequence of characters produced for

---

247) Although the C programming language defines no modifiers, most vendors do.
248) Interpretation of the modifier argument is implementation-defined, but should follow POSIX conventions.

Draft
those specifiers that are described as depending on the C locale are instead implementation-defined.\(^{(249)}\)

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`

```cpp
namespace std {
    template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
    class time_put_byname : public time_put<charT, OutputIterator> {
        public:
            typedef charT char_type;
            typedef OutputIterator iter_type;
            explicit time_put_byname(const char*, size_t refs = 0);
            explicit time_put_byname(const string&, size_t refs = 0);
        protected:
            "time_put_byname();
    }
}
```

### 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 74 and 75 (22.1.1.1.1). Their members use their `ios_base&`, `ios_base::iostate&`, and fill arguments as described in (22.2), and the `moneypunct<>` and `ctype<>` facets, to determine formatting details.

### 22.2.6.1 Class template `money_get`

```cpp
namespace std {
    template <class charT, class InputIterator = istreambuf_iterator<charT> >
    class money_get : public locale::facet {
        public:
            typedef charT char_type;
            typedef InputIterator iter_type;
            typedef basic_string<charT> string_type;
            explicit money_get(size_t refs = 0);

            iter_type get(iter_type s, iter_type end, bool intl, 
            ios_base& f, ios_base::iostate& err, 
            long double& units) const;
            iter_type get(iter_type s, iter_type end, bool intl,
```
ios_base& f, ios_base::iostate& err,
string_type& digits) const;

static locale::id id;

protected:
~money_get();
virtual iter_type do_get(iter_type, iter_type, bool, ios_base&,
ios_base::iostate& err, long double& units) const;
virtual iter_type do_get(iter_type, iter_type, bool, ios_base&,
ios_base::iostate& err, string_type& digits) const;
};
}

22.2.6.1.1 money_get members

iter_type get(iter_type s, iter_type end, bool intl,
ios_base& f, ios_base::iostate& err,
long double& quant) const;
iter_type get(s, iter_type end, bool intl, ios_base&f,
ios_base::iostate& err, string_type& quant) const;

Returns: do_get(s, end, intl, f, err, quant)

22.2.6.1.2 money_get virtual functions

iter_type do_get(iter_type s, iter_type end, bool intl,
ios_base& str, ios_base::iostate& err,
long double& units) const;
iter_type do_get(iter_type s, iter_type end, bool intl,
ios_base& str, ios_base::iostate& err,
string_type& digits) const;

Effects: Reads characters from s to parse and construct a monetary value according to the format specified by a moneypunct<charT,Intl> facet reference mp and the character mapping specified by a ctype<charT> facet reference ct obtained from the locale returned by str.getloc(), and str.flags(). If a valid sequence is recognized, does not change err; otherwise, sets err to (err|str.failbit), or (err|str.failbit|str.eofbit) if no more characters are available, and does not change units or digits. Uses the pattern returned by mp.neg_format() to parse all values. The result is returned as an integral value stored in units or as a sequence of digits possibly preceded by a minus sign (as produced by ct.widen(c) where c is \'-\' or in the range from '0' through '9', inclusive) stored in digits. [Example: The sequence $1,056.23 in a common United States locale would yield, for units, 105623, or, for digits, "105623". —end example ] If mp.grouping() indicates that no thousands separators are permitted, any such characters are not read, and parsing is terminated at the point where they first appear. Otherwise, thousands separators are optional; if present, they are checked for correct placement only after all format components have been read.

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

Draft
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 250

\[
\text{for (int i = 0; i < n; ++i) } \\
\text{buf2[i] = src[find(atoms, atoms+sizeof(src), buf1[i]) - atoms]; } \\
\text{buf2[n] = 0; } \\
\text{sscanf(buf2, "%Lf", &units);}
\]

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

\[
\text{static const char src[] = "0123456789-"; } \\
\text{charT atoms[sizeof(src)]; } \\
\text{ct.widen(src, src + sizeof(src) - 1, atoms);} \\
\]

5

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid monetary quantity.

22.2.6.2 Class template money_put

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, 

250) The semantics here are different from ct.narrow.

Draft
char_type fill, const string_type& digits) const;
static locale::id id;

protected:
"money_put();
virtual iter_type do_put(iter_type, bool, ios_base&, char_type fill,
long double units) const;
virtual iter_type do_put(iter_type, bool, ios_base&, char_type fill,
const string_type& digits) const;
};

22.2.6.2.1 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;

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;

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
ct.widen(buf1, buf1 + sprintf(buf1, "%.0Lf", units), buf2)

for character buffers buf1 and buf2. If the first character in digits or buf2 is equal to 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
c) are used; any trailing characters (including digits appearing after a non-digit character) are ignored. Calls
str.width(0).

Remarks: The currency symbol is generated if and only if (str.flags() & str.showbase) is nonzero. If the
number of characters generated for the specified format is less than the value returned by str.width() on entry to
the function, then copies of fill are inserted as necessary to pad to the specified width. For the value af equal to
(str.flags() & str.adjustfield), if (af == str.internal) is true, the fill characters are placed where
none or space appears in the formatting pattern; otherwise if (af == str.left) is true, they are placed after

Draft
the other characters; otherwise, they are placed before the other characters. [Note: It is possible, with some combinations of format patterns and flag values, to produce output that cannot be parsed using num_get<>::get. —end note]

Returns: An iterator pointing immediately after the last character produced.

### 22.2.6.3 Class template moneyprintf

```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 moneyprintf : public locale::facet, public money_base {
        public:
            typedef charT char_type;
            typedef basic_string<charT> string_type;
            
            explicit moneyprintf(size_t refs = 0);
            charT decimal_point() const;
            charT thousands_sep() const;
            string grouping() const;
            string_type curr_symbol() const;
            string_type positive_sign() const;
            string_type negative_sign() const;
            int frac_digits() const;
            pattern pos_format() const;
            pattern neg_format() const;

            static locale::id id;
            static const bool intl = International;

        protected:
            "moneyprintf();
            virtual charT do_decimal_point() const;
            virtual charT do_thousands_sep() const;
            virtual string do_grouping() const;
            virtual string_type do_curr_symbol() const;
            virtual string_type do_positive_sign() const;
            virtual string_type do_negative_sign() const;
            virtual int do_frac_digits() const;
            virtual pattern do_pos_format() const;
            virtual pattern do_neg_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 ith component of the format. In the field member of a pattern object, each value symbol, sign, value, and either space or none appears exactly once. The value none, if present, is not first; the value space, if present, is neither first nor last.

Where none or space appears, white space is permitted in the format, except where none appears at the end, in which case no white space is permitted. The value space indicates that at least one space is required at that position. Where symbol appears, the sequence of characters returned by curr_symbol() is permitted, and can be required. Where sign appears, the first (if any) of the sequence of characters returned by positive_sign() or negative_sign() (respectively as the monetary value is non-negative or negative) is required. Any remaining characters of the sign sequence are required after all other format components. Where value appears, the absolute numeric monetary value is required.

The format of the numeric monetary value is a decimal number:

\[
\text{value ::= units \[ \text{decimal-point \[ digits \]} \]} \mid \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:

\[
\begin{align*}
\text{units ::= digits \[ \text{thousands-sep units} \]} \\
\text{digits ::= adigit \[ digits \]}
\end{align*}
\]

In the syntax specification, the symbol adigit is any of the values ct.widen(c) for c in the range '0' through '9', inclusive, and ct is a reference of type const type<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().

### moneypunct members

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>charT</td>
<td>decimal_point() const;</td>
</tr>
<tr>
<td>charT</td>
<td>thousands_sep() const;</td>
</tr>
<tr>
<td>string</td>
<td>grouping() const;</td>
</tr>
<tr>
<td>string_type</td>
<td>curr_symbol() const;</td>
</tr>
<tr>
<td>string_type</td>
<td>positive_sign() const;</td>
</tr>
<tr>
<td>string_type</td>
<td>negative_sign() const;</td>
</tr>
<tr>
<td>int</td>
<td>frac_digits() const;</td>
</tr>
<tr>
<td>pattern</td>
<td>pos_format() const;</td>
</tr>
<tr>
<td>pattern</td>
<td>neg_format() const;</td>
</tr>
</tbody>
</table>

An array of char, rather than an array of part, is specified for pattern::field purely for efficiency.
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

charT do_decimal_point() const;

Returns: The radix separator to use in case do_frac_digits() is greater than zero.\[252]\)

charT do_thousands_sep() const;

Returns: The digit group separator to use in case do_grouping() specifies a digit grouping pattern.\[253]\)

string do_grouping() const;

Returns: A pattern defined identically as, but not necessarily equal to, the result of numpunct&lt;charT&gt;: : do_grouping().\[254]\)

string_type do_curr_symbol() const;

Returns: A string to use as the currency identifier symbol.\[255]\)

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.

int do_frac_digits() const;

Returns: The number of digits after the decimal radix separator, if any.\[257]\)

pattern do_pos_format() const;
pattern do_neg_format() const;

Returns: The specializations required in Table 75 (22.1.1.1.1), namely moneypunct&lt;char&gt;, moneypunct&lt;wchar_t&gt;, moneypunct&lt;char,true&gt;, and moneypunct&lt;wchar_t,true&gt;, return an object of type pattern initialized to \{ symbol, sign, none, value \}.\[258]\)

22.2.6.4 Class template moneypunct_byname

namespace std {

    template &lt;class charT, bool Intl = false&gt;
    class moneypunct_byname : public moneypunct&lt;charT, Intl&gt; {
        public:
            typedef money_base: :pattern pattern;
            typedef basic_string&lt;charT&gt; string_type;

            \[252]\) In common U.S. locales this is \'.\'.
            \[253]\) In common U.S. locales this is \',\'.
            \[254]\) To specify grouping by 3s, the value is \"\003\" not \"3\".)
            \[255]\) For international specializations (second template parameter true) this is always typically four characters long, usually three letters and a space.
            \[256]\) This is usually the empty string.
            \[257]\) In common U.S. locales, this is 2.
            \[258]\) Note that the international symbol returned by do_curr_sym() usually contains a space, itself; for example, "USD ".

}
22.2.7 The message retrieval category

Class messages<
charT
> implements retrieval of strings from message catalogs.

22.2.7.1 Class template 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;
            string_type get(catalog c, int set, int msgid,
                const string_type& dfault) const;
            void close(catalog c) const;
            static locale::id id;
    
    protected:
        "messages();
        virtual catalog do_open(const basic_string<char>&, const locale&) const;
        virtual string_type do_get(catalog, int set, int msgid,
            const string_type& dfault) const;
        virtual void do_close(catalog) const;
    
    }
}

Values of type messages_base::catalog usable as arguments to members get and close can be obtained only by calling member open.

22.2.7.1.1 messages members

 catalog open(const basic_string<char>& name, const locale& loc) const;
22.2 Standard locale categories

Returns: do_open(name, loc).

string_type get(catalog cat, int set, int msgid,
    const string_type& dfault) const;

Returns: do_get(cat, set, msgid, dfault).

void close(catalog cat) const;

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;

Returns: A value that may be passed to get() to retrieve a message, from the message catalog identified by the string name according to an implementation-defined mapping. The result can be used until it is passed to close().

Returns a value less than 0 if no such catalog can be opened.

Remarks: The locale argument loc is used for character set code conversion when retrieving messages, if needed.

string_type do_get(catalog cat, int set, int msgid,
    const string_type& dfault) const;

Requires: cat shall be a catalog obtained from open() and not yet closed.

Returns: A message identified by arguments set,msgid, and dfault, according to an implementation-defined mapping. If no such message can be found, returns dfault.

void do_close(catalog cat) const;

Requires: cat shall be a catalog obtained from open() and not yet closed.

Effects: Releases unspecified resources associated with cat.

Remarks: The limit on such resources, if any, is implementation-defined.

22.2.7.2 Class template 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 refs = 0);
            explicit messages_byname(const string&, size_t refs = 0);
        protected:
    }
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 `locale::facet` a class containing a static member:

```cpp
static locale::id id;
```

[Note: The locale member function templates verify its type and storage class. —end note]

[Note: This paragraph is intentionally empty. —end note]

[Example: Traditional global localization is still easy:

```cpp
#include <iostream>
#include <locale>

int main(int argc, char** argv) {
    using namespace std;
    locale::global(locale("");
    // set the global locale
    // imbue it on all the std streams
    cin.imbue(locale());
    cout.imbue(locale());
    cerr.imbue(locale());
    wcin.imbue(locale());
    wcout.imbue(locale());
    wcerr.imbue(locale());
    return MyObject(argc, argv).doit();
}
```

—end example]

[Example: Greater flexibility is possible:

```cpp
#include <iostream>
#include <locale>

int main() {
    using namespace std;
    cin.imbue(locale(""));
    // the user’s preferred locale
    cout.imbue(locale::classic());
    double f;
    while (cin >> f) cout << f << endl;
    return (cin.fail() != 0);
}
```

In a European locale, with input 3.456,78, output is 3456.78. —end example]

This can be important even for simple programs, which may need to write a data file in a fixed format, regardless of a user’s preference.

[Example: Here is an example of the use of locales in a library interface.]

Draft
This example illustrates two architectural uses of class `locale`.

The first is as a default argument in `Date::asString()`, where the default is the global (presumably user-preferred) locale.

The second is in the operators `<<` and `>>`, where a locale “hitchhikes” on another object, in this case a stream, to the point where it is needed.

A locale object may be extended with a new facet simply by constructing it with an instance of a class derived from `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: ]

Draft
Localization library

## 22.2 Standard locale categories

```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) { }
```
22.3 Standard code conversion facets

The header `<codecvt>` provides code conversion facets for various character encodings.

**Synopsis**

```cpp
namespace std {
    enum codecvt_mode {
        consume_header = 4,
        generate_header = 2,
        little_endian = 1
    };

    template<class Elem, unsigned long Maxcode = 0x10ffff,
            codecvt_mode Mode = (codecvt_mode)0>
    class codecvt_utf8
        : public codecvt<Elem, char, mbstate_t> {
        // unspecified
    };

    template<class Elem, unsigned long Maxcode = 0x10ffff,
            codecvt_mode Mode = (codecvt_mode)0>
    class codecvt_utf16
        : public codecvt<Elem, char, mbstate_t> {
        // unspecified
    };

    template<class Elem, unsigned long Maxcode = 0x10ffff,
            codecvt_mode Mode = (codecvt_mode)0>
    class codecvt_utf8_utf16
        : public codecvt<Elem, char, mbstate_t> {
        // unspecified
    };
}
```

For each of the three code conversion facets `codecvt_utf8`, `codecvt_utf16`, and `codecvt_utf8_utf16`:

- `Elem` is the wide-character type, such as `wchar_t`, `char16_t`, or `char32_t`. 

Draft
— Maxcode is the largest wide-character code that the facet will read or write without reporting a conversion error.
— If (Mode & consume_header), the facet shall consume an initial header sequence, if present, when reading a multibyte sequence to determine the endianness of the subsequent multibyte sequence to be read.
— If (Mode & generate_header), the facet shall generate an initial header sequence when writing a multibyte sequence to advertise the endianness of the subsequent multibyte sequence to be written.
— If (Mode & little_endian), the facet shall generate a multibyte sequence in little-endian order, as opposed to the default big-endian order.

4 For the facet codecvt_utf8:
— The facet shall convert between UTF-8 multibyte sequences and UCS2 or UCS4 (depending on the size of E1em) within the program.
— Endianness shall not affect how multibyte sequences are read or written.
— The multibyte sequences may be written as either a text or a binary file.

5 For the facet codecvt_utf16:
— The facet shall convert between UTF-16 multibyte sequences and UCS2 or UCS4 (depending on the size of E1em) within the program.
— Multibyte sequences shall be read or written according to the Mode flag, as set out above.
— The multibyte sequences may be written only as a binary file. Attempting to write to a text file produces undefined behavior.

6 For the facet codecvt_utf8_utf16:
— The facet shall convert between UTF-8 multibyte sequences and UTF-16 (one or two 16-bit codes) within the program.
— Endianness shall not affect how multibyte sequences are read or written.
— The multibyte sequences may be written as either a text or a binary file.

See also: ISO/IEC 10646-1:1993.

22.4 C Library Locales

Header <clocale> (Table 86):

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td>LC_ALL, LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, LC_TIME, NULL</td>
</tr>
<tr>
<td>Struct:</td>
<td>lconv</td>
</tr>
<tr>
<td>Functions:</td>
<td>localeconv, setlocale</td>
</tr>
</tbody>
</table>

Draft
The contents are the same as the Standard C library header `<locale.h>`.

SEE ALSO: ISO C clause 7.4.
Chapter 23  Containers library

1 This clause describes components that C++ programs may use to organize collections of information.

2 The following subclauses describe container requirements, and components for sequence containers and associative containers, as summarized in Table 87:

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.1 Requirements</td>
<td></td>
</tr>
<tr>
<td>23.2 Sequence containers</td>
<td>&lt;array&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;deque&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;list&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;queue&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;stack&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;vector&gt;</td>
</tr>
<tr>
<td>23.3 Associative containers</td>
<td>&lt;map&gt;</td>
</tr>
<tr>
<td></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></td>
<td>&lt;unordered_set&gt;</td>
</tr>
</tbody>
</table>

23.1 Container requirements

1 Containers are objects that store other objects. They control allocation and deallocation of these objects through constructors, destructors, insert and erase operations.

2 All of the complexity requirements in this clause are stated solely in terms of the number of operations on the contained objects. [Example: the copy constructor of type vector <vector<int>> has linear complexity, even though the complexity of copying each contained vector<int> is itself linear. — end example ]

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.

3 Objects stored in these components shall be constructed using construct_element (20.6.9). For each operation that inserts an element of type T into a container (insert, push_back, push_front, emplace, etc.) with arguments args...
23.1 Container requirements

T shall be ConstructibleAsElement, as described in table 88. [Note: If the component is instantiated with a scoped allocator of type A (i.e., an allocator for which is_scoped_allocator<A>::value is true), then construct_element may pass an inner allocator argument to T's constructor. —end note]

In table 88, T denotes an object type, A denotes an allocator, I denotes an allocator of type A::inner_allocator_type (if any), and Args denotes a template parameter pack.

<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>CopyConstructible</td>
<td>Compile time</td>
<td></td>
</tr>
<tr>
<td>X::reference</td>
<td>const reference of T</td>
<td></td>
<td>Compile time</td>
<td></td>
</tr>
<tr>
<td>X::const_reference</td>
<td>const reference of T</td>
<td></td>
<td>Compile time</td>
<td></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::outer_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>
</tbody>
</table>

X u; post: u.size() == 0 constant
X(); X().size() == 0 constant
X(a); Requires: T is CopyConstructible, post: a == X(a).

Draft
<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>operational semantics</th>
<th>assertion/note pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X u(a);</code></td>
<td></td>
<td></td>
<td>Requires: <code>T</code> is <code>CopyConstructible</code></td>
<td>linear</td>
</tr>
<tr>
<td><code>X u = a;</code></td>
<td></td>
<td></td>
<td>post: <code>u == a</code></td>
<td></td>
</tr>
<tr>
<td><code>X u(rv);</code></td>
<td></td>
<td></td>
<td>Requires: <code>T</code> is <code>MoveConstructible</code></td>
<td><code>constant</code> (Note B)</td>
</tr>
<tr>
<td><code>X u = rv;</code></td>
<td></td>
<td></td>
<td>post: <code>u</code> shall be equal to the value that <code>rv</code> had before this construction</td>
<td></td>
</tr>
<tr>
<td><code>a = rv;</code></td>
<td><code>X&amp;</code></td>
<td>All existing elements of <code>a</code> are either move assigned or destroyed</td>
<td><code>a</code> shall be equal to the value that <code>rv</code> had before this construction</td>
<td><code>constant</code> (Note C)</td>
</tr>
<tr>
<td><code>(&amp;a)-&gt;~X();</code></td>
<td><code>void</code></td>
<td></td>
<td></td>
<td>linear</td>
</tr>
<tr>
<td><code>a.begin();</code></td>
<td><code>iterator</code>; <code>const_-iterator</code> for constant <code>a</code></td>
<td></td>
<td></td>
<td><code>constant</code></td>
</tr>
<tr>
<td><code>a.end();</code></td>
<td><code>iterator</code>; <code>const_-iterator</code> for constant <code>a</code></td>
<td></td>
<td></td>
<td><code>constant</code></td>
</tr>
<tr>
<td><code>a.cbegin();</code></td>
<td><code>const_-iterator</code></td>
<td><code>const_cast&lt;X&gt;</code></td>
<td><code>const&amp;&gt;(a).begin();</code></td>
<td><code>constant</code></td>
</tr>
<tr>
<td><code>a.cend();</code></td>
<td><code>const_-iterator</code></td>
<td><code>const_cast&lt;X&gt;</code></td>
<td><code>const&amp;&gt;(a).end();</code></td>
<td><code>constant</code></td>
</tr>
<tr>
<td><code>a == b</code></td>
<td>convertible to <code>bool</code></td>
<td><code>==</code> is an equivalence relation. <code>a.size() == b.size() &amp;&amp; equal(a.begin(), a.end(), b.begin())</code></td>
<td></td>
<td>linear</td>
</tr>
<tr>
<td><code>a != b</code></td>
<td>convertible to <code>bool</code></td>
<td>Equivalent to: <code>!(a == b)</code></td>
<td></td>
<td>linear</td>
</tr>
<tr>
<td><code>a.swap(b);</code></td>
<td><code>void</code></td>
<td></td>
<td><code>swap(a,b)</code></td>
<td><code>(Note A)</code></td>
</tr>
<tr>
<td><code>r = a</code></td>
<td><code>X&amp;</code></td>
<td>post: <code>r == a</code></td>
<td></td>
<td>linear</td>
</tr>
<tr>
<td><code>a.size()</code></td>
<td><code>size_type</code></td>
<td><code>a.end() − a.begin()</code></td>
<td></td>
<td><code>(Note A)</code></td>
</tr>
</tbody>
</table>
### 23.1 Container requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>operational semantics</th>
<th>assertion/note</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.max_size()</code></td>
<td><code>size_type</code></td>
<td><code>size()</code> of the largest possible container</td>
<td>(Note A)</td>
<td></td>
</tr>
<tr>
<td><code>a.empty()</code></td>
<td>convertible to <code>bool</code></td>
<td><code>a.size() == 0</code></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td><code>a &lt; b</code></td>
<td>convertible to <code>bool</code></td>
<td><code>lexicographical_compare(a.begin(), a.end(), b.begin(), b.end())</code></td>
<td><code>pre: &lt;</code> is defined for values of <code>T</code>. <code>&lt;</code> is a total ordering relationship.</td>
<td>linear</td>
</tr>
<tr>
<td><code>a &gt; b</code></td>
<td>convertible to <code>bool</code></td>
<td><code>b &lt; a</code></td>
<td></td>
<td>linear</td>
</tr>
<tr>
<td><code>a &lt;= b</code></td>
<td>convertible to <code>bool</code></td>
<td><code>!(a &gt; b)</code></td>
<td></td>
<td>linear</td>
</tr>
<tr>
<td><code>a &gt;= b</code></td>
<td>convertible to <code>bool</code></td>
<td><code>!(a &lt; b)</code></td>
<td></td>
<td>linear</td>
</tr>
</tbody>
</table>

Notes: the algorithms `swap()`, `equal()` and `lexicographical_compare()` are defined in clause 25. Those entries marked “(Note A)” should have constant complexity. Those entries marked “(Note B)” have constant complexity unless `allocator_propagate_never<X::allocator_type>::value` is true, in which case they have linear complexity. Those entries marked “(Note C)” have constant complexity if `a.get_allocator() == rv.get_allocator()` or if either `allocator_propagate_on_move_assignment<X::allocator_type>::value` is true or `allocator_propagate_on_copy_assignment<X::allocator_type>::value` is true and linear complexity otherwise.

6 The member function `size()` returns the number of elements in the container. Its semantics is defined by the rules of constructors, inserts, and erases.

7 `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();`.

8 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.

9 Copy and move constructors for all container types defined in this clause *copy* obtain an allocator argument from by calling `allocator_propagation_map<allocator_type>::select_for_copy_construction()` on 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 or until the allocator is replaced. The allocator may be replaced only via assignment or swap(). Allocator replacement is performed by calling allocator_propogation_map<allocator_type>::move_assign(), allocator_propogation_map<allocator_type>::copy_assign(), or allocator_propogation_map<allocator_type>::swap() within the implementation of the corresponding container operation. In all container types defined in this clause, the member get_allocator() returns a copy of the Allocator object used to construct the container, or to replace the allocator.\textsuperscript{(259)}

If the iterator type of a container belongs to the bidirectional or random access iterator categories (24.1), the container is called \textit{reversible} and satisfies the additional requirements in Table 90.

Table 90: Reversible container requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>assertion/note</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 time</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 time</td>
</tr>
<tr>
<td>a.rbegin()</td>
<td>reverse_iterator; const_reverse_iterator for constant a</td>
<td>reverse_iterator(end())</td>
<td>constant</td>
</tr>
<tr>
<td>a.rend()</td>
<td>reverse_iterator; const_reverse_iterator for constant a</td>
<td>reverse_iterator(begin())</td>
<td>constant</td>
</tr>
<tr>
<td>a.crbegin();</td>
<td>const_reverse_iterator</td>
<td>const_cast&lt;X const&amp;&gt;(a).rbegin();</td>
<td>constant</td>
</tr>
<tr>
<td>a.crend();</td>
<td>const_reverse_iterator</td>
<td>const_cast&lt;X const&amp;&gt;(a).rend();</td>
<td>constant</td>
</tr>
</tbody>
</table>

Unless otherwise specified (see 23.2.2.3 and 23.2.6.4) all container types defined in this clause meet the following additional requirements:

- if an exception is thrown by an \texttt{insert()} function while inserting a single element, that function has no effects.
- if an exception is thrown by a \texttt{push_back()} or \texttt{push_front()} function, that function has no effects.
- no \texttt{erase()}, \texttt{pop_back()} or \texttt{pop_front()} function throws an exception.
- no copy constructor or assignment operator of a returned iterator throws an exception.
- no \texttt{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).

\textsuperscript{(259)} 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

— no `swap()` function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped.

12 Unless otherwise specified (either explicitly or by defining a function in terms of other functions), invoking a container member function or passing a container as an argument to a library function shall not invalidate iterators to, or change the values of, objects within that container.

13 Objects passed to member functions of a container as rvalue references shall not be elements of that container. No diagnostic required.

14 All of the containers defined in this clause and in clause (21.3) except `array` meet the additional requirements of an allocator-aware container, as described in Table 91.

15 In Table 91, `X` denotes an allocator-aware container class with a `value_type` of `T` using allocator of type `A`, `u` denotes a variable, `t` denotes an lvalue or a const rvalue of type `X`, `rv` denotes a non-const rvalue of type `X`, `m` is a value of type `A`, and `Q` is an allocator type.

Table 91: Allocator-aware container requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>assertion/note</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>allocator_type A</code></td>
<td></td>
<td>Requires: <code>allocator_type::value_type</code> is the same as <code>X::value_type.</code></td>
<td>compile time</td>
</tr>
<tr>
<td><code>uses_allocator&lt;X, Q&gt;</code></td>
<td>derived from <code>true_type</code> or <code>false_type</code></td>
<td><code>true_type</code> if <code>Q</code> is convertible to <code>A</code></td>
<td>compile time</td>
</tr>
<tr>
<td><code>constructible_with_allocator_suffix&lt;X&gt;</code></td>
<td>derived from <code>true_type</code></td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td><code>get_allocator()</code></td>
<td><code>A</code></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td><code>X()</code></td>
<td></td>
<td>Requires: <code>A</code> is <code>DefaultConstructible</code></td>
<td>constant</td>
</tr>
<tr>
<td><code>X u;</code></td>
<td></td>
<td><code>DefaultConstructible</code></td>
<td>constant</td>
</tr>
<tr>
<td><code>X(m)</code></td>
<td><code>X u(m);</code></td>
<td><code>post: u.size() == 0, get_allocator() == A()</code></td>
<td>constant</td>
</tr>
<tr>
<td><code>X(t, m)</code></td>
<td><code>X u(t, m);</code></td>
<td><code>post: u.size() == 0, get_allocator() == m</code></td>
<td>constant</td>
</tr>
<tr>
<td><code>X(rv, m)</code></td>
<td></td>
<td>Requires: <code>ConstructibleAsElement&lt;A, T, T&amp;&amp;&gt;</code></td>
<td>constant if <code>m == rv.get_allocator()</code>, otherwise linear</td>
</tr>
</tbody>
</table>
23.1 Container requirements

1 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).

2 `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.

3 In Tables 92 and 93, `X` denotes a sequence container class, `a` denotes a value of `X` containing elements of type `T`. `A` denotes `X::allocator_type` if it exists and `std::allocator<T>` if it doesn’t. `i` and `j` denote iterators satisfying input iterator requirements and refer to elements implicitly convertible to `value_type`, `[i, j)` denotes a valid range, `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; `args` denotes a function parameter pack with the pattern `Args&&`.

4 The complexities of the expressions are sequence dependent.

## Table 92: Sequence container requirements (in addition to container)

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>assertion/note</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X u(rv, m);</code></td>
<td><code>post: u shall be equal to the value that rv had before this construction, get_allocator() == m</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 23.1 Container requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.insert(p,t)</td>
<td>iterator</td>
<td>Requires: T shall be CopyConstructible, ConstructibleAsElement\langle A, T, Args \rangle and T shall be CopyAssignable. Inserts a copy of t before p.</td>
</tr>
<tr>
<td>a.insert(p,rv)</td>
<td>iterator</td>
<td>Requires: ConstructibleAsElement\langle A, T, T&amp;&amp; \rangle and T shall be MoveAssignable. Inserts a copy of rv before p.</td>
</tr>
<tr>
<td>a.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.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. prec: i and j are not iterators into a. Inserts copies of elements in [i, j) before p.</td>
</tr>
<tr>
<td>a.erase(q)</td>
<td>iterator</td>
<td>Requires: T shall be MoveAssignable. Erases the element pointed to by q.</td>
</tr>
<tr>
<td>a.erase(q1,q2)</td>
<td>iterator</td>
<td>Requires: T shall be MoveAssignable. Erases the elements in the range [q1, q2).</td>
</tr>
<tr>
<td>a.clear()</td>
<td>void</td>
<td>erase(begin(), end()) post: size() == 0</td>
</tr>
<tr>
<td>a.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. prec: i, j are not iterators into a. Replaces elements in a with a copy of [i, j).</td>
</tr>
<tr>
<td>a.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>

5. iterator and const_iterator types for sequence containers shall be at least of the forward iterator category.

6. The iterator returned from a.insert(p,t) points to the copy of t inserted into a.

7. 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.

8. 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.

9. For every sequence container defined in this clause and in clause 21:

Draft
— If the constructor

```cpp
template <class InputIterator>
    X(InputIterator first, InputIterator last,
       const allocator_type& alloc = allocator_type())
```

is called with a type `InputIterator` that does not qualify as an input iterator, then the constructor will behave as if the overloaded constructor:

```cpp
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:

```cpp
template <class InputIterator> // such as insert()
    rt fx1(iterator p, InputIterator first, InputIterator last);
```

```cpp
template <class InputIterator> // such as append(), assign()
    rt fx2(InputIterator first, InputIterator last);
```

```cpp
template <class InputIterator> // such as replace()
    rt fx3(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:

```cpp
rt fx1(iterator, size_type, const value_type&);
```

```cpp
rt fx2(size_type, const value_type&);
```

```cpp
rt fx3(iterator, iterator, size_type, const value_type&);
```

were called instead, with the same arguments.

10 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`.

11 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.

12 Table 93 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.
23.1 Container requirements

Table 93: Optional sequence container operations

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>assertion/note pre/post-condition</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>
</tbody>
</table>
| a.back()          | reference; const_reference for constant a | { iterator tmp = a.end();
--tmp;
return *tmp; } | vector, list, deque, basic_string |
| a.push_front(args) | void                 | a.emplace(a.begin(),
std::forward<Args>(args)...)
Requires:
ConstructibleAsElement<A, T, Args> | list, deque |
| a.push_back(args)  | void                 | a.emplace(a.end(),
std::forward<Args>(args)...)
Requires:
ConstructibleAsElement<A, T, Args> | list, deque, vector |
| a.pop_front()     | void                 | a.erase(a.begin()) | list, deque |
| a.pop_back()      | void                 | { iterator tmp = a.end();
--tmp;
a.erase(tmp); } | vector, list, deque, basic_string |
| [n]               | reference; const_reference for constant a | *(a.begin() + n) | vector, deque, basic_string, match_results |
| at(n)             | reference; const_reference for constant a | *(a.begin() + n) | vector, deque |

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.

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. If the Compare type uses an allocator, then it conforms to the same rules as a container item; the container will construct the comparison object with the allocator appropriate to
the allocator-related traits of the Compare type and whether `is_scoped_allocator` is true for the container’s allocator type.

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 94, `X` denotes an associative container class, `a` denotes a value of `X` when `X` supports unique keys, `a_uniq` denotes a value of `X` when `X` supports multiple keys, `u` denotes an identifier, `r` denotes an lvalue or a const rvalue of type `X`, `rv` denotes a non-const rvalue of type `X`, `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`. `A` denotes the storage allocator used by `X`, if any, or `std::allocator<X::value_type>` otherwise, and `m` denotes an allocator of a type convertible to `A`.

### Table 94: Associative container requirements (in addition to container)

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>assertion/note</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><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_type&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 (<code>i.e. Key</code>) for <code>map</code> and <code>multimap</code>.</td>
<td>compile time</td>
</tr>
<tr>
<td><code>X(c)</code> <code>X a(c);</code></td>
<td><code>Requires:</code></td>
<td>constructs an empty container. Uses a <em>copy of</em> <code>c</code> as a comparison object.</td>
<td>constant</td>
</tr>
</tbody>
</table>

Draft
## 23.1 Container requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>assertion/note</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X()</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>X a;</td>
<td>ConstructibleAsElement&lt;A, key_compare, key_compare&gt;</td>
<td>Constructs an empty container. Uses Compare() as a comparison object</td>
<td></td>
</tr>
<tr>
<td>X(i,j,c)</td>
<td>ConstructibleAsElement&lt;A, key_compare, key_compare&gt;</td>
<td>Constructs an empty container and inserts elements from the range [i, j) into it; uses c as a comparison object.</td>
<td></td>
</tr>
<tr>
<td>X(i,j)</td>
<td>ConstructibleAsElement&lt;A, key_compare, key_compare&gt;</td>
<td>Same as above, but uses Compare() as a comparison object</td>
<td></td>
</tr>
<tr>
<td>a.key_compare()</td>
<td>X::key_compare</td>
<td>returns the comparison object out of which a was constructed.</td>
<td>constant</td>
</tr>
<tr>
<td>a.value_compare()</td>
<td>X::value_compare</td>
<td>returns an object of value_compare constructed out of the comparison object</td>
<td>constant</td>
</tr>
<tr>
<td>a_uniq_emplace(args)</td>
<td>pair&lt;iterator, bool&gt;</td>
<td>inserts a T object t constructed with std::forward&lt;Args&gt;(args) if and only if there is no element in the container with key equivalent to the key of t. The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of t.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>a_eq_emplace(args)</td>
<td>iterator</td>
<td>inserts a T object t constructed with std::forward&lt;Args&gt;(args) and returns the iterator pointing to the newly inserted element.</td>
<td>logarithmic</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. emplace(p, args)</td>
<td>iterator</td>
<td>equivalent to a.emplace( std::forward&lt;Args&gt;(args)...). Return value is an iterator pointing to the element with the key equivalent to the newly inserted element. The const_iterator p 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 p</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>
</tbody>
</table>

Draft
### Expression and Return Types

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return Type</th>
<th>Assertion/Note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.erase(k)</code></td>
<td><code>size_type</code></td>
<td>erases all elements in the container with key equivalent to <code>k</code>. Returns the number of erased elements.</td>
<td><code>log(size()) + count(k)</code></td>
</tr>
<tr>
<td><code>a.erase(q)</code></td>
<td><code>iterator</code></td>
<td>erases the element pointed to by <code>q</code>. Returns an iterator pointing to the element immediately following <code>q</code> prior to the element being erased. If no such element exists, returns <code>a.end()</code>.</td>
<td>amortized constant</td>
</tr>
<tr>
<td><code>a.erase(q1, q2)</code></td>
<td><code>iterator</code></td>
<td>erases all the elements in the range <code>[q1, q2)</code>. Returns <code>q2</code>.</td>
<td><code>log(size()) + N</code> where <code>N</code> is the distance from <code>q1</code> to <code>q2</code>.</td>
</tr>
<tr>
<td><code>a.clear()</code></td>
<td><code>void</code></td>
<td><code>erase(a.begin(), a.end())</code> post: <code>size() == 0</code></td>
<td>linear in <code>size()</code>.</td>
</tr>
<tr>
<td><code>a.find(k)</code></td>
<td><code>iterator; const_iterator for constant a</code></td>
<td>returns an iterator pointing to an element with the key equivalent to <code>k</code>, or <code>a.end()</code> if such an element is not found</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>a.count(k)</code></td>
<td><code>size_type</code></td>
<td>returns the number of elements with key equivalent to <code>k</code>.</td>
<td><code>log(size()) + count(k)</code></td>
</tr>
<tr>
<td><code>a.lower_bound(k)</code></td>
<td><code>iterator; const_iterator for constant a</code></td>
<td>returns an iterator pointing to the first element with key not less than <code>k</code>, or <code>a.end()</code> if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>a.upper_bound(k)</code></td>
<td><code>iterator; const_iterator for constant a</code></td>
<td>returns an iterator pointing to the first element with key greater than <code>k</code>, or <code>a.end()</code> if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>a.equal_range(k)</code></td>
<td><code>pair&lt;iterator, iterator&gt;; pair&lt;const_iterator, const_iterator&gt; for constant a</code></td>
<td>equivalent to <code>make_pair(a.lower_bound(k), a.upper_bound(k))</code>.</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,

```
value_comp(*j, *i) == false
```
For associative containers with unique keys the stronger condition holds,

\[ \text{value\_comp}(i, j) \neq \text{false}. \]

When an associative container is constructed by passing a comparison object the container shall not store a pointer or reference to the passed object, even if that object is passed by reference. When an associative container is copied, either through a copy constructor or an assignment operator, the target container shall then use the comparison object from the container being copied, as if that comparison object had been passed to the target container in its constructor.

### 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 95 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>
<thead>
<tr>
<th>unsupported expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a == b )</td>
</tr>
<tr>
<td>( a \neq b )</td>
</tr>
<tr>
<td>( a &lt; b )</td>
</tr>
<tr>
<td>( a &gt; b )</td>
</tr>
<tr>
<td>( a \leq b )</td>
</tr>
<tr>
<td>( a \geq b )</td>
</tr>
</tbody>
</table>

Each unordered associative container is parameterized by \( \text{Key} \), by a function object \( \text{Hash} \) that acts as a hash function for values of type \( \text{Key} \), and by a binary predicate \( \text{Pred} \) that induces an equivalence relation on values of type \( \text{Key} \). Additionally, `unordered_map` and `unordered_multimap` associate an arbitrary mapped type \( T \) with the \( \text{Key} \). If the \( \text{Hash} \) or the \( \text{Pred} \) type uses an allocator, it shall conform to the same rules as container items; the container will construct the \( \text{Hash} \) and \( \text{Pred} \) objects with the allocator appropriate to the the allocator-related traits of the \( \text{Hash} \) and \( \text{Pred} \) types and whether \( \text{is\_scoped\_allocator} \) is true for the container’s allocator type.

A hash function is a function object that takes a single argument of type \( \text{Key} \) and returns a value of type \( \text{std}::\text{size}\_\text{t} \).

Two values \( k1 \) and \( k2 \) of type \( \text{Key} \) are considered equal if the container’s equality function object returns \( \text{true} \) 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 \textit{unique keys} if it may contain at most one element for each key. Otherwise, it supports \textit{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<\text{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 96: X is an unordered associative container class, a is an object of type X, b is a possibly const object of type X, a_uniq is an object of type X when X supports unique keys, a_eq is an object of type X when X supports equivalent keys, i and j are input iterators that refer to `value_type`, `[i, j)` is a valid range, p and q2 are valid const iterators to a, q and q1 are valid dereferenceable const iterators to a, `[q1, q2)` is a valid range in a, t is a value of type `X::value_type`, k is a value of type `key_type`, hf is a possibly const value of type `hasher`, eq is a possibly const value of type `key_equal`, n is a value of type `size_type`, and z is a value of type `float`.

### Table 96: Unordered associative container requirements (in addition to container)

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>assertion/note pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::key_type</td>
<td>Key</td>
<td>Key shall be CopyAssignable and CopyConstructible</td>
<td>compile time</td>
</tr>
<tr>
<td>X::hasher</td>
<td>Hash</td>
<td>Hash shall be a unary function object type such that the expression <code>hf(k)</code> has type <code>std::size_t</code>.</td>
<td>compile time</td>
</tr>
<tr>
<td>X::key_equal</td>
<td>Pred</td>
<td>Pred shall be a binary predicate that takes two arguments of type Key. Pred is an equivalence relation.</td>
<td>compile time</td>
</tr>
<tr>
<td>X::local_iterator</td>
<td>An iterator type whose category, value type, difference type, and pointer and reference types are the same as X::iterator's.</td>
<td>A local_iterator object may be used to iterate through a single bucket, but may not be used to iterate across buckets.</td>
<td>compile time</td>
</tr>
<tr>
<td>X::const_local_iterator</td>
<td>An iterator type whose category, value type, difference type, and pointer and reference types are the same as X::const_iterator's.</td>
<td>A const_local_iterator object may be used to iterate through a single bucket, but may not be used to iterate across buckets.</td>
<td>compile time</td>
</tr>
<tr>
<td>X(n, hf, eq)</td>
<td>X</td>
<td>Constructs an empty container with at least n buckets, using hf as the hash function and eq as the key equality predicate.</td>
<td>O(n)</td>
</tr>
<tr>
<td>X a(n, hf, eq)</td>
<td>X</td>
<td>Constructs an empty container with at least n buckets, using hf as the hash function and key_equal() as the key equality predicate.</td>
<td>O(n)</td>
</tr>
<tr>
<td>X(n, hf)</td>
<td>X</td>
<td>Constructs an empty container with at least n buckets, using hf as the hash function and key_equal() as the key equality predicate.</td>
<td>O(n)</td>
</tr>
<tr>
<td>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(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.</td>
<td>$\mathcal{O}(n)$</td>
</tr>
<tr>
<td>X a(n)</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 $\mathcal{O}(N)$ (N is distance(i, j)), worst case $\mathcal{O}(N^2)$</td>
</tr>
<tr>
<td>X a(i, j, n, hf, eq)</td>
<td></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 $\mathcal{O}(N)$ (N is distance(i, j)), worst case $\mathcal{O}(N^2)$</td>
</tr>
<tr>
<td>X a(i, j, n, hf)</td>
<td></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 $\mathcal{O}(N)$ (N is distance(i, j)), worst case $\mathcal{O}(N^2)$</td>
</tr>
<tr>
<td>X a(i, j, n)</td>
<td></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 $\mathcal{O}(N)$ (N is distance(i, j)), worst case $\mathcal{O}(N^2)$</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><code>a_uniq. emplace(args)</code></td>
<td><code>pair&lt;iterator, bool&gt;</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>Average case $O(1)$, worst case $O(a_uniq._size())$.</td>
</tr>
<tr>
<td><code>a_eq. emplace(args)</code></td>
<td><code>iterator</code></td>
<td>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 $O(1)$, worst case $O(a_eq._size())$.</td>
</tr>
<tr>
<td><code>a. emplace(p, args)</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>Average case $O(1)$, worst case $O(a_size())$.</td>
</tr>
<tr>
<td><code>a_uniq.insert(t)</code></td>
<td><code>pair&lt;iterator, bool&gt;</code></td>
<td>Inserts <code>t</code> if and only if there is no element in the container with key equivalent to the key of <code>t</code>. The <code>bool</code> component of the returned pair indicates whether the insertion takes place, and the iterator component points to the element with key equivalent to the key of <code>t</code>.</td>
<td>Average case $O(1)$, worst case $O(a_uniq._size())$.</td>
</tr>
<tr>
<td><code>a_eq.insert(t)</code></td>
<td><code>iterator</code></td>
<td>Inserts <code>t</code>, and returns an iterator pointing to the newly inserted element.</td>
<td>Average case $O(1)$, worst case $O(a_eq._size())$.</td>
</tr>
<tr>
<td><code>a.insert(q, t)</code></td>
<td><code>iterator</code></td>
<td>Equivalent to <code>a.insert(t)</code>. Return value is an iterator pointing to the element with the key equivalent to that of <code>t</code>. The iterator <code>q</code> is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.</td>
<td>Average case $O(1)$, worst case $O(a_size())$.</td>
</tr>
<tr>
<td>expression</td>
<td>return type</td>
<td>assertion/note pre/post-condition</td>
<td>complexity</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>a.insert(i, j)</td>
<td>void</td>
<td>Pre: i and j are not iterators in a. Equivalent to a.insert(t) for each element in [i, j].</td>
<td>Average case $\mathcal{O}(N)$, where $N$ is distance(i, j). Worst case $\mathcal{O}(N \times a.size())$.</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 $\mathcal{O}(a.count(k))$. Worst case $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td>a.erase(q)</td>
<td>iterator</td>
<td>Erases the element pointed to by q. Return value is the iterator immediately following q prior to the erasure.</td>
<td>Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td>a.erase(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 $\mathcal{O}(a.size())$.</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;</td>
<td>Returns an iterator pointing to an element with key equivalent to k, or b.end() if no such element exists.</td>
<td>Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(b.size())$.</td>
</tr>
<tr>
<td></td>
<td>const_iterator for const b.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b.count(k)</td>
<td>size_type</td>
<td>Returns the number of elements with key equivalent to k.</td>
<td>Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(b.size())$.</td>
</tr>
<tr>
<td>b.equal_range(k)</td>
<td>pair&lt;iterator, iterator&gt;; pair&lt;const_iterator, const_iterator&gt; for const b.</td>
<td>Returns a range containing all elements with keys equivalent to k. Returns make_pair(b.end(), b.end()) if no such elements exist.</td>
<td>Average case $\mathcal{O}(b.count(k))$. Worst case $\mathcal{O}(b.size())$.</td>
</tr>
<tr>
<td>b.bucket_count()</td>
<td>size_type</td>
<td>Returns the number of buckets that b contains.</td>
<td>Constant</td>
</tr>
<tr>
<td>b.max_bucket_count()</td>
<td>size_type</td>
<td>Returns an upper bound on the number of buckets that b might ever contain.</td>
<td>Constant</td>
</tr>
<tr>
<td>b.bucket(k)</td>
<td>size_type</td>
<td>Returns the index of the bucket in which elements with keys equivalent to k would be found, if any such element existed. Post: the return value shall be in the range [0, b.bucket_count()).</td>
<td>Constant</td>
</tr>
</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><code>b.bucket_size(n)</code></td>
<td><code>size_type</code></td>
<td>Pre: <code>n</code> shall be in the range [0, b.bucket_count()). Returns the number of elements in the n\textsuperscript{th} bucket.</td>
<td>( \Theta(b.bucket_size(n)) )</td>
</tr>
<tr>
<td><code>b.begin(n)</code></td>
<td><code>local_iterator; const_local_iterator for const b.</code></td>
<td>Pre: <code>n</code> shall be in the range [0, b.bucket_count()). Note: ([\texttt{b.begin(n)}, \texttt{b.end(n)})) is a valid range containing all of the elements in the n\textsuperscript{th} bucket.</td>
<td>Constant</td>
</tr>
<tr>
<td><code>b.end(n)</code></td>
<td><code>local_iterator; const_local_iterator for const b.</code></td>
<td>Pre: <code>n</code> shall be in the range [0, b.bucket_count()).</td>
<td>Constant</td>
</tr>
<tr>
<td><code>b.load_factor()</code></td>
<td><code>float</code></td>
<td>Returns the average number of elements per bucket.</td>
<td>Constant</td>
</tr>
<tr>
<td><code>b.max_load_factor()</code></td>
<td><code>float</code></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><code>a.max_load_factor(z)</code></td>
<td><code>void</code></td>
<td>Pre: <code>z</code> shall be positive. Changes the container's maximum load load factor, using <code>z</code> as a hint.</td>
<td>Constant</td>
</tr>
<tr>
<td><code>a.rehash(n)</code></td>
<td><code>void</code></td>
<td>Post: ( a.bucket_count() &gt; \frac{a.size()}{a.max_load_factor() and a.bucket_count() \geq n} ). Average case linear in ( a.size() ), worst case quadratic.</td>
<td></td>
</tr>
</tbody>
</table>

10 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

11 The iterator types `iterator` and `const_iterator` of an unordered associative container are of at least the forward iterator category. For unordered associative containers where the key type and value type are the same, both `iterator` and `const_iterator` are `const iterators`.

12 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.

13 The insert members shall not affect the validity of iterators if \((N+n) < z \times B\), where \(N\) is the number of elements in the container prior to the insert operation, \(n\) is the number of elements inserted, \(B\) is the container's bucket count, and \(z\)
is the container’s maximum load factor.

23.1.3.1 Exception safety guarantees

For unordered associative containers, no clear() function throws an exception. No erase() function throws an exception unless that exception is thrown by the container’s Hash or Pred object (if any).

For unordered associative containers, if an exception is thrown by any operation other than the container’s hash function from within an insert() function inserting a single element, the insert() function has no effect.

For unordered associative containers, no 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).

For unordered associative containers, if an exception is thrown from within a rehash() function other than by the container’s hash function or comparison function, the rehash() function has no effect.

23.2 Sequence containers

Headers <array>, <deque>, <forward_list>, <list>, <queue>, <stack>, and <vector>.

Header <array> synopsis

```cpp	namespace std {
  template <class T, size_t N> struct array;
  template <class T, size_t N>
    bool operator==(const array<T,N>& x, const array<T,N>& y);
  template <class T, size_t N>
    bool operator!=(const array<T,N>& x, const array<T,N>& y);
  template <class T, size_t N>
    bool operator<(const array<T,N>& x, const array<T,N>& y);
  template <class T, size_t N>
    bool operator>(const array<T,N>& x, const array<T,N>& y);
  template <class T, size_t N>
    bool operator<=(const array<T,N>& x, const array<T,N>& y);
  template <class T, size_t N>
    bool operator>=(const array<T,N>& x, const array<T,N>& y);
  template <class T, size_t N>
    void swap(array<T,N>& x, array<T,N>& y);

  template <class T> class tuple_size;
  template <int I, class T> class tuple_element;
  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> >;
  template <int I, class T, size_t N>
    T& get(array<T, N>& x);
  template <int I, class T, size_t N>
    const T& get(const array<T, N>& x);
}
```

Header <deque> synopsis

Draft
namespace std {
  template <class T, class Allocator = allocator<T> > class deque;
  template <class T, class Allocator = allocator<T> >
    bool operator==(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    bool operator<(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    bool operator!=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    bool operator>(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    bool operator>=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    bool operator<=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    void swap(deque<T,Allocator>& x, deque<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    void swap(deque<T,Allocator>&& x, deque<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    void swap(deque<T,Allocator>& x, deque<T,Allocator>&& y);
}

Header `<forward_list>` synopsis

namespace std {
  template <class T, class Allocator = allocator<T> > class forward_list;
  template <class T, class Allocator = allocator<T> >
    bool operator==(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    bool operator<(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    bool operator!=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    bool operator>(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    bool operator>=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    bool operator<=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    void swap(forward_list<T,Allocator>&& x, forward_list<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>&& y);
}

Header `<list>` synopsis

namespace std {
  template <class T, class Allocator = allocator<T> > class list;
  template <class T, class Allocator = allocator<T> >
    bool operator==(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    bool operator<(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    bool operator!=(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    bool operator>(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    bool operator>=(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    bool operator<=(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    void swap(list<T,Allocator>& x, list<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    void swap(list<T,Allocator>&& x, list<T,Allocator>& y);
  template <class T, class Allocator = allocator<T> >
    void swap(list<T,Allocator>& x, list<T,Allocator>&& y);
}

Draft
template <class T, class Allocator>
bool operator!=(const list<T,Allocator>& x, const list<T,Allocator>& y);
template <class T, class Allocator>
bool operator> (const list<T,Allocator>& x, const list<T,Allocator>& y);
template <class T, class Allocator>
bool operator>=(const list<T,Allocator>& x, const list<T,Allocator>& y);
template <class T, class Allocator>
bool operator<=(const list<T,Allocator>& x, const list<T,Allocator>& y);
template <class T, class Allocator>
void swap(list<T,Allocator>& x, list<T,Allocator>& y);
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);
}

Header <queue> synopsis

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>,
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);
}

Draft
23.2 Sequence containers

**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);
    template <class Allocator> class vector<bool,Allocator>;
```
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

        // 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;
    };
}
```

Draft
23.2 Sequence containers

```
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);  
reference front();
const_reference front() const;
reference back();
const_reference back() const;

T * data();
const T * data() const;
```

[Note: The member variable `elems` is shown for exposition only, to emphasize that `array` is a class aggregate. The name `elems` is not part of `array`'s interface. — end note]

23.2.1.1 `array` constructors, copy, and assignment

The conditions for an aggregate (8.5.1) shall be met. Class `array` relies on the implicitly-declared special member 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

```
template <class T, size_t N> void swap(array<T,N>& x, array<T,N>& y);
```

Effects:
```
swap_ranges(x.begin(), x.end(), y.begin());
```

23.2.1.3 `array::size`

```
template <class T, size_t N> size_type array<T,N>::size();
```

Returns: N

23.2.1.4 `array::data`

```
T *data();
const T *data() const;
```

Returns: `elems`. 

Draft
23.2.1.5 Zero sized arrays

Array shall provide support for the special case \( N = 0 \).

In the case that \( N = 0 \), \begin{array} \end{array} == \end{array} == unique value. The return value of \texttt{data()} is unspecified.

The effect of calling \texttt{front()} or \texttt{back()} for a zero-sized array is implementation defined.

23.2.1.6 Tuple interface to class template \texttt{array}

\begin{verbatim}

tuple_size<

Return type: integral constant expression.

Value: \( N \)

tuple_element<I, array<T, N>> >::type

Requires: \( 0 \leq I < N \). The program is ill-formed if \( I \) is out of bounds.

Value: The type \( T \).

template <int I, class T, size_t N> T& get(array<T, N>& a);

Requires: \( 0 \leq I < N \). 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.

template <int I, class T, size_t N> const T& get(const array<T, N>& a);

Requires: \( 0 \leq I < N \). The program is ill-formed if \( I \) is out of bounds.

Returns: A const reference to the \( I \)th element of \( a \), where indexing is zero-based.
\end{verbatim}

23.2.2 Class template \texttt{deque}

A \texttt{deque} is a sequence container that, like a \texttt{vector} (23.2.6), supports random access iterators. In addition, it supports constant time insert and erase operations at the beginning or the end; insert and erase in the middle take linear time. That is, a \texttt{deque} is especially optimized for pushing and popping elements at the beginning and end. As with vectors, storage management is handled automatically.

A \texttt{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), and of an allocator-aware container (Table 91). Descriptions are provided here only for operations on \texttt{deque} that are not described in one of these tables or for operations where there is additional semantic information.

\begin{verbatim}

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

Draft
typedef implementation-defined const_iterator; // See 23.1
typedef implementation-defined size_type; // See 23.1
difference_type; // See 23.1
type T value_type;
type Alloc type Allo::pointer pointer;
type typename Alloc::const_pointer const_pointer;
type std::reverse_iterator<iterator> reverse_iterator;
type std::reverse_iterator<const_iterator> const_reverse_iterator;

// 23.2.2.1 construct/copy/destroy:
explicit deque(const Alloc& = Alloc());
explicit deque(size_type n);
d(e(size_type n, const T& value, const Alloc& = Alloc());
template <class InputIterator>
d(e(InputIterator first, InputIterator last, const Alloc& = Alloc());
d(e(const deque<T, Alloc>& x);
d(e(deque&&);
d(e(const Alloc&), const Alloc&);
d(e(deque&, const Alloc&);
d(e(deque, const Alloc&);

-`deq`();
deque<T, Alloc>& operator=(const deque<T, Alloc>& x);
deque<T, Alloc>& operator=(const deque<T, Alloc>&& x);
template <class InputIterator>
void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& t);
alloc_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, const 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:
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, size_type n, const T& x);
template <class InputIterator>
void insert (const_iterator position, InputIterator first, InputIterator last);

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);
template <class T, class Allocator>
bool operator<(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
template <class T, class Allocator>
bool operator!=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
template <class T, class Allocator>
bool operator>(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
template <class T, class Allocator>
bool operator>=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
template <class T, class Allocator>
bool operator<=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);

// specialized algorithms:
template <class T, class Allocator>
void swap(deque<T,Allocator>& x, deque<T,Allocator>& y);
template <class T, class Allocator>
void swap(deque<T,Allocator>&& x, deque<T,Allocator>&& y);
template <class T, class Allocator>

Draft
23.2 Sequence containers

void swap(deque<T, Allocator>& x, deque<T, Allocator>&& y);

```cpp
template <class T, class Alloc
struct constructible_with_allocator_suffix<deque<T, Alloc> >
  : true_type {};
}
```

23.2.2.1 deque constructors, copy, and assignment

```cpp
explicit deque(const Allocator& = Allocator());
```

*Effects:* Constructs an empty deque, using the specified allocator.

*Complexity:* Constant.

```cpp
explicit deque(size_type n);
```

*Effects:* Constructs a deque with n default constructed elements.

*Requires:* T shall be DefaultConstructible.

*Complexity:* Linear in n.

```cpp
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.

```cpp
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).

```cpp
template <class InputIterator>
void assign(InputIterator first, InputIterator last);
```

*Effects:*

```cpp
erase(begin(), end());
insert(begin(), first, last);
```

```cpp
void assign(size_type n, const T& t);
```

*Effects:*

```cpp
erase(begin(), end());
insert(begin(), n, t);
```
23.2.2.2 deque capacity

```cpp
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`.

```cpp
void resize(size_type sz, const T& c);
```

**Effects:**

```cpp
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

```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);
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 a constructor of `T`.

```cpp
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.
23.2 Sequence containers

**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

```cpp
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 forward_list

A **forward_list** is a container that supports forward iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Fast random access to list elements is not supported. [Note: It is intended that forward_list have zero space or time overhead relative to a hand-written C-style singly linked list. Features that would conflict with that goal have been omitted. —end note]

A **forward_list** satisfies all of the requirements of a container (table 89), except that the `size()` member function is not provided. Descriptions are provided here only for operations on **forward_list** that are not described in that table or for operations where there is additional semantic information.

```cpp
namespace std {
    template <class T, class Allocator = allocator<T> >
    class forward_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;

    # 23.2.3.1 construct/copy/destroy:
    explicit forward_list(const Allocator& = Allocator());
    explicit forward_list(size_type n);

    Draft
```
forward_list(size_type n, const T& value,  
    const Allocator& = Allocator());

template <class InputIterator>  
    forward_list(InputIterator first, InputIterator last, 
    const Allocator& = Allocator());

forward_list(const forward_list<T,Allocator>& x);  
forward_list(forward_list<T,Allocator>&& x);  
~forward_list();

forward_list<T,Allocator>& operator=(const forward_list<T,Allocator>& x);  
forward_list<T,Allocator>& operator=(forward_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;

// 23.2.3.2 iterators:
iterator before_begin();
const_iterator before_begin() const;
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

const_iterator cbegin();
const_iterator cbefore_begin();
const_iterator cend();

// capacity:
bool empty() const;
size_type max_size() const;

// 23.2.3.3 element access:
reference front();
const_reference front() const;

// 23.2.3.4 modifiers:
template <class... Args> void push_front(Args&&... args);
void pop_front();

template <class... Args> iterator emplace_after(const_iterator position, Args&&... args);
iterator insert_after(const_iterator position, const T& x);  
iterator insert_after(const_iterator position, T&& x);

void insert_after(const_iterator position, size_type n, const T& x);
template <class InputIterator>  
    void insert_after(const_iterator position, InputIterator first, InputIterator last);

iterator erase_after(const_iterator position);  
iterator erase_after(const_iterator position, iterator last);
void swap(forward_list<T,Allocator>&&);
void resize(size_type sz);
void resize(size_type sz, value_type c);
void clear();

// 23.2.3.5 forward_list operations:
void splice_after(const_iterator position, forward_list<T,Allocator>&& x);
void splice_after(const_iterator position, forward_list<T,Allocator>&& x,
    const_iterator i);
void splice_after(const_iterator position, forward_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(forward_list<T,Allocator>&& x);
template <class Compare> void merge(forward_list<T,Allocator>&& x, Compare comp);

void sort();
template <class Compare> void sort(Compare comp);

void reverse();
};

// Comparison operators
template <class T, class Allocator>
bool operator==(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
template <class T, class Allocator>
bool operator<(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
template <class T, class Allocator>
bool operator!=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
template <class T, class Allocator>
bool operator>(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
template <class T, class Allocator>
bool operator>=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
template <class T, class Allocator>
bool operator<=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);

// 23.2.3.6 specialized algorithms:
template <class T, class Allocator>
void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>& y);
template <class T, class Allocator>
void swap(forward_list<T,Allocator>&& x, forward_list<T,Allocator>& y);
template <class T, class Allocator>
void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>&& y);
}
23.2.3.1  forward_list constructors, copy, assignment

explicit forward_list(const Allocator& = Allocator());
   Effects: Constructs an empty forward_list object using the specified allocator.
   Complexity: Constant.

explicit forward_list(size_type n);
   Effects: Constructs a forward_list object with n default constructed elements.
   Requires: T shall be DefaultConstructible.
   Complexity: Linear in n.

forward_list(size_type n, const T& value, const Allocator& = Allocator());
   Effects: Constructs a forward_list object with n copies of value using the specified allocator.
   Requires: T shall be CopyConstructible.
   Complexity: Linear in n.

template <class InputIterator>
forward_list(InputIterator first, InputIterator last, const Allocator& = Allocator());
   Effects: Constructs a forward_list object equal to the range [first, last).
   Complexity: Linear in distance(first, last).

template <class InputIterator>
void assign(InputIterator first, InputIterator last);
   Effects: clear(); insert_after(before_begin(), first, last);

void assign(size_type n, const T& t);
   Effects: clear(); insert_after(before_begin(), n, t);

23.2.3.2  forward_list iterators

iterator before_begin();
const_iterator before_begin() const;
const_iterator cbefore_begin() const;
   Returns: A non-dereferenceable iterator that, when incremented, is equal to the iterator returned by begin().

23.2.3.3  forward_list element access

reference front();
const_reference front() const;
23.2  Sequence containers

[forwardlist.modifiers]

23.2.3.4  forward_list modifiers

Returns: *begin()

None of the overloads of insert_after shall affect the validity of iterators and reference, and erase_after shall invalidate only the iterators and references to the erased elements. If an exception is thrown during insert_after there shall be no effect. Insertion of \( n \) elements into a forward_list is linear in \( n \), and the number of calls to the copy or move constructor of \( T \) is exactly equal to \( n \). Erasing \( n \) elements from a forward_list is linear time in \( n \) and the number of calls to the destructor of type \( T \) is exactly equal to \( n \).

```cpp
template <class... Args> void push_front(Args&&... args);
```

Effects: Inserts an object of type value_type constructed with value_type(std::forward<Args>(args)...)
at the beginning of the list.

```cpp
void pop_front();
```

Effects: Erases after(before_begin())

```cpp
iterator insert_after(const_iterator position, const T& x);
iterator insert_after(const_iterator position, T&& x);
```

Requires: position is dereferenceable or equal to before_begin().

Effects: Inserts a copy of \( x \) after position.

Returns: An iterator pointing to the copy of \( x \).

```cpp
void insert_after(const_iterator position, size_type n, const T& x);
```

Requires: position is dereferenceable or equal to before_begin().

Effects: Inserts \( n \) copies of \( x \) after position.

```cpp
template <class InputIterator>
void insert_after(const_iterator position, InputIterator first, InputIterator last);
```

Requires: position is dereferenceable or equal to before_begin().first and last are not iterators in *this.

Effects: Inserts copies of elements in [first,last) after position.

```cpp
template <class... Args>
iterator emplace_after(const_iterator position, Args&&... args);
```

Requires: position is dereferenceable or equal to before_begin().

Effects: Inserts an object of type value_type constructed with value_type(std::forward<Args>(args)...)
after position.

```cpp
iterator erase_after(const_iterator position);
```

Requires: The iterator following position is dereferenceable.

Effects: Erases the element pointed to by the iterator following position.
Returns: An iterator pointing to the element following the one that was erased, or end() if no such element exists.

```
iterator erase_after(const_iterator position, iterator last);
```

Requires: All iterators in the range [position, last) are dereferenceable.

Effects: Erases the elements in the range [position, last).

Returns: last

```
void resize(size_type sz);
void resize(size_type sz, value_type c);
```

Effects: If sz < distance(begin(), end()), erases the last distance(begin(), end()) - sz elements from the list. Otherwise, inserts sz - distance(begin(), end()) elements at the end of the list. For the first signature the inserted elements are default constructed, and for the second signature they are copies of c.

```
void clear();
```

Effects: Erases all elements in the range [begin(), end()).

### 23.2.3.5 forward_list operations

```
void splice_after(const_iterator position, forward_list<T,Allocator>&& x);
```

Requires: position is dereferenceable or equal to before_begin(). &x != this.

Effects: If position is dereferenceable or equal to before_begin(). &x != this.

Throws: Nothing.

Complexity: $O(1)$

```
void splice_after(const_iterator position, forward_list<T,Allocator>&& x, const_iterator i);
```

Requires: position is dereferenceable or equal to before_begin(). The iterator following i is a dereferenceable iterator in x.

Effects: Inserts the element following i into *this, following position, and removes it from x. Pointers and references to the moved elements of x now refer to those same elements but as members of *this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this, not into x.

Throws: Nothing.

Complexity: $O(1)$

```
void splice_after(const_iterator position, forward_list<T,Allocator>&& x, const_iterator first, const_iterator last);
```

Draft
23.2 Sequence containers

Requires: position is dereferenceable or equal to before_begin(). (first, last) is a valid range in x, and all iterators in the range (first, last) are dereferenceable. position is not an iterator in the range (first, last).

Effects: Inserts elements in the range (first, last) after position and removes the elements from x. Pointers and references to the moved elements of x now refer to those same elements but as members of *this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this, not into x.

void remove(const T& value);

Effects: Erases all the elements in the list referred by a list iterator i for which the following conditions hold: *i == value for remove(), pred(*i) is true for remove_if(). This operation shall be stable: the relative order of the elements that are not removed is the same as their relative order in the original list.

Throws: Nothing unless an exception is thrown by the equality comparison or the predicate.

Complexity: Exactly distance(begin(), end()) applications of the corresponding predicate.

void unique();

Effects: Eliminates all but the first element from every consecutive group of equal elements referred to by the iterator i in the range [first + 1, last) for which *i == *(i-1) (for the version with no arguments) or pred(*i, *(i - 1)) (for the version with a predicate argument) holds.

Throws: Nothing unless an exception is thrown by the equality comparison or the predicate.

Complexity: If the range [first, last) is not empty, exactly (last - first) - 1 applications of the corresponding predicate, otherwise no applications of the predicate.

void merge(forward_list<T, Allocator>&& x);

Effects: Merges x into *this. This operation shall be stable: for equivalent elements in the two lists, the elements from *this shall always precede the elements from x. x is empty after the merge. If an exception is thrown other than by a comparison there are no effects.

Complexity: At most size() + x.size() - 1 comparisons.

void sort();

Effects: Sorts the list according to the operator< or the comp function object. This operation shall be stable: the relative order of the equivalent elements is preserved. If an exception is thrown the order of the elements in *this is unspecified.

Complexity: Approximately N log N comparisons, where N is distance(begin(), end()).
void reverse();

Effects: Reverses the order of the elements in the list.

Throws: Nothing.

Complexity: Linear time.

### 23.2.3.6 forward_list specialized algorithms

```cpp
template <class T, class Allocator>
void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>& y);
template <class T, class Allocator>
void swap(forward_list<T,Allocator>&& x, forward_list<T,Allocator>& y);
template <class T, class Allocator>
void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>&& y);
```

Effects: \texttt{x.swap(y)}

### 23.2.4 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 (23.2.6) 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), and of an allocator-aware container (Table 91). The exceptions are the \texttt{operator[]} and \texttt{at} member functions, which are not provided.\footnote{These member functions are only provided by containers whose iterators are random access iterators.}

Descriptions are provided here only for operations on \texttt{list} that are not described in one of these tables or for operations where there is additional semantic information.

```cpp
namespace std {
    template <class T, class Allocator = allocator<T> >
    class list {
    public:
        // types:
        typedef typename Allocator::reference reference;
        typedef typename Allocator::const_reference const_reference; // See 23.1
        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<T> reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    }
} // namespace std
```
// 23.2.4.1 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(const list&, const Allocator&);
list(list&&, const Allocator&);
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();
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;

// 23.2.4.2 capacity:
bool empty() const;
size_type size() const;
size_type max_size() const;
void resize(size_type sz);
void resize(size_type sz, const T& c);

// element access:
reference front();
const_reference front() const;
reference back();
const_reference back() const;

// 23.2.4.3 modifiers:
template <class... Args> void push_front(Args&&... args);
void pop_front();
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);
template <class InputIterator>
iterator 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();

// 23.2.4.4 list operations:
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);

template <class T, class Alloc>
struct constructible_with_allocator_suffix<list<T, Alloc> >
  : true_type { };
void assign(size_type n, const T& t);

Effects: Replaces the contents of the list with n copies of t.

23.2.4.2 list capacity

void resize(size_type sz);

Effects: If sz < size(), equivalent to list<T>::iterator it = begin(); advance(it, sz); erase(it, 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()) {
      iterator i = begin();
      advance(i, sz);
      erase(i, end());
   } else
      // do nothing

Requires: T shall be CopyConstructible.

23.2.4.3 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);
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 a constructor of T. Insertion of multiple elements into a list is linear in the number of elements inserted, and the number of calls to the copy constructor or move constructor of T is exactly equal to the number of elements inserted.


23.2 Sequence containers

Containers library  724

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.

23.2.4.4 list operations

Since lists allow fast insertion and erasing from the middle of a list, certain operations are provided specifically for them.261)

list provides three splice operations that destructively move elements from one list to another. The behavior of splice operations is undefined if get_allocator() != x.get_allocator().

void splice(const_iterator position, list<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.

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.

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.

261) As specified in 20.1.2, the requirements in this clause apply only to lists whose allocators compare equal.
Requires: \([\text{first, last})\) is a valid range in \(x\). The result is undefined if \(\text{position}\) is an iterator in the range \([\text{first, last})\). Pointers and references to the moved elements of \(x\) now refer to those same elements but as members of \(*\text{this}\). Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into \(*\text{this}\), not into \(x\).

Throws: Nothing

Complexity: Constant time if \&\(x == \text{this}\); otherwise, linear time.

```cpp
void remove(const T& value);
```

Effects: Erases all the elements in the list referred by a list iterator \(i\) for which the following conditions hold: \(*i == value, \text{pred}(*i) != false.\)

Throws: Nothing unless an exception is thrown by \(*i == value\) or \(\text{pred}(*i) != false.\)

Remarks: Stable.

Complexity: Exactly \(\text{size()}\) applications of the corresponding predicate.

```cpp
void unique();
```

```cpp
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 \text{unique} with no arguments) or \(\text{pred}(*i, *(i - 1))\) (for the version of \text{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 \([\text{first, last})\) 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);
```

```cpp
template <class Compare> void merge(list<T, Allocator>&& x, Compare comp);
```

Requires: \(\text{comp}\) 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 \(\text{comp}\); that is, for every iterator \(i\), in the range other than the first, the condition \(\text{comp}(*i, *(i - 1))\) 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 \(\text{size()} + x. \text{size()} - 1\) applications of \(\text{comp}\) if \(\&\!x == \text{this}\); otherwise, no applications of \(\text{comp}\) 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.
23.2  Sequence containers

Complexity: Linear time.

void sort();
template <class Compare> void sort(Compare comp);

Requires: operator< (for the first version) or comp (for the second version) shall define 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 $N = \text{size()}$.

23.2.4.5  list 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);

Effects:
  x.swap(y);

23.2.5  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. If the container takes an allocator, then a compatible allocator may be passed in to the adaptor’s constructor. Otherwise, normal copy or move construction is used for the container argument. [Note: it is not necessary for an implementation to distinguish between the one-argument constructor that takes a Container and the one-argument constructor that takes an allocator_type. Both forms use their argument to construct an instance of the container. —end note]

23.2.5.1  Class template queue

Any sequence container supporting operations front(), back(), push_back() and pop_front() can be used to instantiate queue. In particular, list (23.2.4) and deque (23.2.2) can be used.

23.2.5.1.1  queue definition

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)) {}  
  template <class Alloc> explicit queue(const Alloc&);
  template <class Alloc> queue(const Container&, const Alloc&);
  template <class Alloc> queue(Container&, const Alloc&);
  template <class Alloc> queue(queue&&, const Alloc&);
  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); }


};

template <class T, class Container>
bool operator==(const queue<T, Container>& x, const queue<T, Container>& y);

template <class T, class Container>
bool operator<(const queue<T, Container>& x, const queue<T, Container>& y);

template <class T, class Container>
bool operator!=(const queue<T, Container>& x, const queue<T, Container>& y);

template <class T, class Container>
bool operator>(const queue<T, Container>& x, const queue<T, Container>& y);

template <class T, class Container>
bool operator>=(const queue<T, Container>& x, const queue<T, Container>& y);

template <class T, class Container>
bool operator<=(const queue<T, Container>& x, const queue<T, Container>& y);

template <class T, class Container>
void swap(queue<T, Container>& x, queue<T, Container>& y);

template <class T, class Container>
void swap(queue<T, Container>&& x, queue<T, Container>&& y);

template <class T, class Container, class Alloc>
struct uses_allocator<queue<T, Container>, Alloc> :
  uses_allocator<Container, Alloc>::type {};

Draft
23.2 Sequence containers

23.2.5.1.2 queue operators

```cpp
template <class T, class Container>
bool operator==(const queue<T, Container>& x,
               const queue<T, Container>& y);

1 Returns: x.c == y.c.

template <class T, class Container>
bool operator!=(const queue<T, Container>& x,
               const queue<T, Container>& y);

2 Returns: x.c != y.c.

template <class T, class Container>
bool operator<( const queue<T, Container>& x,
               const queue<T, Container>& y);

3 Returns: x.c < y.c.

template <class T, class Container>
bool operator<=(const queue<T, Container>& x,
               const queue<T, Container>& y);

4 Returns: x.c <= y.c.

template <class T, class Container>
bool operator>( const queue<T, Container>& x,
               const queue<T, Container>& y);

5 Returns: x.c > y.c.

template <class T, class Container>
bool operator>=(const queue<T, Container>& x,
               const queue<T, Container>& y);

6 Returns: x.c >= y.c.
```

23.2.5.1.3 queue specialized algorithms

```cpp
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).

23.2.5.2 Class template priority_queue

Any sequence container with random access iterator and supporting operations front(), push_back() and pop_back() can be used to instantiate priority_queue. In particular, vector (23.2.6) 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,
                        const Compare& x, const Container&);
        template <class InputIterator>
        priority_queue(InputIterator first, InputIterator last,
                        const Compare& x = Compare(), Container&& = 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&&);
    }
23.2 Sequence containers

Containers library 730

};

// no equality is provided
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);

template <class T, class Container, class Compare, class Alloc>
struct uses_allocator<priority_queue<T, Container, Compare>, Alloc>
  : uses_allocator<Container, Alloc>::type { };

template <class T, class Container, class Compare>
struct constructible_with_allocator_suffix<priority_queue<T, Container, Compare>>
  \added{ true_type \{ \};}

23.2.5.2.1 priority_queue constructors

priority_queue(const Compare& x,
  const Container& y);
explicit priority_queue(const Compare& x = Compare(),
  Container&& y = Container());

1 Requires: x shall define a strict weak ordering (25.3).
2 Effects: Initializes comp with x and c with y (copy constructing or move constructing as appropriate); calls make_heap(c.begin(), c.end(), comp).

template <class InputIterator>
priority_queue(InputIterator first, InputIterator last,
  const Compare& x,
  const Container& y);

template <class InputIterator>
priority_queue(InputIterator first, InputIterator last,
  const Compare& x = Compare(),
  Container&& y = Container());

3 Requires: x shall define a strict weak ordering (25.3).
4 Effects: Initializes comp with x and c with y (copy constructing or move constructing as appropriate); calls c.insert(c.end(), first, last); and finally calls make_heap(c.begin(), c.end(), comp).

23.2.5.2.2 priority_queue members

void push(const value_type& x);
1 Effects:
   c.push_back(x);
   push_heap(c.begin(), c.end(), comp);
void push(value_type&& x);

Effects:
    c.push_back(std::move(x));
    push_heap(c.begin(), c.end(), comp);

void pop();

Effects:
    pop_heap(c.begin(), c.end(), comp);
    c.pop_back();

23.2.5.2.3 priority_queue specialized algorithms

[ priqueue.special ]

template < class T, class Container, Compare>
    void swap(priority_queue<T, Container, Compare>& x, priority_queue<T, Container, Compare>& y);

Effects: x.swap(y).

23.2.5.3 Class template stack

[ stack ]

Any sequence container supporting operations back(), push_back() and pop_back() can be used to instantiate stack. In particular, vector (23.2.6), list (23.2.4) and deque (23.2.2) can be used.

23.2.5.3.1 stack definition

[ stack.defn ]

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;

            Container c;

        public:
            explicit stack(const Container&);
            explicit stack(Container&& = Container());
            template < class Alloc> explicit stack(const Alloc&);
            template < class Alloc> stack(const Container&, const Alloc&);
            template < class Alloc> stack(Container&&, const Alloc&);

    };
template <class Alloc> stack(stack&&, const Alloc&); 

bool empty() const { return c.empty(); } 
size_type size() const { return c.size(); } 
reference top() { return c.back(); } 
const_reference top() const { return c.back(); } 
void push(const value_type& x) { c.push_back(x); } 
void push(value_type&& x) { c.push_back(std::move(x)); } 
void pop() { c.pop_back(); } 
void swap(stack&& s) { c.swap(s.c); } 

};

template <class T, class Container>
bool operator==(const stack<T, Container>& x, const stack<T, Container>& y);

template <class T, class Container>
bool operator!=(const stack<T, Container>& x, const stack<T, Container>& y);

template <class T, class Container, class Alloc>
struct uses_allocator<stack<T, Container>, Alloc> : uses_allocator<Container, Alloc>::type {};

template <class T, class Container>
struct constructible_with_allocator_suffix<stack<T, Container>> : true_type {};

\subsection{stack operators} \label{stack.ops}

template <class T, class Container>
bool operator==(const stack<T, Container>& x, const stack<T, Container>& y);

\begin{verbatim}
Returns: x.c == y.c.
\end{verbatim}

template <class T, class Container>
bool operator!=(const stack<T, Container>& x, const stack<T, Container>& y);

\begin{verbatim}
Returns: x.c != y.c.
\end{verbatim}
23.2 Sequence containers

23.2.5.3.3 stack specialized algorithms

```cpp
template <class T, class Container>
void swap(stack<T, Container>& x, stack<T, Container>& y);
```

Effects: \(x.\text{swap}(y)\).

23.2.6 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 \(0 \leq n < v.\text{size}()\).

A vector 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 optional sequence container requirements (23.1.1), and of an allocator-aware container (Table 91). The exceptions are 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.

```cpp
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;

            // 23.2.6.1 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(const vector<T,Allocator>&, const Allocator&);
            vector(vector&&, const Allocator&);
            ~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;
    }
}
```

Draft
const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// 23.2.6.2 capacity:
size_type size() const;
size_type max_size() const;
void resize(size_type sz);
void resize(size_type sz, T const& 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 front();
const_reference front() const;
reference back();
const_reference back() const;

// 23.2.6.3 data access
pointer data();
const_pointer data() const;

// 23.2.6.4 modifiers:
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, 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 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);
template <class T, class Allocator>
bool operator<(const vector<T,Allocator>& x, const vector<T,Allocator>& y);
template <class T, class Allocator>
23.2 Sequence containers

Containers library 736

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);

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);

temlate <class T, class Alloc>
struct constructible_with_allocator_suffix<vector<T, Alloc>>
    : true_type {};
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:**
- `erase(begin(), end());`
- `insert(begin(), first, last);`

```cpp
void assign(size_type n, const T& t);
```

**Effects:**
- `erase(begin(), end());`
- `insert(begin(), n, t);`

### 23.2.6.2 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 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 reserve(), capacity() is greater or equal to the argument of reserve if reallocation happens; and equal to the previous value of capacity() otherwise. Reallocation happens at this point if and only if the current capacity is less than the argument of reserve(). If an exception is thrown, there are no effects.

**Complexity:** It does not change the size of the sequence and takes at most linear time in the size of the sequence.

**Throws:** length_error if n > max_size().

**Remarks:** Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. It is guaranteed that no reallocation takes place during insertions that happen after a call to reserve() until the time when an insertion would make the size of the vector greater than the value of capacity().

```cpp
void swap(vector<T,Allocator>&& x);
```

**Effects:** Exchanges the contents and capacity() of *this with that of x.

**Complexity:** Constant time.

```cpp
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.

---

262) reserve() uses Allocator::allocate() which may throw an appropriate exception.
23.2 Sequence containers

Containers library

void resize(size_type sz, const T& c);

Effects:

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.

23.2.6.3 vector data

pointer data();
const_pointer data() const;

Returns: A pointer such that [data(), data() + size()) is a valid range. For a non-empty vector, data() == &front().

Complexity: Constant time.

Throws: Nothing.

23.2.6.4 vector 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_back(Args&&... args);
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.

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.

Draft
Throws: Nothing unless an exception is thrown by the copy constructor or assignment operator of T.

### 23.2.6.5 `vector` specialized algorithms

```cpp
template <class T, class Allocator>
void swap(vector<T,Allocator>& x, vector<T,Allocator>& y);
```

```cpp
template <class T, class Allocator>
void swap(vector<T,Allocator>&& x, vector<T,Allocator>& y);
```

```cpp
template <class T, class Allocator>
void swap(vector<T,Allocator>& x, vector<T,Allocator>&& y);
```

Effects:

```cpp
x.swap(y);
```

### 23.2.7 Class `vector<bool>`

To optimize space allocation, a specialization of vector for `bool` elements is provided:

```cpp
namespace std {
    template <class Allocator> class vector<bool, Allocator> { 
        public:
            // types:
            typedef bool const_reference;
            typedef implementation-defined iterator;     // See 23.1
            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(const vector&, const Allocator&);

vector(vector&, const Allocator&);

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);
    const_reference operator[](size_type n) const;
    const_reference at(size_type n) const;
    reference at(size_type n);
    reference front();
    const_reference front() const;
    reference back();
    const_reference back() const;
23.3 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;
    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>
```
23.3 Associative containers

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>
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
multimap<Key,T,Compare,Allocator>&& y);
}

**Header <set> synopsis**

amespace 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>
    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>
    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);
}
23.3 Associative containers

23.3.1 Class template map

1. A map is an associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of values of another type T based on the keys. The map class supports bidirectional iterators.

2. A map satisfies all of the requirements of a container, and of a reversible container (23.1), and of an associative container (23.1.2), and of an allocator-aware container (Table 91). A map also provides most operations described in (23.1.2) for unique keys. This means that a map supports the a_uniq operations in (23.1.2) but not the a_eq operations. For a map<Key,T> the key_type is Key and the value_type is pair<const Key,T>. Descriptions are provided here only for operations on map that are not described in one of those tables or for operations where there is additional semantic information.

namespace std {
    template <class Key, class T, class Compare = less<Key>,
              class Allocator = allocator<pair<const Key, T> > >
    class map {
        public:
            // types:
            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 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 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;
        }
    }

    template <class Key, class Compare, class Allocator>
    bool operator>(const multiset<Key,Compare,Allocator>& x,
                   const multiset<Key,Compare,Allocator>& y);

    template <class Key, class Compare, class Allocator>
    bool operator>=(const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);

    template <class Key, class Compare, class Allocator>
    bool operator<=(const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);

    template <class Key, class Compare, class Allocator>
    void swap(multiset<Key,Compare,Allocator>& x,
              multiset<Key,Compare,Allocator>& y);

    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);
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:

Draft
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:
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);
template <class P> pair<iterator, bool> insert(P&& x);
iterator insert(const_iterator position, const value_type& x);

// observers:
key_compare key_comp() const;
value_compare value_comp() const;

// 23.3.1.4 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 map<Key,T,Compare,Allocator>& x,
               const map<Key,T,Compare,Allocator>& y);

// specialized algorithms:

// 23.3.1.1 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].
23.3 Associative containers

Complexity: Linear in \(N\) if the range \([\text{first}, \text{last})\) is already sorted using \(\text{comp}\) and otherwise \(N \log N\), where \(N\) is \(\text{last} - \text{first}\).

23.3.1.2 map element access

\[\text{T& operator[] (const key_type& x);}\]

Effects: If there is no key equivalent to \(x\) in the map, inserts \(\text{value_type}(x, \text{T}())\) into the map.

Requires: \(\text{key_type}\) shall be \text{CopyConstructible} and \(\text{mapped_type}\) shall be \text{DefaultConstructible}.

Returns: A reference to the \(\text{mapped_type}\) corresponding to \(x\) in \(*\text{this}\).

Complexity: logarithmic.

\[\text{T& operator[] (const key_type&& x);}\]

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.

Requires: \(\text{mapped_type}\) shall be \text{DefaultConstructible}.

Returns: A reference to the \(\text{mapped_type}\) corresponding to \(x\) in \(*\text{this}\).

Complexity: logarithmic.

\[\text{T& at(const key_type& x);}\]

\[\text{const T& at(const key_type& x) const;}\]

Returns: A reference to the element whose key is equivalent to \(x\).

Throws: An exception object of type \text{out_of_range} if no such element is present.

Complexity: logarithmic.

23.3.1.3 map modifiers

\[\text{template <class P> pair<iterator, bool> insert(P&& x);}\]

\[\text{template <class P> pair<iterator, bool> insert(const_iterator position, P&& x);}\]

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}, \text{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 pair<\text{key_type},\text{mapped_type}>. Otherwise \text{CopyConstructible} is required for both \text{key_type} and \text{mapped_type}.

Draft
### 23.3.1.4 map 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;

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;
```

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` (23.1.2).

### 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);

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);
```

**Effects:**

```cpp
x.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), and of an allocator-aware container (Table 91). A multimap also provides most operations described in (23.1.2) for equal keys. This means that a multimap supports the `a_eq` operations in (23.1.2) but not the `a_uniq` operations. For a `multimap<Key,T>` the `key_type` is `Key` and the `value_type` is `pair<const Key,T>`. Descriptions are provided here only for operations on `multimap` that are not described in one of those tables or for operations where there is additional semantic information.
namespace std {
    template <class Key, class T, class Compare = less<Key>,
             class Allocator = allocator<pair<const Key, T> > >
    class multimap {
public:
    // types:
    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;
    typedef implementation-defined const_iterator;
    typedef implementation-defined size_type;
    typedef implementation-defined difference_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;

class value_compare
    : public binary_function<value_type,value_type,bool> {
friend class multimap;
protected:
    Compare comp;
    value_compare(Compare c) : comp(c) { }
public:
    bool operator()(const value_type& x, const value_type& y) const {
        return comp(x.first, y.first);
    }
};

    // 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(const Allocator&);
    multimap(const multimap&, const Allocator&);
    multimap(multimap&, const Allocator&);
    ~multimap();
    multimap<Key,T,Compare,Allocator>&
    operator=(const multimap<Key,T,Compare,Allocator>& x);
    multimap<Key,T,Compare,Allocator>&
    Draft
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 position, 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);

/* 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);

/* template implementations of the algorithms: */

template <class Key, class T, class Compare, class Allocator>
struct constructible_with_allocator_suffix<
multimap<Key, T, Compare, Alloc>>
{
    true_type {};
};

23.3.2.1 multimap constructors

explicit multimap(const Compare& comp = Compare(),
const Allocator& = Allocator());

Draft
Effects: Constructs an empty multimap using the specified comparison object and allocator.

Complexity: Constant.

```cpp
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 comp and otherwise N log N, where N is 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 todedmapped_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 find, lower_bound, upper_bound, and 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 Associative containers

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);
```

```cpp
template <class Key, class T, class Compare, class Allocator>
void swap(multimap<Key,T,Compare,Allocator>&& x,
          multimap<Key,T,Compare,Allocator>& y);
```

```cpp
template <class Key, class T, class Compare, class Allocator>
void swap(multimap<Key,T,Compare,Allocator>& x,
          multimap<Key,T,Compare,Allocator>&& y);
```

Effects:

```cpp
x.swap(y);
```

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), and of an allocator-aware container (Table 91). 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 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:
```

Draft
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(const Allocator&);
set(const set<Key,Compare,Allocator>& x);
set(set&&, const Allocator&);
~set();
set<Key,Compare,Allocator>& operator=(const set<Key,Compare,Allocator>& x);
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);
23.3 Associative containers

Containers library

```
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);

template <class Key, class Compare, class Allocator>
bool operator< (const set<Key,Compare,Allocator>& x,
const set<Key,Compare,Allocator>& y);

template <class Key, class Compare, class Allocator>
bool operator!=(const set<Key,Compare,Allocator>& x,
const set<Key,Compare,Allocator>& y);

template <class Key, class Compare, class Allocator>
bool operator> (const set<Key,Compare,Allocator>& x,
const set<Key,Compare,Allocator>& y);

template <class Key, class Compare, class Allocator>
bool operator>=(const set<Key,Compare,Allocator>& x,
const set<Key,Compare,Allocator>& y);

template <class Key, class Compare, class Allocator>
bool operator<=(const set<Key,Compare,Allocator>& x,
const set<Key,Compare,Allocator>& y);

// specialized algorithms:

```
```
template <class Key, class Compare, class Allocator>
void swap(set<Key,Compare,Allocator>& x,
        set<Key,Compare,Allocator>&& y);

template <class Key, class Compare, class Alloc>
struct constructible_with_allocator_suffix<
    set<Key, Compare, Alloc> >
    : true_type {};
}

### 23.3.3.1 set constructors, copy, and assignment

```cpp
template <class InputIterator>
set(InputIterator first, InputIterator last,
    const Compare& comp = Compare(),
    const Allocator& = Allocator());
```

1 Effects: Constructs an empty set using the specified comparison object and allocator.

2 Complexity: Constant.

```cpp
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

```cpp
template <class Key, class Compare, class Allocator>
void swap(set<Key,Compare,Allocator>& x,
        set<Key,Compare,Allocator>&& y);
```

```cpp
template <class Key, class Compare, class Allocator>
void swap(set<Key,Compare,Allocator>& x,
        set<Key,Compare,Allocator>&& y);
```

```cpp
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

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.

Draft
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), and of an allocator-aware container (Table 91). `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(const multiset<Key,Compare,Allocator>&& x);
      multiset(const Allocator&);
      multiset(const multiset&, const Allocator&);
      multiset(multiset&&, const Allocator&);
      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();
```
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;

// 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);
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;
23.3Associative containers Containers library 760

```cpp
    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);
    template <class Key, class Compare, class Allocator>
    bool operator<=(const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
```

```cpp
    // specialized algorithms:
    template <class Key, class Compare, class Allocator>
    void swap(multiset<Key,Compare,Allocator>& x,
              multiset<Key,Compare,Allocator>& y);
    template <class Key, class Compare, class Allocator>
    void swap(multiset<Key,Compare,Allocator&& x,
                   multiset<Key,Compare,Allocator>& y);
    template <class Key, class Compare, class Allocator>
    void swap(multiset<Key,Compare,Allocator& x,
                    multiset<Key,Compare,Allocator>& y);
```

```cpp
    template <class Key, class Compare, class Alloc>
    struct constructible_with_allocator_suffix<
        multiset<Key, Compare, Alloc> >
    { true_type {};
    }
}
```

### 23.3.4.1 multiset constructors

```cpp
    explicit multiset(const Compare& comp = Compare(),
                      const Allocator& = Allocator());
```

**Effects:** Constructs an empty set using the specified comparison object and allocator.

**Complexity:** Constant.
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);

template <class Key, class Compare, class Allocator>
void swap(multiset<Key,Compare,Allocator>&& x,
          multiset<Key,Compare,Allocator>& y);

Effects:

x.swap(y);

23.3.5 Class template bitset

Header <bitset> synopsis

#include <cstdint>     // 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;

    // 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 <charT, class traits, size_t N>
    basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>& x, bitset<N>& x);
    template <charT, class traits, size_t N>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& x, const bitset<N>& x);

    // constructor:
    template <charT, class traits, size_t N>
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 long val);
  template<class charT, class traits, class Allocator>
  explicit bitset(
    typename 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);

  // 23.3.5.2 bitset operations:
  bitset<N>& operator&=(const bitset<N>& rhs);
  bitset<N>& operator|=(const bitset<N>& rhs);
  bitset<N>& operator^=(const bitset<N>& rhs);
  bitset<N>& operator<<=(size_t pos);
  bitset<N>& operator>>=(size_t pos);
  bitset<N>& set();
  bitset<N>& set(size_t pos, bool val = true);
  bitset<N>& reset();
  bitset<N>& reset(size_t pos);
  bitset<N> operator~() const;
  bitset<N>& flip();
  bitset<N>& flip(size_t pos);

  // element access:
  constexpr bool operator[](size_t pos) const; // for b[i];
  reference operator[](size_t pos); // for b[i];
};
```

Draft
The template class `bitset<N>` describes an object that can store a sequence consisting of a fixed number of bits, \( N \).

3 Each bit represents either the value zero (reset) or one (set). To `toggle` a bit is to change the value zero to one, or the value one to zero. Each bit has a non-negative position \( \text{pos} \). When converting between an object of class `bitset<N>` and a value of some integral type, bit position \( \text{pos} \) corresponds to the `bit value` \( 1 \ll \text{pos} \). The integral value corresponding to two or more bits is the sum of their bit values.

4 The functions described in this subclause can report three kinds of errors, each associated with a distinct exception:

   — 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 long val);
```

*Effects:* Constructs an object of class `bitset<N>`, initializing the first \( \text{M} \) bit positions to the corresponding bit values in `val`. \( \text{M} \) is the smaller of \( \text{N} \) and the number of bits in the value representation (section 3.9) of `unsigned long` long. If \( \text{M} < \text{N} \), the remaining bit positions are initialized to zero.
23.3 Associative containers

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 <= 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

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.

Draft


765 Containers library 23.3 Associative containers

8

    Returns: *this.

bitset<N>& operator>>(size_t pos);

9

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.

10

    Returns: *this.

bitset<N>& set();

11

Effects: Sets all bits in *this.

12

    Returns: *this.

bitset<N>& set(size_t pos, bool val = true);

13

Requires: pos is valid

14

Throws: out_of_range if pos does not correspond to a valid bit position.

15

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.

16

    Returns: *this.

bitset<N>& reset();

17

Effects: Resets all bits in *this.

18

    Returns: *this.

bitset<N>& reset(size_t pos);

19

Requires: pos is valid

20

Throws: out_of_range if pos does not correspond to a valid bit position.

21

Effects: Resets the bit at position pos in *this.

22

    Returns: *this.

bitset<N> operator~() const;

23

Effects: Constructs an object x of class bitset<N> and initializes it with *this.

24

    Returns: x.flip().

bitset<N>& flip();

25

Effects: Toggles all bits in *this.

26

    Returns: *this.

bitset<N>& flip(size_t pos);
23.3 Associative containers

Requires: pos is valid

Throws: out_of_range if pos does not correspond to a valid bit position.

Effects: Toggles the bit at position pos in *this.

Returns: *this.

unsigned long to_ulong() const;

Throws: overflow_error if the integral value x corresponding to the bits in *this cannot be represented as type unsigned long.

Returns: x.

unsigned long long to_ulong() const;

Throws: overflow_error if the integral value x corresponding to the bits in *this cannot be represented as type unsigned long long.

Returns: x.

template <class charT, 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;

Draft
Returns: A nonzero value if the value of each bit in *this equals the value of the corresponding bit in rhs.

```cpp
bool operator!=(const bitset<N>& rhs) const;
```

Returns: A nonzero value if !(this == rhs).

```cpp
bool test(size_t pos) const;
```

Requires: pos is valid

Throws: out_of_range if pos does not correspond to a valid bit position.

Returns: true if the bit at position pos in *this has the value one.

```cpp
bool all() const;
```

Returns: count() == size()

```cpp
bool any() const;
```

Returns: count() != 0 true if any bit in *this is one.

```cpp
bool none() const;
```

Returns: count() == 0 true if no bit in *this is one.

```cpp
bitset<N> operator<<(size_t pos) const;
```

Returns: bitset<N>(*this) <<= pos.

```cpp
bitset<N> operator>>(size_t pos) const;
```

Returns: bitset<N>(*this) >>= pos.

```cpp
constexpr bool operator[](size_t pos) const;
```

Requires: pos shall be valid.

Throws: nothing.

Returns: test(pos).

```cpp
bitset<N>::reference operator[](size_t pos);
```

Requires: pos shall be valid.

Throws: nothing.

Returns: An object of type bitset<N>::reference such that (*this)[pos] == this->test(pos), and such that (*this)[pos] = val is equivalent to this->set(pos, val).

### 23.3.5.3 bitset operators

```cpp
bitset<N> operator&=(const bitset<N>& lhs, const bitset<N>& rhs);
```

Returns: bitset<N>(lhs) &= rhs.
bitset<N> operator|(const bitset<N>& lhs, const bitset<N>& rhs);

Returns: bitset<N>(lhs) |= rhs.

bitset<N> operator~(const bitset<N>& lhs, const bitset<N>& rhs);

Returns: bitset<N>(lhs) ^= rhs.

template <class charT, class traits, size_t N>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& is, bitset<N>& x);

A formatted input function (27.6.1.2).

Effects: Extracts up to N characters from is. Stores these characters in a temporary object str of type basic_string<charT, traits>, then evaluates the expression x = bitset<N>(str). Characters are extracted and stored until any of the following occurs:

— N characters have been extracted and stored;
— end-of-file occurs on the input sequence;
— the next input character is neither is.widen('0') nor is.widen('1') (in which case the input character is not extracted).

If no characters are stored in str, calls is.setstate(ios::failbit) (which may throw ios_base::failure (27.4.4.3)).

Returns: is.

template <class charT, class traits, size_t N>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);

Returns: os << x.template to_string<charT, traits, allocator<charT>>(); (27.6.2.6).

23.4 Unordered associative containers [unord]

Headers <unordered_map> and <unordered_set>:

Header <unordered_map> synopsis

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);
} // namespace std

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, and of an allocator-aware container (Table 91). 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>.

Draft
This section only describes operations on unordered_map that are not described in one of the requirement tables, or for which there is additional semantic information.

namespace std {
    template <class Key,
        class T,
        class Hash = hash<Key>,
        class Pred = std::equal_to<Key>,
        class 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());
        unordered_map(const unordered_map&);
        unordered_map(const Allocator&);
        unordered_map(const unordered_map&, const Allocator&);
        unordered_map(unordered_map&, const Allocator&);
        unordered_map();
        unordered_map& operator=(const unordered_map&);
        allocator_type get_allocator() const;

Draft
// 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);
template <class InputIterator> void insert(InputIterator first, InputIterator last);

iterator erase(const_iterator position);
size_type erase(const key_type& k);
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);

// 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;
23.4 Unordered associative containers  Containers library  772

// 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  [unord.map.cnstr]

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  [unord.map.elem]

mapped_type& operator[](const key_type& k);

Effects: If the unordered_map does not already contain an element whose key is equivalent to k, inserts the value std::pair<const key_type, mapped_type>(k, mapped_type()).

Returns: A reference to x.second, where x is the (unique) element whose key is equivalent to k.

Draft
23.4.1.3 unordered_map swap

```cpp
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, and of an allocator-aware container (Table 91). 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.

```cpp
namespace std {
    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 {
    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 size_type
        typedef difference_type;
        typedef local_iterator;
        typedef const_local_iterator;
```
23.4 Unordered associative containers

Containers library 774

// 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(const Allocator&);
unordered_multimap(const unordered_multimap&, const Allocator&);
unordered_multimap(unordered_multimap&, const Allocator&);
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);
template <class InputIterator> void insert(InputIterator first, InputIterator last);
iterator erase(const_iterator position);
size_type erase(const key_type& k);
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;
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);
};

templatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatetemplatel
23.4 Unordered associative containers

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.) 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

[unord.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

[unord.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, and of an allocator-aware container (Table 91). 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.

namespace std {
    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(const A l);
unordered_set(const unordered_set&, const allocator&);
unordered_set(unordered_set&&, const allocator&);
~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);
template <class InputIterator> void insert(InputIterator first, InputIterator last);
iterator erase(const_iterator position);
size_type erase(const key_type& k);
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);

template <class Value, class Hash, class Pred, class Alloc>
struct constructible_with_allocator_suffix<
    map<Value, Hash, Pred, Alloc> >
    : true_type {};
}

### 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());
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.

Complexity: Constant.

```
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());
```

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

```
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: \( x\.swap(y) \).

### 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, and of an allocator-aware container (Table 91). It provides the operations described in the preceding requirements table for equivalent keys; that is, an unordered_multiset supports the \( \text{a\_eq} \) operations in that table, not the \( \text{a\_uniq} \) operations. For an unordered_multiset\(<\text{Value}\>\) the \text{key type} and the \text{value type} are both \text{Value}. The \text{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.

```
namespace std {
    template <class Value, 
            class Hash = hash<Value>,
            class Pred = std::equal_to<Value>,
            class Alloc = std::allocator<Value> >
    class unordered_multiset 
    {}
    public:
        // types
        
        Draft
```
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(const unordered_multiset&, const Allocator&);
unordered_multiset(unordered_multiset&&, const Allocator&);
~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
Draft
template <class... Args> iterator emplace(Args&&... args);

iterator insert(const_iterator position, Args&&... args);

iterator insert(const_iterator hint, const value_type& obj);

template <class InputIterator> void insert(InputIterator first, InputIterator last);

iterator erase(const_iterator position);

size_type erase(const key_type& k);

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);

template <class Value, class Hash, class Pred, class Alloc>
struct constructible_with_allocator_suffix<
    unordered_multiset<Value, Hash, Pred, Alloc> >
    : true_type { };
23.4 Unordered associative containers

23.4.4.1 unordered_multiset constructors

```cpp
explicit unordered_multiset(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. \( \text{max_load_factor()} \) returns 1.0.

```
```
Complexity: Constant.

```cpp
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 \([f, l)\). \( \text{max_load_factor()} \) returns 1.0.

```
```
Complexity: Average case linear, worst case quadratic.

23.4.4.2 unordered_multiset swap

```cpp
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);
```

Effects: \( x.\text{swap}(y) \);

Draft
This clause describes components that C++ programs may use to perform iterations over containers (clause 23), streams (27.6), and stream buffers (27.5).

The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 97.

### Table 97: Iterators library summary

<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

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 \((*i).m\) 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.

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 98.

#### Table 98: Relations among iterator categories

<table>
<thead>
<tr>
<th>Random Access</th>
<th>Bidirectional</th>
<th>Forward</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
</table>

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. [Example: After the declaration of an uninitialized pointer x (as with int* x); x must always be assumed to have a singular value of a pointer. — end example] Results of most expressions are undefined for singular values; the only exceptions are destroying an iterator that holds a singular value 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.

An invalid iterator is an iterator that may be singular.\(^\text{263}\)

In the following sections, a and b denote values of type X or const X, n denotes a value of the difference type Distance, u, tmp, and m denote identifiers, r denotes a value of 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 99.

In Table 99, the term the domain of == is used in the ordinary mathematical sense to denote the set of values over which == is (required to be) defined. This set can change over time. Each algorithm places additional requirements on the domain of == for the iterator values it uses. These requirements can be inferred from the uses that algorithm makes of == and !=. [Example: the call find(a, b, x) is defined only if the value of a has the property p if (\*i==x) or if (\*i!=x and ++i has property p). — end example]

\(^{263}\)This definition applies to pointers, since pointers are iterators. The effect of dereferencing an iterator that has been invalidated is undefined.
Table 99: Input iterator requirements

<table>
<thead>
<tr>
<th>operation</th>
<th>type</th>
<th>semantics, pre/post-conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>X u(a);</td>
<td>X</td>
<td>post: u is a copy of a. A destructor is assumed to be present and accessible.</td>
</tr>
<tr>
<td>u = a;</td>
<td>X&amp;</td>
<td>result: u</td>
</tr>
<tr>
<td>a == b</td>
<td>convertible to bool</td>
<td>== is an equivalence relation over its domain.</td>
</tr>
<tr>
<td>a != b</td>
<td>convertible to bool</td>
<td>bool(a == b) != bool(a != b) over the domain of ==</td>
</tr>
<tr>
<td>*a</td>
<td>convertible to T</td>
<td>pre: a is dereferenceable. If a == b and (a, b) is in the domain of == then *a is equivalent to *b.</td>
</tr>
<tr>
<td>a-&gt;m</td>
<td></td>
<td>pre: (*a).m is well-defined. Equivalent to (*a).m.</td>
</tr>
<tr>
<td>++r</td>
<td>X&amp;</td>
<td>pre: r is dereferenceable. post: r is dereferenceable or r is past-the-end. post: any copies of the previous value of r are no longer required either to be dereferenceable or to be in the domain of ==.</td>
</tr>
<tr>
<td>(void)r++</td>
<td></td>
<td>equivalent to (void)++r</td>
</tr>
<tr>
<td>*r++</td>
<td>convertible to T</td>
<td>{ T tmp = *r; ++r; return tmp; }</td>
</tr>
</tbody>
</table>

3 [Note: For input iterators, a == b does not imply ++a == ++b. (Equality does not guarantee the substitution property or referential transparency.) Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be single pass algorithms. Value type T is not required to be an Assignable type (23.1). These algorithms can be used with istreams as the source of the input data through the istream_iterator class. — end note]

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 100.

Table 100: 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>X(a)</td>
<td></td>
<td></td>
<td>a = t is equivalent to X(a) = t.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>note: a destructor is assumed.</td>
</tr>
<tr>
<td>X u(a);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X u = a;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*r = o</td>
<td></td>
<td>result is not used</td>
<td></td>
</tr>
<tr>
<td>++r</td>
<td>X&amp;</td>
<td></td>
<td>kr == &amp;++r.</td>
</tr>
</tbody>
</table>
24.1 Iterator requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>operational semantics</th>
<th>assertion/note pre/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>r++</td>
<td>convertible to</td>
<td>{ X tmp = r; ++r; return tmp; }</td>
<td></td>
</tr>
<tr>
<td>const X&amp;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*r++ = o</td>
<td></td>
<td>result is not used</td>
<td></td>
</tr>
</tbody>
</table>

[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 101.

Table 101: Forward iterator requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>operational semantics</th>
<th>assertion/note pre/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X u;</td>
<td></td>
<td>note: u might have a singular value. note: a destructor is assumed.</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; post: u == a.</td>
<td></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 X&amp;</td>
<td></td>
<td>post: r == a</td>
<td></td>
</tr>
<tr>
<td>*a T&amp; if X is mutable, otherwise const</td>
<td>pre: a is dereferenceable. a == b implies *a == *b. If X is mutable, *a = t is valid.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a-&gt;m U&amp; if X is mutable, otherwise const</td>
<td>pre: (*a).m is well-defined.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
24.1.4 Bidirectional iterators

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 102.

Table 102: 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><code>r++</code></td>
<td>convertible</td>
<td><code>{ X tmp = r; ++r; return tmp; }</code></td>
<td></td>
</tr>
<tr>
<td><code>*r++</code></td>
<td>T&amp; if X is mutable, otherwise const T&amp;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

— 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.

[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]
24.1.5 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 103.

Table 103: Random access iterator requirements (in addition to bidirectional iterator)

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>operational semantics</th>
<th>assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>r += n</td>
<td>X&amp;</td>
<td>`{ 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>a + n</td>
<td>X</td>
<td>`{ X tmp = a;</td>
<td></td>
</tr>
<tr>
<td>n + a</td>
<td></td>
<td>return tmp += n; }</td>
<td></td>
</tr>
<tr>
<td>r -= n</td>
<td>X&amp;</td>
<td>return r += -n;</td>
<td></td>
</tr>
<tr>
<td>a - n</td>
<td>X</td>
<td>`{ X tmp = a;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>return tmp -= n; }</td>
<td></td>
</tr>
<tr>
<td>b - a</td>
<td>Distance</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>a[n]</td>
<td>convertible to</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>const T &amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>*(a + n)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a &lt; b</td>
<td>convertible to</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bool</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b - a &gt; 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a &gt; b</td>
<td>convertible to</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bool</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b &lt; a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a &gt;= b</td>
<td>convertible to</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bool</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>!(a &lt; b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>convertible to</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bool</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>!(a &gt; b)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

24.2 Header <iterator> synopsis

```
namespace std {
    // 24.3, primitives:
    template<class Iterator> struct iterator_traits;
    template<class T> struct iterator_traits<T*>{
```

Draft
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+(
    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 typename Container::iterator i);

template <class Iterator> 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+(typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x);

template <class Iterator>
  move_iterator<Iterator> make_move_iterator(const Iterator& i);
template <class T, class charT = char, class traits = char_traits<charT>,
class Distance = ptrdiff_t>
class istream_iterator;
template <class T, class charT, class traits, class Distance>
bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
    const istream_iterator<T,charT,traits,Distance>& y);
template <class T, class charT, class traits, class Distance>
bool operator!=(const istream_iterator<T,charT,traits,Distance>& x,
    const istream_iterator<T,charT,traits,Distance>& y);

template <class T, class charT = char, class traits = char_traits<charT> >
class ostream_iterator;
template<class charT, class traits = char_traits<charT> >
class istreambuf_iterator;
template <class charT, class traits>
bool operator==(const istreambuf_iterator<charT,traits>& a,
    const istreambuf_iterator<charT,traits>& b);
template <class charT, class traits>
bool operator!=(const istreambuf_iterator<charT,traits>& a,
    const istreambuf_iterator<charT,traits>& b);

24.3 Iterator primitives

To simplify the task of defining iterators, the library provides several classes and functions:

24.3.1 Iterator traits

To implement algorithms only in terms of iterators, it is often necessary to determine the value and difference types that correspond to a particular iterator type. Accordingly, it is required that if Iterator is the type of an iterator, the types

\[
\text{iterator_traits<Iterator>::difference_type} \\
\text{iterator_traits<Iterator>::value_type} \\
\text{iterator_traits<Iterator>::iterator_category}
\]

be defined as the iterator’s difference type, value type and iterator category, respectively. In addition, the types

\[
\text{iterator_traits<Iterator>::reference} \\
\text{iterator_traits<Iterator>::pointer}
\]

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

\[
\text{iterator_traits<Iterator>::difference_type} \\
\text{iterator_traits<Iterator>::value_type}
\]
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
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;
};
```
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;
    }
}
```

24.3.2 Basic iterator

The iterator template may be used as a base class to ease the definition of required types for new iterators.

```cpp
namespace std {
    template<class Category, class T, class Distance = ptrdiff_t,
              class Pointer = T*, class Reference = T&>
    struct iterator {
        typedef T value_type;
        typedef Distance difference_type;
        typedef Pointer pointer;
        typedef Reference reference;
        typedef Category iterator_category;
    };
}
```

24.3.3 Standard iterator tags

It is often desirable for a function template specialization to find out what is the most specific category of its iterator argument, so that the function can select the most efficient algorithm at compile time. To facilitate this, the library introduces category tag classes which are used as compile time tags for algorithm selection. They are: input_iterator_tag, output_iterator_tag, forward_iterator_tag, bidirectional_iterator_tag and random_access_iterator_tag. For every iterator of type Iterator, iterator_traits<Iterator>::iterator_category shall be defined to be the most specific category tag that describes the iterator’s behavior.

```cpp
namespace std {
    struct input_iterator_tag { };
    struct output_iterator_tag { };
    struct forward_iterator_tag: public input_iterator_tag { };
    struct bidirectional_iterator_tag: public forward_iterator_tag { };
    struct random_access_iterator_tag: public bidirectional_iterator_tag { };
}
```
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 ---

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 ---

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);
```

2 Requires: \( n \) shall be negative only for bidirectional and random access iterators.

3 Effects: Increments (or decrements for negative \( n \)) iterator reference \( i \) by \( n \).

```cpp
template<class InputIterator>
typename iterator_traits<InputIterator>::difference_type
  distance(InputIterator first, InputIterator last);
```

4 Effects: Returns the number of increments or decrements needed to get from \( \text{first} \) to \( \text{last} \).

5 Requires: \( \text{last} \) shall be reachable from \( \text{first} \).

```cpp
template <class InputIterator>
InputIterator next(InputIterator x,
  typename std::iterator_traits<InputIterator>::difference_type n = 1);
```

6 Effects: Equivalent to \( \text{advance}(x, n) \); return \( x \);

```cpp
template <class BidirectionalIterator>
BidirectionalIterator prev(BidirectionalIterator x,
  typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);
```

7 Effects: Equivalent to \( \text{advance}(x, -n) \); return \( x \);

24.4 Predefined iterators

24.4.1 Reverse iterators

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: \( \&\ast(\text{reverse} \_ \text{iterator}(i)) = \&\ast((i - 1)) \).

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;
  };
```
24.4 Predefined iterators

Iterators library

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);

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);
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>
typename reverse_iterator<Iterator>::difference_type operator- (const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

template <class Iterator>
reverse_iterator<Iterator> operator+ (typename reverse_iterator<Iterator>::difference_type n, const reverse_iterator<Iterator>& x);

24.4.1.2 reverse_iterator requirements

The template parameter Iterator shall meet all the requirements of a Bidirectional Iterator (24.1.4).

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();

Effects: Default initializes 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 x);

Effects: Initializes current with x.

template <class U> reverse_iterator(const reverse_iterator<U>& u);

Effects: Initializes current with u.current.

24.4.1.3.2 reverse_iterator::operator=

template <class U>
reverse_iterator&
operator=(const reverse_iterator<U>& u);

Effects: Assigns u.base() to current.

Returns: *this.
24.4 Predefined iterators

24.4.1.3.3 Conversion

Iterator base() const;  // explicit

Returns: current.

24.4.1.3.4 operator*

reference operator*() const;

Effects:
  this->tmp = current;
  --this->tmp;
  return *this->tmp;

[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->

pointer operator->() const;

Returns:
  &(operator*());

24.4.1.3.6 operator++

reverse_iterator& operator++();

Effects: --current;

Returns: *this.

reverse_iterator operator++(int);

Effects:
  reverse_iterator tmp = *this;
  --current;
  return tmp;

24.4.1.3.7 operator--

reverse_iterator& operator--();
799 Iterators library

24.4 Predefined iterators

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);
1 Effects: current += n;
2 Returns: *this.

24.4.1.3.12 operator[]

unspecified operator[](typename reverse_iterator<Iterator>::difference_type n) const;
24.4 Predefined iterators

24.4.1.3.13 operator==

```cpp
template <class Iterator1, class Iterator2>
bool operator==(const reverse_iterator<Iterator1>& x,
                const reverse_iterator<Iterator2>& y);
```

Returns: \(x.\text{current} == y.\text{current}\).

24.4.1.3.14 operator<

```cpp
template <class Iterator1, class Iterator2>
bool operator<(const reverse_iterator<Iterator1>& x,
               const reverse_iterator<Iterator2>& y);
```

Returns: \(x.\text{current} > y.\text{current}\).

24.4.1.3.15 operator!=

```cpp
template <class Iterator1, class Iterator2>
bool operator!=(const reverse_iterator<Iterator1>& x,
               const reverse_iterator<Iterator2>& y);
```

Returns: \(x.\text{current} \neq y.\text{current}\).

24.4.1.3.16 operator>

```cpp
template <class Iterator1, class Iterator2>
bool operator>(const reverse_iterator<Iterator1>& x,
               const reverse_iterator<Iterator2>& y);
```

Returns: \(x.\text{current} < y.\text{current}\).

24.4.1.3.17 operator>=

```cpp
template <class Iterator1, class Iterator2>
bool operator>=(const reverse_iterator<Iterator1>& x,
                const reverse_iterator<Iterator2>& y);
```

Returns: \(x.\text{current} \leq y.\text{current}\).
24.4.1.3.18  operator<=  

```cpp
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-

```cpp
template <class Iterator1, class Iterator2>
type name 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+

```cpp
template <class Iterator>
reverse_iterator<Iterator> operator+(n,
    const reverse_iterator<Iterator>& x);
```

*Returns:* \(\text{reverse}\_\_\_\_\_\text{iterator}\<\text{Iterator}> (x\.current - n)\).

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,  
```cpp
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=(\text{const } T & x)\ is defined on insert iterators to allow writing into them. If the insert iterator is pointing. In other words, an insert iterator is like a cursor pointing into the container where the insertion takes place. \(\text{back}\_\_\_\_\_\text{insert}\_\_\_\_\text{iterator}\) inserts elements at the end of a container, \(\text{front}\_\_\_\_\_\text{insert}\_\_\_\_\text{iterator}\) inserts elements at the beginning of a container, and \(\text{insert}\_\_\_\_\_\text{iterator}\) inserts elements where the iterator points in to a container. \text{back}\_\_\_\_\text{inserter}, \text{front}\_\_\_\_\text{inserter}, and \text{inserter} are three functions making the insert iterators out of a container.

24.4.2.1  Class template back_insert_iterator

Draft
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

explicit back_insert_iterator(Container& x);

    Effects: Initializes container with &x.

24.4.2.2.2 back_insert_iterator::operator=

back_insert_iterator<Container>&
    operator=(typename Container::const_reference value);

    Effects: container->push_back(value);

    Returns: *this.

back_insert_iterator<Container>&
    operator=(typename Container::value_type&& value);

    Effects: container->push_back(std::move(value));

    Returns: *this.

24.4.2.2.3 back_insert_iterator::operator*

Draft
back_inserter<Container>& operator*();  

Returns: *this.

24.4.2.4 back_inserter::operator++  
back_inserter<Container>& operator++();
back_inserter<Container> operator++(int);

Returns: *this.

24.4.2.5 back_inserter  
template <class Container>  
back_inserter<Container> back_inserter(Container& x);

Returns: back_inserter<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++();
        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);
24.4 Predefined iterators

24.4.2.4.2 front_insert_iterator::operator=

front_insert_iterator<Container>&
operator=(typename Container::const_reference value);

Effects: container->push_front(value);
Returns: *this.

front_insert_iterator<Container>&
operator=(typename Container::value_type&& value);

Effects: container->push_front(std::move(value));
Returns: *this.

24.4.2.4.3 front_insert_iterator::operator*

front_insert_iterator<Container>& operator*();

Returns: *this.

24.4.2.4.4 front_insert_iterator::operator++

front_insert_iterator<Container>& operator++();
front_insert_iterator<Container> operator++(int);

Returns: *this.

24.4.2.5 front_inserter

template <class Container>
front_insert_iterator<Container> front_inserter(Container& x);

Returns: front_insert_iterator<Container>(x).

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 typename Container::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.

24.4.2.6.3 insert_iterator::operator*
    insert_iterator<Container>& operator*();

Draft
24.4 Predefined iterators

24.4.2.6.4 insert_iterator::operator++

insert_iterator<Container>& operator++();
insert_iterator<Container>& operator++(int);

Returns: *this.

24.4.2.6.5 inserter

template <class Container, class Inserter>
insert_iterator<Container> inserter(Container& x, Inserter typename Container::iterator i);

Returns: insert_iterator<Container>(x, typename Container::iterator(i) 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:

```cpp
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
```

24.4.3.1 Class template move_iterator

```cpp
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 Iterator;
        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);

template <class Iterator1, class Iterator2>
typename move_iterator<Iterator>::difference_type operator-(
  const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
move_iterator<Iterator> operator+(()
  typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x);

template <class Iterator1, class Iterator2>
move_iterator<Iterator> make_move_iterator(const Iterator& i);
24.4 Predefined iterators

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

move_iterator();

Effects: Constructs a move_iterator, default initializing current.

explicit move_iterator(Iterator i);

Effects: Constructs a move_iterator, initializing current with i.

template <class U> move_iterator(const move_iterator<U>& u);

Effects: Constructs a move_iterator, initializing current with u.base().

Requires: U shall be convertible to Iterator.

24.4.3.3.2 move_iterator::operator=

template <class U> move_iterator& operator=(const move_iterator<U>& u);

Effects: Assigns u.base() to current.

Requires: U shall be convertible to Iterator.

24.4.3.3.3 move_iterator conversion

Iterator base() const;

Returns: current.

24.4.3.3.4 move_iterator::operator*

reference operator*() const;

Returns: *current, implicitly converted to an rvalue reference.

24.4.3.3.5 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).

24.4.3.9 move_iterator::operator+=

move_iterator& operator+=(difference_type n);

Effects: current += n.

Returns: *this.

24.4.3.10 move_iterator::operator-

move_iterator operator-(difference_type n) const;

Returns: move_iterator(current - n).
24.4 Predefined iterators

24.4.3.3.11 move_iterator::operator-=

move_iterator& operator-=(difference_type n);
1  \textit{Effects:} current \ -= \ n.
2  \textit{Returns:} \*this.

24.4.3.3.12 move_iterator::operator[]

reference operator[](difference_type n) const;
1  \textit{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  \textit{Returns:} x.base() \ == \ y.base().

template <class Iterator1, class Iterator2>
bool operator!=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
2  \textit{Returns:} !(x \ == \ y).

template <class Iterator1, class Iterator2>
bool operator<(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
3  \textit{Returns:} x.base() \ < \ y.base().

template <class Iterator1, class Iterator2>
bool operator<=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
4  \textit{Returns:} !(y \ < \ x).

template <class Iterator1, class Iterator2>
bool operator>(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
5  \textit{Returns:} y \ < \ x.

template <class Iterator1, class Iterator2>
bool operator>=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
6  \textit{Returns:} !(x \ < \ y).

24.4.3.3.14 move_iterator non-member functions

template <class Iterator1, class Iterator2>
typename move_iterator<Iterator>::difference_type operator-=(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

Draft
1. Returns: \(x.\text{base}() - y.\text{base}()\).

```cpp
template <class Iterator>
move_iterator<Iterator> operator+(    
    typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x);
```

2. Returns: \(x + n\).

```cpp
template <class Iterator>
move_iterator<Iterator> make_move_iterator(const Iterator& i);
```

3. Returns: \(\text{move iterators} <\text{Iterator}> (i)\).

### 24.5 Stream iterators

To make it possible for algorithmic templates to work directly with input/output streams, appropriate iterator-like class templates are provided.

[Example:]

```cpp
partial_sum_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` 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 main peculiarity of the `istream iterators` is the fact that `++` operators are not equality preserving, that is, \(i == j\) does not guarantee at all that `++i == ++j`. Every time `++` is used a new value is read.

2. 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.

3. 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<
typedef basic_istream<charT,traits> istream_type;
typedef basic_istream<charT,traits> istream_type;
istream_iterator();
istream_iterator();
istream_iterator(istream_type& s);
istream_iterator(istream_type& s);
istream_iterator(const istream_iterator<T,charT,traits,Distance>& x);
istream_iterator(const istream_iterator<T,charT,traits,Distance>& x);
~istream_iterator();
~istream_iterator();
const T& operator*() const;
const T& operator*() const;
const T* operator->() const;
const T* operator->() const;
istream_iterator<T,charT,traits,Distance>& operator++();
istream_iterator<T,charT,traits,Distance>& operator++();
istream_iterator<T,charT,traits,Distance> operator++(int);
istream_iterator<T,charT,traits,Distance> operator++(int);
private:
private:
    // basic_istream<charT,traits>* in_stream; exposition only
    // T value; exposition only
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);**

*Effects:* Constructs a copy of *x*.

**~istream_iterator();**

*Effects:* The iterator is destroyed.

24.5.1.2 **istream_iterator operations**

**const T& operator*() const;**

*Returns:* *value*.

**const T* operator->() const;**

*Returns:* &(*operator*()).

**istream_iterator<T,charT,traits,Distance>& operator++();**

**istream_iterator<T,charT,traits,Distance>& operator++();**

Draft
Effects: \*in\_stream >> value.

Returns: \*this.

```
istream_iterator<T,charT,traits,Distance> operator++(int);
```

Effects:
```
istream_iterator<T,charT,traits,Distance> tmp = \*this;
*in\_stream >> value;
return (tmp);
```

```
template <class T, class charT, class traits, class Distance>
bool operator==(const istream_iterator<T,charT,traits,Distance> &x,
               const istream_iterator<T,charT,traits,Distance> &y);
```

Returns: (x.in\_stream == y.in\_stream).

```
template <class T, class charT, class traits, class Distance>
bool operator!=(const istream_iterator<T,charT,traits,Distance> &x,
               const istream_iterator<T,charT,traits,Distance> &y);
```

Returns: !(x == y)

### 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
```
while (first != last)
  *result++ = *first++;
```

**ostream\_iterator** is defined as:
```
namespace std {
  template <class T, class charT = char, class traits = char_traits<charT> >
  class ostream_iterator:
    public 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++();
```
24.5 Stream iterators

```cpp
ostream_iterator<T, charT, traits>& operator++(int);
private:
   // basic_ostream<charT,traits>* out_stream; exposition only
   // const charT* delim; exposition only
};
```

### 24.5.2.1 ostream_iterator constructors and destructor

- **ostream_iterator(ostream_type& s);**

  *Effects:* Initializes `out_stream` with `s` and `delim` with null.

- **ostream_iterator(ostream_type& s, const charT* delimiter);**

  *Effects:* Initializes `out_stream` with `s` and `delim` with `delimiter`.

- **ostream_iterator(const ostream_iterator& x);**

  *Effects:* Constructs a copy of `x`.

  `~ostream_iterator();`

  *Effects:* The iterator is destroyed.

### 24.5.2.2 ostream_iterator operations

- **ostream_iterator& operator=(const T& value);**

  *Effects:

  ```
  *out_stream << value;
  if(delim != 0)
  *out_stream << delim;
  return (*this);
  ```

- **ostream_iterator& operator*();**

  *Returns:* *this.

- **ostream_iterator& operator++();**

  *Returns:* *this.

- **ostream_iterator& operator++(int);**

  *Returns:* *this.

### 24.5.3 Class template istreambuf_iterator

```cpp
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> {
```
The class template `istreambuf_iterator` reads successive characters from the streambuf for which it was constructed. `operator*()` provides access to the current input character, if any. Each time `operator++` is evaluated, the iterator advances to the next input character. If the end of stream is reached (streambuf_type::sgetc() returns traits::eof()), the iterator becomes equal to the end of stream iterator value. The default constructor `istreambuf_iterator()` and the constructor `istreambuf_iterator(0)` both construct an end of stream iterator object suitable for use as an end-of-range.

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.

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.

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 Stream iterators

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.cons]

istreambuf_iterator() throw();

1  Effects: Constructs the end-of-stream iterator.

istreambuf_iterator(basic_istream<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.

24.5.3.3 istreambuf_iterator::operator* [istreambuf.iterator::op*]

charT operator*() const

1  Returns: The character obtained via the streambuf member sbuf_->sgetc().

24.5.3.4 istreambuf_iterator::operator++ [istreambuf.iterator::op++]

istreambuf_iterator<charT,traits>&

1    istreambuf_iterator<charT,traits>::operator++();

2        Effects: sbuf_->sbumpc().

3        Returns: *this.

proxy istreambuf_iterator<charT,traits>::operator++(int);

3        Returns: proxy(sbuf_->sbumpc(), sbuf_).
24.5.3.5 istreambuf_iterator::equal

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==

template <class charT, class traits>
bool operator==(const istreambuf_iterator<charT,traits>& a,
const istreambuf_iterator<charT,traits>& b);

Returns: a.equal(b).

24.5.3.7 operator!=

template <class charT, class traits>
bool operator!=(const istreambuf_iterator<charT,traits>& a,
const istreambuf_iterator<charT,traits>& b);

Returns: !a.equal(b).

24.5.4 Class template ostreambuf_iterator

namespace std {
  template <class charT, class traits = char_traits<charT> >
  class ostreambuf_iterator :
       public iterator<output_iterator_tag, void, void, void, void> {
       public:
          typedef charT char_type;
          typedef traits traits_type;
          typedef basic_streambuf<charT,traits> streambuf_type;
          typedef basic_ostream<charT,traits> ostream_type;

       public:
          ostreambuf_iterator(ostream_type& s) 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
           
       };  
  
}  

Draft
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

```cpp
ostreambuf_iterator(ostream_type& s) throw();
```

**Requires:** `s.rdbuf()` shall not null pointer.

**Effects:** `sbuf_`

```cpp
ostreambuf_iterator(streambuf_type* s) throw();
```

**Requires:** `s` shall not be a null pointer.

**Effects:** `sbuf_`

### 24.5.4.2 `ostreambuf_iterator` operations

```cpp
ostreambuf_iterator<charT,traits>& operator=(charT c);
```

**Effects:** If `failed()` yields `false`, calls `sbuf_->sputc(c)`; otherwise has no effect.

**Returns:** *this.

```cpp
ostreambuf_iterator<charT,traits>& operator*();
```

**Returns:** *this.

```cpp
ostreambuf_iterator<charT,traits>& operator++();
ostreambuf_iterator<charT,traits>& operator++(int);
```

**Returns:** *this.

```cpp
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.

Draft
Chapter 25  Algorithms library

1 This clause describes components that C++ programs may use to perform algorithmic operations on containers (clause 23) and other sequences.

2 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 104.

<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></td>
</tr>
<tr>
<td>25.3 Sorting and related operations</td>
<td></td>
</tr>
<tr>
<td>25.4 C library algorithms</td>
<td>&lt;cstdlib&gt;</td>
</tr>
</tbody>
</table>

**Header <algorithm> synopsis**

```cpp
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, class BinaryPredicate>
    ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1, 
    ForwardIterator2 first2, ForwardIterator2 last2);
    template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
    ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1, 
    ForwardIterator2 first2, ForwardIterator2 last2, 
    BinaryPredicate pred);

    template<class InputIterator, class ForwardIterator>
    InputIterator
    
```
find_first_of(InputIterator first1, InputIterator last1,
         ForwardIterator first2, ForwardIterator last2);

template<class InputIterator, class ForwardIterator,
         class BinaryPredicate>
InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2,
                  BinaryPredicate pred);

template<class ForwardIterator>
    ForwardIterator adjacent_find(ForwardIterator first,
                                   ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
    ForwardIterator adjacent_find(ForwardIterator first,
                                   ForwardIterator last,
                                   BinaryPredicate pred);

template<class InputIterator, class T>
    typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& value);

template<class InputIterator, class Predicate>
    typename iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred);

template<class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
    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);
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>
ForwardIterator1 search_n(ForwardIterator first, ForwardIterator last,
Size count, const T& value,
BinaryPredicate pred);

// 25.2, modifying sequence operations:
// 25.2.1, copy:
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);

// 25.2.2, move:
template<class InputIterator, class OutputIterator>
OutputIterator move(InputIterator first, InputIterator last,
OutputIterator result);
template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 move_backward(
BidirectionalIterator1 first, BidirectionalIterator1 last,
BidirectionalIterator2 result);

// 25.2.3, swap:
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 ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);

template<class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last,
OutputIterator result, UnaryOperation op);
template<class InputIterator1, class InputIterator2, class OutputIterator,
class BinaryOperation>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, OutputIterator result,
BinaryOperation binary_op);

template<class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
const T& old_value, const T& new_value);

Draft
template<class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last,
    Predicate pred, const T& new_value);

template<class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
    OutputIterator result,
    const T& old_value, const T& new_value);

template<class InputIterator, class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if(InputIterator first, InputIterator last,
    OutputIterator result,
    Predicate pred, const T& new_value);

template<class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value);

template<class OutputIterator, class Size, class T>
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);

template<class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
    BinaryPredicate pred);

template<class InputIterator, class OutputIterator>
OutputIterator unique_copy(InputIterator first, InputIterator last,
    OutputIterator result);

template<class InputIterator, class OutputIterator, class BinaryPredicate>
OutputIterator unique_copy(InputIterator first, InputIterator last,
    OutputIterator result, BinaryPredicate pred);

template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);

Draft
template<class BidirectionalIterator, class OutputIterator>
    OutputIterator reverse_copy(BidirectionalIterator first,
                                BidirectionalIterator last,
                                OutputIterator result);

template<class ForwardIterator>
    ForwardIterator rotate(ForwardIterator first, ForwardIterator middle,
                            ForwardIterator last);

template<class ForwardIterator, class OutputIterator>
    OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
                                ForwardIterator last, OutputIterator result);

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:

template<class BidirectionalIterator, class Predicate>
    BidirectionalIterator partition(BidirectionalIterator first,
                                     BidirectionalIterator last,
                                     Predicate pred);

template<class BidirectionalIterator, class Predicate>
    BidirectionalIterator stable_partition(BidirectionalIterator first,
                                            BidirectionalIterator last,
                                            Predicate pred);

// 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);

Draft
void partial_sort(RandomAccessIterator first,
    RandomAccessIterator middle,
    RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void partial_sort(RandomAccessIterator first,
    RandomAccessIterator middle,
    RandomAccessIterator last, Compare comp);

template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(
    InputIterator first, InputIterator last,
    RandomAccessIterator result_first,
    RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator, class Compare>
RandomAccessIterator partial_sort_copy(
    InputIterator first, InputIterator last,
    RandomAccessIterator result_first,
    RandomAccessIterator result_last,
    Compare comp);

template<class ForwardIterator>
bool is_sorted(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
bool is_sorted(ForwardIterator first, ForwardIterator last,
    Compare comp);

template<class ForwardIterator>
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last,
    Compare comp);

template<class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
    RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
    RandomAccessIterator last, Compare comp);

    // 25.3.3, 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);

Draft
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);

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>
Draft
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
void pop_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

void make_heap(RandomAccessIterator first, RandomAccessIterator last);
void make_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
void sort_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
bool is_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

RandomAccessIterator is_heap_until(RandomAccessIterator first, RandomAccessIterator last);
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> const T& min(const T& a, const T& b, const T& c);
template<class T, class... Args>
  const T& min(const T& a, const Args&... args);
template<class T, class Compare>
  const T& min(const T& a, const T& b, Compare comp);
template<class T, class U, class... Args>
  const T& min(const T& a, const U& b, const Args&... args);

template<class T> const T& max(const T& a, const T& b);
template<class T> const T& max(const T& a, const T& b, const T& c);
template<class T, class... Args>
  const T& max(const T& a, const Args&... args);
template<class T, class Compare>
  const T& max(const T& a, const T& b, Compare comp);
template<class T, class U, class... Args>
  const T& max(const T& a, const U& b, const Args&... args);

template<class T> pair<const T&, const T&> minmax(const T& a, const T& b);
template<class T> pair<const T&, const T&> minmax(const T& a, const T& b, const T& c);
template<class T, class... Args>
  pair<const T&, const T&> minmax(const T& a, const Args&... args);
template<class T, class Compare>
    pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
template<class T, class U, class... Args>
    pair<const T&, const T&> minmax(const T& a, const U& b, const Args&... args);

template<class ForwardIterator>
    ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
    ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
                                Compare comp);
template<class ForwardIterator>
    ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
    ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
                                Compare comp);

template<class ForwardIterator>
    pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
    pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first, ForwardIterator last,
                                                            Compare comp);

template<class InputIterator1, class InputIterator2>
    bool lexicographical_compare(
        InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class Compare>
    bool lexicographical_compare(
        InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
        Compare comp);

// 25.3.9, permutations:
template<class BidirectionalIterator>
    bool next_permutation(BidirectionalIterator first,
                          BidirectionalIterator last);
template<class BidirectionalIterator, class Compare>
    bool next_permutation(BidirectionalIterator first,
                          BidirectionalIterator last, Compare comp);
template<class BidirectionalIterator>
    bool prev_permutation(BidirectionalIterator first,
                          BidirectionalIterator last);
template<class BidirectionalIterator, class Compare>
    bool prev_permutation(BidirectionalIterator first,
                          BidirectionalIterator last, Compare comp);

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. 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.

[264] 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`.

Draft
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

```
X tmp = a;
advance(tmp, n);
return tmp;
```

and that of b−a is the same as of

```
return distance(a, b);
```

25.1 Non-modifying sequence operations

25.1.1 For each

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);
```

1. **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]

2. **Returns:** f.

3. **Complexity:** Applies f exactly last − first times.

4. **Remarks:** If f returns a result, the result is ignored.

25.1.2 Find

```
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);
```

1. **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.

2. **Complexity:** At most last − first applications of the corresponding predicate.

25.1.3 Find End

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1
find_end(ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2);
```

Draft
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: \(*i + n == *(first2 + n)\), \(\text{pred}(i + n, *(first2 + n)) != false\). Returns \(last1\) if no such iterator is found.

**Complexity**: At most \((last2 - first2) \times (last1 - first1 - (last2 - first2) + 1)\) applications of the corresponding predicate.

### 25.1.4 Find First

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);

**Effects**: Finds an element that matches one of a set of values.

**Returns**: The first iterator \(i\) in the range \([first1, last1)\) such that for some iterator \(j\) in the range \([first2, last2)\) the following conditions hold: \(*i == *j\), \(\text{pred}(i, *j) != false\). Returns \(last1\) if no such iterator is found.

**Complexity**: At most \((last1 - first1) \times (last2 - first2)\) applications of the corresponding predicate.

### 25.1.5 Adjacent find

template<class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last,
    BinaryPredicate pred);

**Returns**: The first iterator \(i\) such that both \(i\) and \(i + 1\) are in the range \([first, last)\) for which the following corresponding conditions hold: \(*i == *(i + 1), \text{pred}(i, *(i + 1)) != false\). Returns \(last\) if no such iterator is found.

Draft
25.1 Non-modifying sequence operations

Complexity: For a nonempty range, exactly \( \min((i - \text{first}) + 1, (\text{last} - \text{first}) - 1) \) applications of the corresponding predicate, where \( i \) is adjacent_find’s return value.

25.1.6 Count  [alg.count]

\[
\text{template<class InputIterator, class T>}
\text{typename iterator_traits<InputIterator>::difference_type}
\text{count(InputIterator first, InputIterator last, const T& value);}
\]

\[
\text{template<class InputIterator, class Predicate>}
\text{typename iterator_traits<InputIterator>::difference_type}
\text{count_if(InputIterator first, InputIterator last, Predicate pred);}
\]

Effects: Returns the number of iterators \( i \) in the range \([\text{first}, \text{last})\) for which the following corresponding conditions hold: \( *i == \text{value}, \text{pred}(*i) != \text{false} \).

Complexity: Exactly \( \text{last} - \text{first} \) applications of the corresponding predicate.

25.1.7 Mismatch  [mismatch]

\[
\text{template<class InputIterator1, class InputIterator2>}
\text{pair<InputIterator1, InputIterator2>}
\text{mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);}
\]

\[
\text{template<class InputIterator1, class InputIterator2, class BinaryPredicate>}
\text{pair<InputIterator1, InputIterator2>}
\text{mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, BinaryPredicate pred);}
\]

Returns: A pair of iterators \( i \) and \( j \) such that \( j == \text{first2} + (i - \text{first1}) \) and \( i \) is the first iterator in the range \([\text{first1}, \text{last1})\) for which the following corresponding conditions hold:

\[
!(i == *(first2 + (i - first1)))
\text{pred}(*i, *(first2 + (i - first1))) \text{== false}
\]

Returns the pair \( \text{last1} \) and \( \text{first2} + (\text{last1} - \text{first1}) \) if such an iterator \( i \) is not found.

Complexity: At most \( \text{last1} - \text{first1} \) applications of the corresponding predicate.

25.1.8 Equal  [alg.equal]

\[
\text{template<class InputIterator1, class InputIterator2>}
\text{bool equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);}
\]

\[
\text{template<class InputIterator1, class InputIterator2, class BinaryPredicate>}
\]

Draft
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, BinaryPredicate pred);

Returns: true if for every iterator \(i\) in the range \([\text{first1}, \text{last1})\) the following corresponding conditions hold: \(*i == *(\text{first2} + (i - \text{first1})), \text{pred}(\*i, *(\text{first2} + (i - \text{first1}))) \neq \text{false}\). Otherwise, returns false.

Complexity: At most \(\text{last1} - \text{first1}\) applications of the corresponding predicate.

25.1.9 Search

```

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);
```

Effects: Finds a subsequence of equal values in a sequence.

Returns: The first iterator \(i\) in the range \([\text{first1}, \text{last1} - (\text{last2} - \text{first2}))\) such that for any non-negative integer \(n\) less than \(\text{last2} - \text{first2}\) the following corresponding conditions hold: \(*(i + n) == *(\text{first2} + n), \text{pred}(*(i + n), *(\text{first2} + n)) \neq \text{false}\). Returns \(\text{last1}\) if no such iterator is found.

Complexity: At most \((\text{last1} - \text{first1}) \times (\text{last2} - \text{first2})\) applications of the corresponding predicate.

```

template<class ForwardIterator, class Size, class T>
ForwardIterator
search_n(ForwardIterator first, ForwardIterator last, Size count,
         const T& value);

template<class ForwardIterator, class Size, class T,
         class BinaryPredicate>
ForwardIterator
search_n(ForwardIterator first, ForwardIterator last, Size count,
         const T& value, BinaryPredicate pred);
```

Requires: The type \(\text{Size}\) shall be convertible to integral type (4.7, 12.3).

Effects: Finds a subsequence of equal values in a sequence.

Returns: The first iterator \(i\) in the range \([\text{first}, \text{last} - \text{count})\) such that for any non-negative integer \(n\) less than \(\text{count}\) the following corresponding conditions hold: \(*(i + n) == \text{value}, \text{pred}(*(i + n), \text{value}) \neq \text{false}\). Returns \(\text{last}\) if no such iterator is found.

Complexity: At most \((\text{last} - \text{first}) \times \text{count}\) applications of the corresponding predicate if \(\text{count}\) is positive, or 0 otherwise.
25.2 Mutating sequence operations

25.2.1 Copy

```cpp
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                     OutputIterator result);
```

1. **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)*.\n
2. **Returns:** \(result + (last - first)\).

3. **Requires:** \(result\) shall not be in the range \([first, last)\).

4. **Complexity:** Exactly \(last - first\) assignments.

```cpp
template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2
    copy_backward(BidirectionalIterator1 first,
                  BidirectionalIterator1 last,
                  BidirectionalIterator2 result);
```

5. **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 \(n \leq (last - first)\), performs \(*(result - n) = *(last - n)*.\n
6. **Requires:** \(result\) shall not be in the range \([first, last)\).

7. **Returns:** \(result - (last - first)\).

8. **Complexity:** Exactly \(last - first\) assignments.

25.2.2 Move

```cpp
template<class InputIterator, class OutputIterator>
OutputIterator move(InputIterator first, InputIterator last,
                     OutputIterator result);
```

1. **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))*.

2. **Returns:** \(result + (last - first)\).

3. **Requires:** \(result\) shall not be in the range \([first, last)\).

4. **Complexity:** Exactly \(last - first\) move assignments.

---

\[^{265}\text{copy_backward should be used instead of copy when last is in the range [result - (last - first), result).}\]
25.2 Mutating sequence operations

25.2.3 Swap

\[ \text{template<class T> void swap(T& a, T& b);} \]

\[ \text{Requires: Type T shall be MoveConstructible (33) and MoveAssignable (35).} \]

\[ \text{Effects: Exchanges values stored in two locations.} \]

\[ \text{template<class ForwardIterator1, class ForwardIterator2>} \]

\[ \text{ForwardIterator2} \]

\[ \text{swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,} \]

\[ \text{ForwardIterator2 first2);} \]

\[ \text{Effects: For each non-negative integer } n < (\text{last1} - \text{first1}) \text{ performs: swap(*(first1 + n), *(first2 + n)).} \]

\[ \text{Requires: The two ranges [first1, last1) and [first2, first2 + (last1 - first1)) shall not overlap.} \]

\[ \text{The type of *first1 shall be the same as the type of *first2 and that type shall satisfy the Swappable requirements (37).} \]

\[ \text{Returns: first2 + (last1 - first1).} \]

\[ \text{Complexity: Exactly last1 - first1 swaps.} \]

\[ \text{template<class ForwardIterator1, class ForwardIterator2>} \]

\[ \text{void iter_swap(ForwardIterator1 a, ForwardIterator2 b);} \]

\[ \text{Effects: swap(*a, *b).} \]

\[ \text{Requires: The type of *a shall be the same as the type of *b and that type shall satisfy the Swappable requirements (37).} \]

25.2.4 Transform

\[ \text{move_backward should be used instead of move when last is in the range [result - (last - first), result).} \]
25.2 Mutating sequence operations

template<class InputIterator, class OutputIterator,
class UnaryOperation>
OutputIterator
  transform(InputIterator first, InputIterator last,
            OutputIterator result, UnaryOperation op);

template<class InputIterator1, class InputIterator2,
class OutputIterator, class BinaryOperation>
OutputIterator
  transform(InputIterator1 first1, InputIterator1 last1,
            InputIterator2 first2, OutputIterator result,
            BinaryOperation binary_op);

Effects: Assigns through every iterator \( i \) in the range \([\text{result}, \text{result} + (\text{last1} - \text{first1}))\) a new corresponding value equal to \( \text{op}(*(\text{first1} + (i - \text{result}))) \) or \( \text{binary_op}(*(\text{first1} + (i - \text{result})), *(\text{first2} + (i - \text{result}))) \).

Requires: \( \text{op} \) and \( \text{binary_op} \) shall not invalidate iterators or subranges, or modify elements in the ranges \([\text{first1}, \text{last1}]\), \([\text{first2}, \text{first2} + (\text{last1} - \text{first1})]\), and \([\text{result}, \text{result} + (\text{last1} - \text{first1})]\).

Returns: \( \text{result} + (\text{last1} - \text{first1}) \).

Complexity: Exactly \( \text{last1} - \text{first1} \) applications of \( \text{op} \) or \( \text{binary_op} \).

Remarks: \( \text{result} \) may be equal to \( \text{first} \) in case of unary transform, or to \( \text{first1} \) or \( \text{first2} \) in case of binary transform.

25.2.5 Replace

template<class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
             const T& old_value, const T& new_value);

template<class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last,
                Predicate pred, const T& new_value);

Requires: The expression \(*\text{first} = \text{new_value}\) shall be valid.

Effects: Substitutes elements referred by the iterator \( i \) in the range \([\text{first}, \text{last})\) with \( \text{new_value} \), when the following corresponding conditions hold: \( *i == \text{old_value}, \text{pred}(*i) != \text{false} \).

Complexity: Exactly \( \text{last} - \text{first} \) applications of the corresponding predicate.

Remarks: The use of fully closed ranges is intentional.
template<class InputIterator, class OutputIterator, class Predicate, class T>
OutputIterator
   replace_copy_if(InputIterator first, InputIterator last,
   OutputIterator result,
   Predicate pred, const T& new_value);

   Requires: The results of the expressions *first and new_value shall be writable to the result output iterator. The ranges [first,last) and [result,result + (last - first)) shall not overlap.

   Effects: Assigns to every iterator i in the range [result,result + (last - first)) either new_value or *(first + (i - result)) depending on whether the following corresponding conditions hold:
   *(first + (i - result)) == old_value
   pred(*(first + (i - result))) != false

   Returns: result + (last - first).

   Complexity: Exactly last - first applications of the corresponding predicate.

25.2.6 Fill

   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).

   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.

   Complexity: Exactly last - first, n, or 0 assignments, respectively.

25.2.7 Generate

   Requires: gen takes no arguments, Size shall be convertible to an integral type (4.7, 12.3).
25.2 Mutating sequence operations

Completeness: Exactly $\text{last} - \text{first}$, $n$, or 0 invocations of gen and assignments, respectively.

25.2.8 Remove

```
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);
```

Requires: The type of *first shall satisfy the MoveAssignable requirements (Table 35).

Effects: Eliminates all the elements referred to by iterator $i$ in the range $[\text{first}, \text{last})$ for which the following corresponding conditions hold: $*i == \text{value}$, $\text{pred}(*i) != \text{false}$.

Returns: The end of the resulting range.

Remarks: Stable.

Complexity: Exactly $\text{last} - \text{first}$ applications of the corresponding predicate.

```
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);
```

Requires: Type $T$ is EqualityComparable (31). The ranges $[\text{first}, \text{last})$ and $[\text{result}, \text{result} + (\text{last} - \text{first}))$ shall not overlap.

Effects: Copies all the elements referred to by the iterator $i$ in the range $[\text{first}, \text{last})$ for which the following corresponding conditions do not hold: $*i == \text{value}$, $\text{pred}(*i) != \text{false}$.

Returns: The end of the resulting range.

Complexity: Exactly $\text{last} - \text{first}$ applications of the corresponding predicate.

Remarks: Stable.

25.2.9 Unique

```
template<class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last);
```

```
template<class ForwardIterator, class BinaryPredicate>
```

Draft
For a nonempty range, eliminates all but the first element from every consecutive group of equivalent elements referred to by the iterator \( i \) in the range \([\text{first} + 1, \text{last})\) for which the following conditions hold:

\[ *(i - 1) == *i \text{ or pred}(*(i - 1), *i) != false. \]

Requires: The comparison function shall be an equivalence relation. The type of \(*\text{first}\) shall satisfy the MoveAssignable requirements (Table 35).

Returns: The end of the resulting range.

Complexity: For nonempty ranges, exactly \((\text{last} - \text{first}) - 1\) applications of the corresponding predicate.

```
template<class InputIterator, class OutputIterator>
OutputIterator
unique_copy(InputIterator first, InputIterator last,
            OutputIterator result);
```

Requires: The ranges \([\text{first}, \text{last})\) and \([\text{result}, \text{result}+(\text{last}-\text{first}))\) shall not overlap. The expression \(*\text{result} = *\text{first}\) shall be valid. 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 \([\text{first}, \text{last})\) for which the following corresponding conditions hold: \(*i == *(i - 1)\) or \(\text{pred}(*i, *(i - 1)) != false.\)

Returns: The end of the resulting range.

Complexity: For nonempty ranges, exactly \(\text{last} - \text{first} - 1\) applications of the corresponding predicate.

### 25.2.10 Reverse

```
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

Effects: For each non-negative integer \( i <= (\text{last} - \text{first})/2\), applies iter_swap to all pairs of iterators \( \text{first} + i, (\text{last} - i) - 1.\)

Requires: The type of \(*\text{first}\) shall satisfy the Swappable requirements (37).

Complexity: Exactly \((\text{last} - \text{first})/2\) swaps.

```
template<class BidirectionalIterator, class OutputIterator>
OutputIterator
reverse(BidirectionalIterator first, BidirectionalIterator last,
        OutputIterator result);
```
25.2 Mutating sequence operations

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

\[ \text{rotate}(\text{ForwardIterator} \ first, \ \text{ForwardIterator} \ middle, \ \text{ForwardIterator} \ last) ; \]

Effects: For each non-negative integer \( i < (last - first) \), places the element from the position \( first + i \) into position \( first + (i + (last - middle)) \% (last - first) \).

Returns: \( first + (last - middle) \).

Remarks: This is a left rotate.

Requires: \( \text{[first,middle)} \) and \( \text{[middle,last)} \) shall be valid ranges. The type of \( *\text{first} \) shall satisfy the Swappable requirements (37), the MoveConstructible requirements (Table 33), and the MoveAssignable requirements (Table 35).

Complexity: At most last - first swaps.

rotate_copy(ForwardIterator first, ForwardIterator middle,
            ForwardIterator last, OutputIterator result);

Effects: Copies the range \( \text{[first, last)} \) to the range \( \text{[result, result + (last - first))} \) 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 \( \text{[first, last)} \) and \( \text{[result, result + (last - first))} \) shall not overlap.

Complexity: Exactly last - first assignments.

25.2.12  Random shuffle

\[ \text{random_shuffle}(\text{RandomAccessIterator} \ first, \ \text{RandomAccessIterator} \ last) ; \]

Draft
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& g);

Effects: Permutates the elements in the range \([\text{first}, \text{last})\) such that each possible permutation of those elements has equal probability of appearance.

Requires: The type of \(*\text{first}\) shall satisfy the Swappable requirements (Table 37). The random number generating function object \(\text{rand}\) shall have a return type that is convertible to \(\text{iterator_traits<RandomAccessIterator>::difference_type}\), and the call \(\text{rand}(n)\) shall return a randomly chosen value in the interval \([0, n)\), for \(n > 0\) of type \(\text{iterator_traits<RandomAccessIterator>::difference_type}\). The function object \(g\) shall meet the requirements of uniform random number generator (26.4.1.2).

Complexity: Exactly \((\text{last} - \text{first}) - 1\) swaps.

Remarks: To the extent that the implementation of these functions makes use of random numbers, the implementation shall use the following sources of randomness:

The underlying source of random numbers for the first form of the function is implementation-defined. An implementation may use the \(\text{rand}\) function from the standard C library.

In the second form of the function, the function object \(\text{rand}\) shall serve as the implementation’s source of randomness.

In the third form of the function, the object \(g\) shall serve as the implementation’s source of randomness.

25.2.13 Partitions [alg.partitions]

template<class BidirectionalIterator, class Predicate>
    BidirectionalIterator partition(BidirectionalIterator first,
                                    BidirectionalIterator last, Predicate pred);

Effects: Places all the elements in the range \([\text{first}, \text{last})\) that satisfy \(\text{pred}\) before all the elements that do not satisfy it.

Returns: An iterator \(i\) such that for any iterator \(j\) in the range \([\text{first}, i)\) \(\text{pred}(\*j) \neq \text{false}\), and for any iterator \(k\) in the range \([i, \text{last})\), \(\text{pred}(\*k) = \text{false}\).

Requires: The type of \(*\text{first}\) shall satisfy the Swappable requirements (37).

Complexity: At most \((\text{last} - \text{first})/2\) swaps. Exactly \(\text{last} - \text{first}\) applications of the predicate are done.
stable_partition(BidirectionalIterator first, 
    BidirectionalIterator last, Predicate pred);

**Effects:** Places all the elements in the range \([\text{first},\text{last})\) that satisfy \(\text{pred}\) before all the elements that do not satisfy it.

**Returns:** An iterator \(i\) such that for any iterator \(j\) in the range \([\text{first},i)\), \(\text{pred}(\ast j) \neq \text{false}\), and for any iterator \(k\) in the range \([i,\text{last})\), \(\text{pred}(\ast k) = \text{false}\). The relative order of the elements in both groups is preserved.

**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:** At most \((\text{last} - \text{first}) \ast \log(\text{last} - \text{first})\) swaps, but only linear number of swaps if there is enough extra memory. Exactly \(\text{last} - \text{first}\) applications of the predicate.

### 25.3 Sorting and related operations [alg.sorting]

All the operations in 25.3 have two versions: one that takes a function object of type \(\text{Compare}\) and one that uses an \(\text{operator}<\).

\(\text{Compare}\) is used as a function object which returns \(\text{true}\) if the first argument is less than the second, and \(\text{false}\) otherwise. \(\text{Compare}\) \(\text{comp}\) is used throughout for algorithms assuming an ordering relation. It is assumed that \(\text{comp}\) will not apply any non-constant function through the dereferenced iterator.

For all algorithms that take \(\text{Compare}\), there is a version that uses \(\text{operator}<\) instead. That is, \(\text{comp}(\ast i, \ast j) \neq \text{false}\) defaults to \(*i < *j \neq \text{false}\). For algorithms other than those described in 25.3.3 to work correctly, \(\text{comp}\) has to induce a strict weak ordering on the values.

The term **strict** refers to the requirement of an irreflexive relation (\(!\text{comp}(x, x)\) for all \(x\)), and the term **weak** to requirements that are not as strong as those for a total ordering, but stronger than those for a partial ordering. If we define \(\text{equiv}(a, b)\) as \(!\text{comp}(a, b) \&\& !\text{comp}(b, a)\), then the requirements are that \(\text{comp}\) and \(\text{equiv}\) both be transitive relations:

- \(\text{comp}(a, b) \&\& \text{comp}(b, c)\) implies \(\text{comp}(a, c)\)
- \(\text{equiv}(a, b) \&\& \text{equiv}(b, c)\) implies \(\text{equiv}(a, c)\) [Note: Under these conditions, it can be shown that
  - \(\text{equiv}\) is an equivalence relation
  - \(\text{comp}\) induces a well-defined relation on the equivalence classes determined by \(\text{equiv}\)
  - The induced relation is a strict total ordering. — end note]

A sequence is **sorted with respect to a comparator \(\text{comp}\)** 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, \(\text{comp}(\ast(i + n), \ast i) = \text{false}\).

A sequence \([\text{start},\text{finish})\) is **partitioned with respect to an expression \(f(e)\)** if there exists an integer \(n\) such that for all \(0 \leq i < \text{distance}(\text{start}, \text{finish}), f(\ast(\text{start} + i))\) is true if and only if \(i < n\).

In the descriptions of the functions that deal with ordering relationships we frequently use a notion of equivalence to describe concepts such as stability. The equivalence to which we refer is not necessarily an \(\text{operator}==\), but an
equivalence relation induced by the strict weak ordering. That is, two elements \(a\) and \(b\) are considered equivalent if and only if \(! (a < b) \&\& ! (b < a)\).

### 25.3.1 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);
```

**Effects:** Sorts the elements in the range \([\text{first}, \text{last})\).

**Requires:** The type of \(*\text{first}\) shall satisfy the Swappable requirements (37), the MoveConstructible requirements (Table 33), and the MoveAssignable requirements (Table 35).

**Complexity:** Approximately \(N \log(N)\) (where \(N = \text{last} - \text{first}\)) comparisons on the average.\(^{268}\)

#### 25.3.1.2 stable_sort

```cpp
template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

**Effects:** Sorts the elements in the range \([\text{first}, \text{last})\).

**Requires:** The type of \(*\text{first}\) shall satisfy the Swappable requirements (37), the MoveConstructible requirements (Table 33), and the MoveAssignible requirements (Table 35).

**Complexity:** It does at most \(N \log_2(N)\) (where \(N = \text{last} - \text{first}\)) comparisons; if enough extra memory is available, it is \(N \log(N)\).

**Remarks:** Stable.

#### 25.3.1.3 partial_sort

```cpp
template<class RandomAccessIterator>
void partial_sort(RandomAccessIterator first,
                  RandomAccessIterator middle,
                  RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
```

If the worst case behavior is important \texttt{stable\_sort()} (25.3.1.2) or \texttt{partial\_sort()} (25.3.1.3) should be used.

\(^{268}\) Draft
```cpp
void partial_sort(RandomAccessIterator first,
                  RandomAccessIterator middle,
                  RandomAccessIterator last,
                  Compare comp);
```

**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.

**Requires:** The type of `*first` shall satisfy the Swappable requirements (37), the MoveConstructible requirements (Table 33), and the the MoveAssignable requirements (Table 35).

**Complexity:** It takes approximately `(last - first) * log(middle - first)` comparisons.

### 25.3.1.4 partial_sort_copy

```cpp
template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(InputIterator first, InputIterator last,
                                        RandomAccessIterator result_first,
                                        RandomAccessIterator result_last);
```

```cpp
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 the MoveAssignable requirements (Table 35).

**Complexity:** Approximately `(last - first) * log(min(last - first, result_last - result_first))` comparisons.

### 25.3.1.5 is_sorted

```cpp
template<class ForwardIterator>
bool is_sorted(ForwardIterator first, ForwardIterator last);
```

**Returns:** `is_sorted_until(first, last) == last`

```cpp
template<class ForwardIterator, class Compare>
bool is_sorted(ForwardIterator first, ForwardIterator last,
               Compare comp);
```
25.3 Sorting and related operations

25.3.2 Nth element

```
template<class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
 RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
 RandomAccessIterator last, Compare comp);
```

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
```
25.3 Sorting and related operations

lower_bound(ForwardIterator first, ForwardIterator last,  
const T& value, Compare comp);

Requires: The elements e of [first, last) shall be 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 \( \log_2 (last - first) + O(1) \) comparisons.

25.3.3.2 upper_bound

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);

Requires: The elements e of [first, last) shall be partitioned with respect to the expression !(value < e) or !comp(value, e).

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.

Complexity: At most \( \log_2 (last - first) + O(1) \) comparisons.

25.3.3.3 equal_range

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);

Requires: The elements e of [first, last) shall be partitioned with respect to the expressions e < value and !(value < e) or comp(e, value) and !comp(value, e). Also, for all elements e of [first, last), e < value shall imply !(value < e) or comp(e, value) shall imply !comp(value, e).

Returns:
make_pair(lower_bound(first, last, value),
    upper_bound(first, last, value))

or

make_pair(lower_bound(first, last, value, comp),
    upper_bound(first, last, value, comp))

Complexity: At most \(2 \times \log_2(last - first) + O(1)\) comparisons.

25.3.3.4 binary_search

\[\begin{align*}
\text{template}<&\text{class ForwardIterator, class T}> \\
\text{bool binary_search}(&\text{ForwardIterator first, ForwardIterator last,} \\
&\text{const T}&\text{& value}); \\
\text{template}<&\text{class ForwardIterator, class T, class Compare}> \\
\text{bool binary_search}(&\text{ForwardIterator first, ForwardIterator last,} \\
&\text{const T}&\text{& value, Compare comp}); \\
\end{align*}\]

 Requires: The elements \(e\) of \([first, last)\) are partitioned with respect to the expressions \(e < value\) and
\(! (value < e) \text{ or } \text{comp}(e, value)\) and \(! \text{comp}(value, e)\). Also, for all elements \(e\) of \([first, last)\), \(e < value\) implies \(! (value < e) \text{ or } \text{comp}(e, value)\) implies \(! \text{comp}(value, e)\).

 Returns: true if there is an iterator \(i\) in the range \([first, last)\) that satisfies the corresponding conditions:
\(! (i < value) \text{ and } ! (value < i) \text{ or } \text{comp}(i, value) == false \text{ and } \text{comp}(value, i) == false\).

 Complexity: At most \(\log(last - first) + 2\) comparisons.

25.3.4 Merge

\[\begin{align*}
\text{template}<&\text{class InputIterator1, class InputIterator2,} \\
&\text{class OutputIterator}> \\
\text{OutputIterator merge}(&\text{InputIterator1 first1, InputIterator1 last1,} \\
&\text{InputIterator2 first2, InputIterator2 last2,} \\
&\text{OutputIterator result}); \\
\text{template}<&\text{class InputIterator1, class InputIterator2,} \\
&\text{class OutputIterator, class Compare}> \\
\text{OutputIterator merge}(&\text{InputIterator1 first1, InputIterator1 last1,} \\
&\text{InputIterator2 first2, InputIterator2 last2,} \\
&\text{OutputIterator result, Compare comp}); \\
\end{align*}\]

 Effects: Merges two sorted ranges \([first1, last1)\) and \([first2, last2)\) into the range \([result, result + (last1 - first1) + (last2 - first2))\).
2 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.

3 Returns: result + (last1 - first1) + (last2 - first2).

4 Complexity: At most (last1 - first1) + (last2 - first2) - 1 comparisons.

5 Remarks: Stable.

6 template<class BidirectionalIterator>
   void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);

7 template<class BidirectionalIterator, class Compare>
   void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last, Compare comp);

6 Effects: Merges two sorted consecutive ranges [first,middle) and [middle,last), putting the result of the merge into the range [first,last). The resulting range will be in non-decreasing order; that is, for every iterator i in [first,last) other than first, the condition *i < *(i - 1) or, respectively, comp(*i, *(i - 1)) will be false.

7 Requires: The type of *first shall satisfy the Swappable requirements (37), the MoveConstructible requirements (Table 33), and the MoveAssignable requirements (Table 35).

8 Complexity: When enough additional memory is available, (last - first) - 1 comparisons. If no additional memory is available, an algorithm with complexity N log(N) (where N is equal to last - first) may be used.

9 Remarks: Stable.

25.3.5 Set operations on sorted structures

1 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 set_union() to contain the maximum number of occurrences of every element, set_intersection() to contain the minimum, and so on.

25.3.5.1 includes

1 template<class InputIterator1, class InputIterator2>
   bool includes(InputIterator1 first1, InputIterator1 last1,
                 InputIterator2 first2, InputIterator2 last2);

2 template<class InputIterator1, class InputIterator2, class Compare>
   bool includes(InputIterator1 first1, InputIterator1 last1,
                 InputIterator2 first2, InputIterator2 last2,
                 Compare comp);
**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 \(2 \times ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1\) comparisons.

### 25.3.5.2 set_union

```cpp
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);
```

```cpp
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);
```

**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

```cpp
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);
```

```cpp
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);
```

**Draft**
25.3 Sorting and related operations

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 $2 * ((last1 - first1) + (last2 - first2)) - 1$ comparisons.

Remarks: If \([first1, last1)\) contains \(m\) elements that are equivalent to each other and \([first2, last2)\) contains \(n\) elements that are equivalent to them, the first \(\min(m, n)\) elements shall be copied from the first range to the output range, in order.

25.3.5.4 set_difference

\[
\text{template}<\text{class InputIterator1, class InputIterator2, class OutputIterator}>
\]
\[
\text{OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);};
\]

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 $2 * ((last1 - first1) + (last2 - first2)) - 1$ 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

\[
\text{template}<\text{class InputIterator1, class InputIterator2, class OutputIterator}>
\]
\[
\text{OutputIterator set_symmetric_difference(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);};
\]
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);

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 * ((last1 - first1) + (last2 - first2)) - 1 comparisons.

Remarks: If [first1, last1) contains m elements that are equivalent to each other and [first2, last2) con- 
tains n elements that are equivalent to them, then |m - n| of those elements shall be copied to the output range: 
the last m - n of these elements from [first1, last1) if m > n, and the last n - m of these elements from 
[first2, last2) if m < n.

25.3.6 Heap operations [alg.heap.operations]

A heap is a particular organization of elements in a range between two random access iterators [a, b). Its two key 
properties are:

1. There is no element greater than *a in the range and
2. *a may be removed by pop_heap(), or a new element added by push_heap(), in $O(\log(N))$ time.

These properties make heaps useful as priority queues.

make_heap() converts a range into a heap and sort_heap() turns a heap into a sorted sequence.

25.3.6.1 push_heap [push.heap]

template<class RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last, 
Compare comp);

Effects: Places the value in the location last - 1 into the resulting heap [first, last).

Requires: The range [first, last - 1) shall be a valid heap. The type of *first shall satisfy the MoveConstructible 
requirements (Table 33) and the the MoveAssignable requirements (Table 35).

Accuracy: At most $\log(last - first)$ comparisons.
25.3 Sorting and related operations

25.3.6.2 pop_heap

```cpp
template<class RandomAccessIterator>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
             Compare comp);
```

**Effects:** Swaps the value in the location `first` with the value in the location `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.

25.3.6.3 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);
```

**Effects:** Constructs a heap out of the range `[first, last)`.

**Requires:** The type of `*first` shall satisfy the MoveConstructible requirements (Table 33) and the the MoveAssignable requirements (Table 35).

**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);
```

**Effects:** Sorts elements in the heap `[first, last)`.

**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 $N \log(N)$ comparisons (where $N == last - first$).
### 25.3.6.5 is_heap

**template<class RandomAccessIterator>**

```cpp
bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
```

*Returns:* `is_heap_until(first, last) == last`

**template<class RandomAccessIterator, class Compare>**

```cpp
bool is_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

*Returns:* `is_heap_until(first, last, comp) == last`

**template<class RandomAccessIterator>**

```cpp
RandomAccessIterator is_heap_until(RandomAccessIterator first, RandomAccessIterator last);
```

**template<class RandomAccessIterator, class Compare>**

```cpp
RandomAccessIterator is_heap_until(RandomAccessIterator first, RandomAccessIterator 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 a heap.

*Complexity:* Linear.

### 25.3.7 Minimum and maximum

**template<class T>**

```cpp
const T& min(const T& a, const T& b);
```

**template<class T, class Compare>**

```cpp
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>**

```cpp
 const T& min(const T& a, const T& b, const T& c);
```

**template<class T, class... Args>**

```cpp
 const T& min(const T& a, const Args&... args);
```

*Requires:* `T` is `LessThanComparable`, and all types forming `Args...` are the same as `T`.

*Returns:* The smallest value in the set of all the arguments.

*Remarks:* Returns the leftmost argument when several arguments are equivalent to the smallest. Returns `a` if `sizeof...(Args)` is 0.

**template<class T, class U, class... Args>**

```cpp
 const T& min(const T& a, const U& b, const Args&... args);
```

*Requires:* The types of all the arguments except the last one are the same as `T`. The last argument is a binary predicate over `T`. 
Returns: The first element in a partial ordering of all the arguments except the last one, where the ordering is defined by the predicate.

Remarks: Returns the leftmost argument when several arguments are equivalent to the first element in the ordering. Returns a if sizeof...(Args) is 0.

```cpp
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.

```cpp
template<class T> const T& max(const T& a, const T& b, const T& c);
template<class T, class... Args>
const T& max(const T& a, const Args&... args);
```

Requires: T is LessThanComparable, and all types forming Args... are the same as T.

Returns: The largest value in the set of all the arguments.

Remarks: Returns the leftmost argument when several arguments are equivalent to the largest. Returns a if sizeof...(Args) is 0.

```cpp
template<class T, class U, class... Args>
const T& max(const T& a, const U& b, const Args&... args);
```

Requires: The types of all the arguments except the last one are the same as T. The last argument is a binary predicate over T.

Returns: The last element in a partial ordering of all the arguments except the last one, where the ordering is defined by the predicate.

Remarks: Returns the leftmost argument when several arguments are equivalent to the first element in the ordering. Returns a if sizeof...(Args) is 0.

```cpp
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.

```cpp
template<class T> pair<const T&, const T&> minmax(const T& a, const T& b, const T& c);
template<class T, class... Args>
pair<const T&, const T&> minmax(const T& a, const Args&... args);
```
23 Requires: T is LessThanComparable, and all types forming Args... are the same as T.
24 Returns: pair<const T&, const T&>(x, y) where x is the first element and y is the last element in a partial ordering of all the arguments.
25 Remarks: x is the leftmost argument when several arguments are equivalent to the smallest. y is the rightmost argument when several arguments are equivalent to the largest. Returns pair<const T&, const T&>(a, a) if sizeof...(Args) is 0.
26 Complexity: At most \((3/2)\) sizeof...(Args) applications of the corresponding predicate.

```cpp
template<class T, class U, class... Args>
pair<const T&, const T&> minmax(const T& a, const U& b, const Args&... args);
```

27 Requires: The types of all the arguments except the last one are the same as T. The last argument is a binary predicate over T.
28 Returns: pair<const T&, const T&>(x, y) where x is the first element and y is the last element in a partial ordering of all the arguments except the last one, where the ordering is defined by the predicate.
29 Remarks: x is the leftmost argument when several arguments would order equivalent as first in the ordering. y is the rightmost argument when several arguments would order equivalent as last in the ordering. Returns pair<const T&, const T&>(a, a) if sizeof...(Args) is 0.
30 Complexity: At most \((3/2)\) sizeof...(Args) applications of the corresponding predicate.

```cpp
template<class ForwardIterator>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
```

31 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.
32 Complexity: Exactly max((last - first) - 1, 0) applications of the corresponding comparisons.

```cpp
template<class ForwardIterator, class Compare>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last, Compare comp);
```

33 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.
34 Complexity: Exactly max((last - first) - 1, 0) applications of the corresponding comparisons.

```cpp
template<class ForwardIterator>
pair<ForwardIterator, ForwardIterator> max_element(ForwardIterator first, ForwardIterator last);
```

35 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.
36 Complexity: Exactly max((last - first) - 1, 0) applications of the corresponding comparisons.

```cpp
template<class ForwardIterator, class Compare>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last, Compare comp);
```
25.3 Sorting and related operations

minmax_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
pair<ForwardIterator, ForwardIterator>
minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);

Returns: make_pair(m, M), where m is min_element(first, last) or min_element(first, last, comp) and M is max_element(first, last) or max_element(first, last, comp).

Complexity: At most max(2 * (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);

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.

Complexity: At most 2*min((last1 - first1), (last2 - first2)) applications of the corresponding comparison.

Remarks: If two sequences have the same number of elements and their corresponding elements are equivalent, then neither sequence is lexicographically less than the other. If one sequence is a prefix of the other, then the shorter sequence is lexicographically less than the longer sequence. Otherwise, the lexicographical comparison of the sequences yields the same result as the comparison of the first corresponding pair of elements that are not equivalent.

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 [alg.permutation.generators]

template<class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,
                    BidirectionalIterator last);

Draft
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);

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 105).

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>size_t</td>
</tr>
<tr>
<td>Functions:</td>
<td>bsearch  qsort</td>
</tr>
</tbody>
</table>

The contents are the same as the Standard C library header <stdlib.h> with the following exceptions:

The function signature:

```c
bsearch(const void *, const void *, size_t, size_t, 
    int (*)(const void *, const void *));
```

is replaced by the two declarations:
both of which have the same behavior as the original declaration.

4 The function signature:

```c
qsort(void *, size_t, size_t,
     int (*)(const void *, const void *));
```

is replaced by the two declarations:

```c
extern "C" void qsort(void* base, size_t nmemb, size_t size,
                       int (*compar)(const void*, const void*));
extern "C++" void qsort(void* base, size_t nmemb, size_t size,
                        int (*compar)(const void*, const void*));
```

both of which have the same behavior as the original declaration. The behavior is undefined unless the objects in the array pointed to by `base` are of trivial type.

[ Note: Because the function argument `compar()` may throw an exception, `bsearch()` and `qsort()` are allowed to propagate the exception (17.4.4.8). — 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-at-a-time) arrays, generalized numeric algorithms, and facilities included from the ISO C library, as summarized in Table 106.

Table 106: Numerics library summary

<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>&lt;complex&gt;</td>
</tr>
<tr>
<td>26.4 Random number generation</td>
<td>&lt;random&gt;</td>
</tr>
<tr>
<td>26.5 Numeric arrays</td>
<td>&lt;valarray&gt;</td>
</tr>
<tr>
<td>26.6 Generalized numeric operations</td>
<td>&lt;numeric&gt;</td>
</tr>
<tr>
<td>26.7 C library</td>
<td>&lt;cmath&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;ctgmath&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;tgmath.h&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cstdlib&gt;</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:

1. T is not an abstract class (it has no pure virtual member functions);
2. T is not a reference type;
3. T is not cv-qualified;
4. If T is a class, it has a public default constructor;
5. If T is a class, it has a public copy constructor with the signature T::T(const T&);
6. If T is a class, it has a public destructor;
7. If T is a class, it has a public assignment operator whose signature is either T& T::operator=(const T&) or
   T& T::operator=(T)

In other words, value types. These include arithmetic types, pointers, the library class complex, and instantiations of valarray for value types.

[numeric.requirements]
— 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 (23.2.6) 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<complex>, but operator() will not be successfully instantiated for valarray<complex> operands, since complex does not have any ordering operators. —end example]

26.2 The floating-point environment [cfenv]

26.2.1 Header <cfenv> synopsis [cfenv.syn]

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 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);
}

Draft
The header also defines the macros:

```cpp
FE_ALL_EXCEPT
FE_DIVBYZERO
FE_INEXACT
FE_INVALID
FE_OVERFLOW
FE_UNDERFLOW
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
26.3 Complex numbers

```
template<class T> complex<T> operator*(
    const complex<T>&, const complex<T>&);
template<class T> complex<T> operator*(const complex<T>&, const T&);
template<class T> complex<T> operator*(const T&, const complex<T>&);

template<class T> complex<T> operator/(const complex<T>&, const complex<T>&);
template<class T> complex<T> operator/(const complex<T>&, const T&);
template<class T> complex<T> operator/(const T&, const complex<T>&);

template<class T> complex<T> operator+(const complex<T>&);
template<class T> complex<T> operator-(const complex<T>&);

template<class T> 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, class charT, class traits>
    basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>&, complex<T>&);

template<class T, class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>&, const complex<T>&);
```

```
// 26.3.7 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>&);
```

Draft
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>&);

// Other complex arithmetic functions...

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&);

        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>&);
    };
}
26.3 Complex numbers

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
namespace std {
    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);
    }

    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);
    }
}
```

Draft
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 complex<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 rhs from the real part of the complex value *this and stores the result in the
real part of *this, leaving the imaginary part unchanged.

Returns: *this.

template <class T> complex<T>& operator*(const T& rhs);

Effects: Multiplies the scalar value rhs by the complex value *this and stores the result in *this.

Returns: *this.

template <class T> complex<T>& operator/=(const T& rhs);

Effects: Divides the scalar value rhs into the complex value *this and stores the result in *this.

Returns: *this.

template <class T> complex<T>& operator+=(const complex<T>& rhs);

Effects: Adds the complex value rhs to the complex value *this and stores the sum in *this.

Returns: *this.

template <class T> complex<T>& operator-=(const complex<T>& rhs);

Effects: Subtracts the complex value rhs from the complex value *this and stores the difference in *this.

Returns: *this.

template <class T> complex<T>& operator*=(const complex<T>& rhs);

Effects: Multiplies the complex value rhs by the complex value *this and stores the product in *this.

Returns: *this.

template <class T> complex<T>& operator/=(const complex<T>& rhs);

Effects: Divides the complex value rhs into the complex value *this and stores the quotient in *this.

Returns: *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 T& rhs);

Returns: complex<T>(lhs) *= rhs.

template<class T>
complex<T> operator/(const complex<T>& lhs, const complex<T>& rhs);

Returns: complex<T>(lhs) /= rhs.

template<class T>
bool operator==(const complex<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);

Returns: rhs.real() != lhs.real() || rhs.imag() != lhs.imag().

template<class T, class charT, class traits>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& is, complex<T>& x);

Effects: Extracts a complex number x of the form: u, (u), or (u,v), where u is the real part and v is the imaginary part (27.6.1.2).

Requires: The input values shall 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:

```cpp
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

```cpp
template<class T> T real(const complex<T>& x);

Returns: \( x \).real().
```

```cpp
template<class T> T imag(const complex<T>& x);

Returns: \( x \).imag().
```

```cpp
template<class T> T abs(const complex<T>& x);

Returns: the magnitude of \( x \).
```

```cpp
template<class T> T arg(const complex<T>& x);

Returns: the phase angle of \( x \), or \( \text{atan2}(\text{imag}(x), \text{real}(x)) \).
```

```cpp
template<class T> T norm(const complex<T>& x);

Returns: the squared magnitude of \( x \).
```

```cpp
template<class T> complex<T> conj(const complex<T>& x);

Returns: the complex conjugate of \( x \).
```

```cpp
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.
```

```cpp
template<class T> complex<T> polar(const T& rho, const T& theta = 0);

Returns: the complex value corresponding to a complex number whose magnitude is \( \rho \) and whose phase angle is \( \theta \).
```
26.3.8 complex transcendentals

```cpp
template<class T> complex<T> acos(const complex<T>& x);

Effects: Behaves the same as C99 function 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 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 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 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 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 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, imag(log(x)) is pi.

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).

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);
```

Draft
26.3 Complex numbers

```
template<class T> complex<T> pow (const T& x, const complex<T>& y);

Remarks: the branch cuts are along the negative real axis.

Returns: the complex power of base x raised to the y-th power, defined as exp(y*ln(x)). The value returned for pow(0,0) is implementation-defined.
```

```
template<class T> complex<T> sin (const complex<T>& x);

Returns: the complex sine of x.
```

```
template<class T> complex<T> sinh (const complex<T>& x);

Returns: the complex hyperbolic sine of x.
```

```
template<class T> complex<T> sqrt (const complex<T>& x);

Remarks: the branch cuts are along the negative real axis.

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);

Returns: the complex tangent of x.
```

```
template<class T> complex<T> tanh (const complex<T>& x);

Returns: the complex hyperbolic tangent of x.
```

26.3.9 Additional Overloads

The following function templates shall have additional overloads:

```
arg norm
conj polar
imag real
```

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>`.

Draft
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>`. [Note: `<complex.h>` does not promote any interface into the global namespace as there is no C interface to promote. —end note]

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_n` denotes a bitwise left shift with zero-valued bits appearing in the low bits of the result, and whose result is always taken modulo $2^n$.

26.4.1 Requirements

26.4.1.1 General requirements

Throughout this subclause 26.4, the effect of instantiating a template

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 107 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>
<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>Returns the least value potentially returned by operator().</td>
<td>compile-time</td>
</tr>
<tr>
<td>X::max()</td>
<td>T</td>
<td>Returns 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 108 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,

a) T is the type named by X’s associated result_type;

b) u is a value of X, v is an lvalue of X, x and y are (possibly const) values of X;
c) \( s \) is a value of arithmetic type (3.9.1);
d) \( q \) is an lvalue of type `seed_seq` (26.4.7.1);
e) \( z \) is a value of type `unsigned long long`;
f) \( os \) is an lvalue of the type of some class template specialization `basic_ostream<charT, traits>`, and
g) \( is \) is an lvalue of the type of some class template specialization `basic_istream<charT, traits>`;

where `charT` and `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, `seed` member function, `operator=`, or a suitable `operator>>`.

The specification of each random number engine defines the size of its state in multiples of the size of its `result_type`, given as an integral constant expression. The specification of each random number engine also defines

a) the `transition algorithm` \( TA \) by which the engine’s state \( x_i \) is advanced to its successor state \( x_{i+1} \), and

b) the `generation algorithm` \( GA \) by which an engine’s state is mapped to a value of type `result_type`.

Table 108: Random number engine requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X() )</td>
<td>—</td>
<td>Creates an engine with the same initial state as all other default-constructed engines of type ( X ).</td>
<td>( O(size of state) )</td>
</tr>
<tr>
<td>( X(x) )</td>
<td>—</td>
<td>Creates an engine that compares equal to ( x ).</td>
<td>( O(size of state) )</td>
</tr>
</tbody>
</table>
| \( X(s) \) | —           | Creates an engine with initial state determined by `static_cast<X::result_type>(s)`.
| \( X(q) \) | —           | Creates an engine \( u \) with an initial state that depends on a sequence produced by one call to `q.generate`. Same as complexity of `q.generate` when called on a sequence whose length is size of state. |
| \( u.seed() \) | void | post: \( u == X() \)
| \( u.seed(s) \) | void | post: \( u == X(s) \)
| \( u.seed(q) \) | void | post: \( u == X(q) \).
| \( u() \) | T           | Sets the state to \( u_{i+1} = TA(u_i) \) and returns \( GA(u_i) \). amortized constant |

\(^{270}\)This constructor (as well as the corresponding `seed()` function below) may be particularly useful to applications requiring a large number of independent random sequences.
26.4 Random number generation

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>u.discard(z)</code> (^{271})</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><code>x == y</code></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><code>x != y</code></td>
<td>bool</td>
<td>!((x == y))</td>
<td>(O(\text{sizeof state}))</td>
</tr>
<tr>
<td><code>os &lt;&lt; x</code></td>
<td>reference to the type of <code>os</code></td>
<td>With (os.fmtflags) set to (ios_base::dec|ios_base::left) and the fill character set to the space character, writes to (os) the textual representation of (x)'s current state. In the output, adjacent numbers are separated by one or more space characters. post: The (os.fmtflags) and fill character are unchanged.</td>
<td>(O(\text{sizeof state}))</td>
</tr>
<tr>
<td><code>is &gt;&gt; v</code></td>
<td>reference to the type of <code>is</code></td>
<td>With (is.fmtflags) set to (ios_base::dec), sets (v)'s state as determined by reading its textual representation from (is). If bad input is encountered, ensures that (v)'s state is unchanged by the operation and calls (is.setstate(ios::failbit)) (which may throw (ios::failure) (27.4.4.3)). pre: The textual representation was previously written using an (os) whose imbued locale and whose type’s template specialization arguments (\text{charT}) and (\text{traits}) were the same as those of (is). post: The (is.fmtflags) are unchanged.</td>
<td>(O(\text{sizeof state}))</td>
</tr>
</tbody>
</table>

\(^{271}\) 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 \(u()\).

\(^4\) \(X\) shall satisfy the requirements of uniform random number generator (26.4.1.2) as well as of \(\text{CopyConstructible}\) (34) and of \(\text{Assignable}\) (23.1). Copy construction and assignment shall each be of complexity \(O(\text{sizeof state})\).
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 109 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,

- `B_i` is the type of the `i`th of `X`’s base engines, `1 ≤ i ≤ n`; and
- `b_i` is a value of `B_i`.

If `X` is unary, `i` is omitted and understood to be 1.

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X::base_i_type</code></td>
<td><code>B_i</code></td>
<td>—</td>
<td>compile time</td>
</tr>
<tr>
<td><code>X::base_i()</code></td>
<td><code>const B_i&amp;</code></td>
<td>Returns a reference to <code>b_i</code></td>
<td>constant</td>
</tr>
</tbody>
</table>

`X` shall satisfy the requirements of random number engine (26.4.1.3), subject to the following:

- a) 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.
- b) The complexity of each function is at most the sum of the complexities of the corresponding functions applied to each base engine.
- c) 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`.
- d) The textual representation of `X` includes, in turn, the textual representation of each of its base engines.
- e) 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::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, ...`. When `X::X` is invoked with an argument of type `seed_seq`, each of `X`’s base engines is constructed, in turn, with that object as argument.

`X` shall have one additional constructor with `n` or more parameters such that the type of parameter `i`, `1 ≤ i ≤ n`, is `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

1 A class \( X \) satisfies the requirements of a random number distribution if the expressions shown in table 110 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 result_type;
- b) \( P \) is the type named by \( X \)'s associated param_type;
- c) \( u \) is a value of \( X \) and \( x \) is a (possibly const) value of \( X \);
- d) \( \text{glb} \) and \( \text{lub} \) are values of \( T \) respectively corresponding to the greatest lower bound and the least upper bound on the values potentially returned by \( u \)'s \( \text{operator}() \), 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) \( \text{os} \) is an lvalue of the type of some class template specialization basic_ostream<charT, traits>; and
- h) \( \text{is} \) is an lvalue of the type of some class template specialization basic_istream<charT, traits>;

where charT and traits are constrained according to 21 and 27.

2 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 110: Random number distribution requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X::\text{result_type} )</td>
<td>( T )</td>
<td>( T ) is an arithmetic type.</td>
<td>compile-time</td>
</tr>
<tr>
<td>( X::\text{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.\text{reset()} )</td>
<td>void</td>
<td>Subsequent uses of ( u ) do not depend on values produced by ( e ) prior to invoking ( \text{reset} ).</td>
<td>constant</td>
</tr>
<tr>
<td>( x.\text{param()} )</td>
<td>( P )</td>
<td>Returns a value ( p ) such that ( X(p).\text{param()} == p ).</td>
<td>no worse than the complexity of ( X(p) )</td>
</tr>
<tr>
<td>( u.\text{param}(p) )</td>
<td>void</td>
<td>post: ( u.\text{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><code>u(e)</code></td>
<td>T</td>
<td>With ( p = u.\text{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_i</td>
</tr>
<tr>
<td><code>u(e,p)</code></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_i</td>
</tr>
<tr>
<td><code>x.min()</code></td>
<td>T</td>
<td>Returns glb.</td>
<td>constant</td>
</tr>
<tr>
<td><code>x.max()</code></td>
<td>T</td>
<td>Returns lub.</td>
<td>constant</td>
</tr>
<tr>
<td><code>os &lt;&lt; x</code></td>
<td>reference to the type of <code>os</code></td>
<td>Writes to <code>os</code> a textual representation for the parameters and the additional internal data of <code>x</code>. post: The <code>os.fmtflags</code> and fill character are unchanged.</td>
<td>—</td>
</tr>
<tr>
<td><code>is &gt;&gt; u</code></td>
<td>reference to the type of <code>is</code></td>
<td>Restores from <code>is</code> the parameters and additional internal data of <code>u</code>. If bad input is encountered, ensures that <code>u</code> is unchanged by the operation and calls <code>is.setstate(ios::failbit)</code> (which may throw <code>ios::failure (27.4.4.3)</code>). pre: <code>is</code> provides a textual representation that was previously written using an <code>os</code> whose imbued locale and whose type’s template specialization arguments <code>charT</code> and <code>traits</code> were the same as those of <code>is</code>. post: The <code>is.fmtflags</code> 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` 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 \( X::\text{param\_type} \) is declared as a (nested) class or via a typedef. In this subclause 26.4, declarations of \( X::\text{param\_type} \) 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;

// 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;

```

Draft
// 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;
typedef see below default_random_engine;

// 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, 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;

// 26.4.8.3.2 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
template <class RealType = double>
class general_pdf_distribution;
}
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 108, 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 \( T(x_i) = (a \cdot x_i + c) \mod m \); the generation algorithm is \( G(x_i) = x_i + 1 \).

```
namespace std {
    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 constexpr result_type min() { return c == 0u ? 1u : 0u; }
        static constexpr result_type max() { return 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 \( \text{numeric_limits<result_type>::max()} + 1 \). [ 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 \( c < m \).

The textual representation consists of the value of \( x_i \).

```
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 $k = \lceil \frac{\log_2 m}{32} \rceil$ and $a$ an array (or equivalent) of length $k + 3$, invokes $q$.generate($a + 0, a + k + 3$) and then computes $S = \left( \sum_{j=0}^{k-1} a_{j+3} \cdot 2^{32j} \right) \mod m$. 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` [rand.eng.mers]

A `mersenne_twister_engine` random number engine\(^{272}\) 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 $X_{i+m-n} \text{xor} (Y \text{rshift} 1) \text{xor} \alpha$.

d) Let $z_4 = z_3 \text{xor} (z_3 \text{rshift} \ell)$.

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 $z_2 = z_1 \text{xor} ((z_1 \text{lshift} w) \text{bitand} b)$.

c) Let $z_3 = z_2 \text{xor} ((z_2 \text{lshift} w) \text{bitand} c)$.

namespace std {
    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;

            // engine characteristics
            static const size_t word_size = w;
            static const size_t state_size = n;
            static const size_t shift_size = m;

        ...

    }  

    }  

\(^{272}\) The name of this engine refers, in part, to a property of its period: For properly-selected values of the parameters, the period is closely related to a large Mersenne prime number.
static 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 constexpr result_type min() { return 0; }
static constexpr result_type max() { return $2^w - 1$; }
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: $1 \leq m \leq n$; $0 \leq r,u,s,t,1 \leq w \leq \text{numeric_limits<result_type>::digits}$; $0 \leq a,b,c \leq 2^w - 1$.
5 The textual representation of $x_i$ consists of the values of $X_{i-n}, \ldots, X_{i-1}$, in that order.

explicit mersenne_twister_engine(result_type value = default_seed);

Effects: Constructs a mersenne_twister_engine object. Sets $X_{n}$ to value mod $2^w$. Then, iteratively for $i = 1 - n, \ldots, -1$, sets $X_{i}$ to

$$[1812433253 \cdot (X_{i-1} \text{ xor } (X_{i-1} \text{ rshift } (w - 2))) + i \text{ mod } n] \text{ mod } 2^w.$$

Complexity: $O(n)$.

explicit mersenne_twister_engine(seed_seq& q);

Effects: Constructs a mersenne_twister_engine object. With $k = \lceil w/32 \rceil$ and $a$ an array (or equivalent) of length $n \cdot k$, invokes $q$.generate($a + 0$, $a + n \cdot k$) and then, iteratively for $i = -n, \ldots, -1$, sets $X_{i}$ to

$$\left(\sum_{j=0}^{k-1} a_{k(i+n)+j} \cdot 2^{32j}\right) \text{ mod } 2^w.$$ 

Finally, if the most significant $w - r$ bits of $X_{-n}$ are zero, and if each of the other resulting $X_{i}$ is 0, changes $X_{-n}$ to $2^w - 1$.

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 \texttt{subtract\_with\_carry\_engine} object \( x \) is of size \( \mathcal{O}(r) \), and consists of a sequence \( X \) of \( r \) integer values \( 0 \leq X_i < m = 2^w \); all subscripts applied to \( X \) are to be taken modulo \( r \). The state \( x_i \) additionally consists of an integer \( c \) (known as the carry) whose value is either 0 or 1.

The state transition is performed as follows:

a) Let \( Y = X_{i-s} - X_{i-r} - c \).

b) Set \( X_i \) to \( y = Y \mod m \). Set \( c \) to 1 if \( Y < 0 \), otherwise set \( c \) to 0.

[\textit{Note: This algorithm corresponds to a modular linear function of the form } TA(x_i) = (a \cdot x_i) \mod b, \text{ where } b \text{ is of the form } m^r - m^s + 1 \text{ and } a = b - (b - 1)/m. —\textit{end note}\]

The generation algorithm is given by \( GA(x_i) = y \), where \( y \) is the value produced as a result of advancing the engine’s state as described above.

```cpp
namespace std {
    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 constexpr result_type min() { return 0; }
        static constexpr result_type max() { return $m - 1$; }
        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 \( 0 < w \leq \text{numeric\_limits<result\_type>::digits} \).

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 \texttt{subtract\_with\_carry\_engine} object. Sets the values of \( X_{i-r}, \ldots, X_{i-1}, \) in that order, as required below. If \( X_{i-1} \) 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:

```cpp
class linear_congruential_engine{
  // definion...
}
```

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 \( \left( \sum_{j=0}^{n-1} z_j \cdot 2^{32j} \right) \mod m \). 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.generate\( (a + 0, \ a + r \cdot k) \) and then, iteratively for \( i = -r, \ldots, -1 \), sets \( X_i \) to \( \left( \sum_{j=0}^{k-1} a_{k(i+r)+j} \cdot 2^{32j} \right) \mod m \). If \( X_{-1} \) is then 0, sets \( c \) to 1; otherwise sets \( c \) to 0.

### 26.4.4 Random number engine adaptor class templates

Except where specified otherwise, the complexity of all functions specified in the following sections is constant.

Except as required by table 108, 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

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 \( e(\cdot) \) while advancing \( e \)'s state as described above.

```cpp
namespace std {
  template <class Engine, size_t p, size_t r>
  class discard_block_engine {
    public:
      // types
      typedef Engine base_type;
      typedef typename base_type::result_type result_type;

      // engine characteristics
      static const size_t block_size = p;

Draft
```
26.4 Random number generation  

static const size_t used_block = r;
static constexpr result_type min() { return base_type::min(); }
static constexpr result_type max() { return 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);
explicit discard_block_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
  int n;  // exposition only
};

4 The following relations shall hold: 1 \leq r \leq p.

5 The textual representation consists of the textual representation of e followed by the value of n.

6 In addition to its behavior pursuant to 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

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.

2 The transition and generation algorithms are described in terms of the following integral constants:

a) Let R = e.max() - e.min() + 1 and m = \lceil \log_2 R \rceil.

b) With n as determined below, let w_0 = \lfloor w/n \rfloor, n_0 = n - w \mod n, y_0 = 2^w_0 \lfloor R/2^w_0 \rfloor, and y_1 = 2^{w_0+1} \lfloor R/2^{w_0+1} \rfloor.

c) Let n = \lceil w/m \rceil if and only if the relation R - y_0 \leq \lfloor y_0/n \rfloor holds as a result. Otherwise let n = 1 + \lfloor w/m \rfloor.

[Note: The relation w = n_0w_0 + (n - n_0)(w_0 + 1) always holds. — end note]

3 The transition algorithm is carried out by invoking e() as often as needed to obtain n_0 values less than y_0 + e.min() and n - n_0 values less than y_1 + e.min().
The generation algorithm uses the values produced while advancing the state as described above to yield a quantity $S$ obtained as if by the following algorithm:

$$S = 0;$$
$$\text{for } (k = 0; k \neq n_0; k += 1) \{$$
$$\text{do } u = e() - e.\min(); \text{ while } (u \geq y_0);$$
$$S = 2^w \cdot S + u \mod 2^w;$$
$$\}$$
$$\text{for } (k = n_0; k \neq n; k += 1) \{$$
$$\text{do } u = e() - e.\min(); \text{ while } (u \geq y_1);$$
$$S = 2^{w+1} \cdot S + u \mod 2^{w+1};$$
$$\}$$

namespace std {
    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 constexpr result_type min() { return 0; }
            static constexpr result_type max() { return $2^w - 1$; }

            // constructors and seeding functions
            independent_bits_engine();
            explicit independent_bits_engine(const 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 $e$. Draft
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 \( k + 1 \).

1. The transition algorithm permutes the values produced by \( e \). The state transition is performed as follows:
   
   a) Calculate an integer \( j \) as \( \left\lfloor \frac{k(Y - b_{\min})}{b_{\max} - b_{\min} + 1} \right\rfloor \).
   
   b) Set \( Y \) to \( V_j \) and then set \( V_j \) to \( b() \).

2. The generation algorithm yields the last value of \( Y \) produced while advancing \( e \)'s state as described above.

```cpp
namespace std {
    template <class Engine, size_t k>
    class shuffle_order_engine {
    public:
        // types
        typedef Engine base_type;
        typedef typename base_type::result_type result_type;

        // engine characteristics
        static const size_t table_size = k;
        static constexpr result_type min() { return base_type::min(); }
        static constexpr result_type max() { return base_type::max(); }

        // constructors and seeding functions
        shuffle_order_engine();
        explicit shuffle_order_engine(const base_type& urng);
        explicit shuffle_order_engine(result_type s);
        explicit shuffle_order_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
        result_type Y;                   // exposition only
        result_type V[k];                // exposition only
    };
}
```
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], ..., 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_i and e2_i 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 \( G_A(x_i) = (v_1 \text{lshift}_w s1) \text{xor} (v_2 \text{lshift}_w s2) \), where \( w \) denotes the value of \text{numeric_limits<result_type>::digits} and \( v_1 \) and \( v_2 \), respectively, denote the values of \((e1()-e1.min())\) and \((e2()-e2.min())\).

```
namespace std {
    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 constexpr result_type min() { return 0; }
            static constexpr result_type max() { return \textit{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;
```
```cpp
const base2_type& base2() const;

private:
  base1_type e1; // exposition only
  base2_type e2; // exposition only
};

The following relations shall hold: s1 ≥ s2 ≥ 0.

[Note: An xor_combine_engine engine adaptor that fails to observe the following recommendations may have sig-
ificantly worse uniformity properties than either of the base engines it is based on:

a) While two shift values (template parameters s1 and s2) are provided for simplicity of interface, it is advisable
   that s2 be zero. (If s2 is non-zero then the low bits will always be 0.)

b) It is also advisable for e2's max() to be 2^n − 1 − min() for some non-negative integer n, and for the shift value
   s1 to be no greater than that n.
— end note]

Both Engine1::result_type and Engine2::result_type shall denote (possibly different) unsigned integral types. The member result_type shall denote either the type Engine1::result_type or the type Engine2::result_type, whichever provides the most storage according to clause 3.9.1.

With

a) \( m_1 = \min(\text{Engine1}:\max() - \text{Engine1}:\min(), 2^{w-s1} - 1) \),

b) \( m_2 = \min(\text{Engine2}:\max() - \text{Engine2}:\min(), 2^{w-s2} - 1) \), and

c) \( s = s1 - s2 \),

the value returned by the member max() is \( M(m_1, m_2, s) \mid^{w} s2 \), where \( M(a, b, d) \) is defined as follows:

If \( a = 0 \) or \( b < 2^d \), define \( M(a, b, d) = a \cdot 2^d + b \).

Otherwise, let \( r \) and \( u \) denote the greater and the lesser, respectively, of \( a \cdot 2^d \) and \( b \). With \( p = \lfloor \log_2 u \rfloor \), if \( k = \lfloor r/2^p \rfloor \) is odd, define \( M(a, b, d) = (k + 1) \cdot 2^p - 1 \).

Otherwise, if \( a \cdot 2^d \geq b \), define \( M(a, b, d) = (k + 1) \cdot 2^p + M((r \mod 2^p)/2^d, u \mod 2^p, d) \).

Otherwise, define \( M(a, b, d) = (k + 1) \cdot 2^p + M((u \mod 2^p)/2^d, t \mod 2^p, d) \).

The textual representation consists of the textual representation of e1 followed by the textual representation of e2.

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 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type minstd_rand shall produce the value 399268537.

typedef mersenne_twister_engine<uint_fast32_t, 32,624,397,31,0x9908b0df,11,7,0x9d2c5680,15,0xefc60000,18>
    mt19937;
3
    Required behavior: The 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type mt19937 shall produce the value 4123659995.

typedef subtract_with_carry_engine<uint_fast32_t, 24, 10, 24>
    ranlux24_base;
4
    Required behavior: The 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type ranlux24_base shall produce the value 7937952.

typedef subtract_with_carry_engine<uint_fast64_t, 48, 5, 12>
    ranlux48_base;
5
    Required behavior: The 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type ranlux48_base shall produce the value 6183912858275.

typedef discard_block_engine<ranlux24_base, 223, 23>
    ranlux24;
6
    Required behavior: The 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type ranlux24 shall produce the value 9901578.

typedef discard_block_engine<ranlux48_base, 389, 11>
    ranlux48;
7
    Required behavior: The 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type ranlux48 shall produce the value 249142670248501.

typedef shuffle_order_engine<minstd_rand0,256>
    knuth_b;
8
    Required behavior: The 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type knuth_b shall produce the value 1112339016.

typedef implementation-defined_default_random_engine;
9
    Required behavior: The named entity shall meet the requirements of a Random Number Engine (26.4.1.3).

10 The choice of engine type named by this typedef is implementation defined. [Note: The implementation may select this type on the basis of performance, size, quality, or any combination of such factors, so as to provide at least acceptable engine behavior for relatively casual, inexpert, and/or lightweight use. Because different implementations may select different underlying engine types, code that uses this typedef need not generate identical sequences across implementations. — end note]
26.4.6 Class random_device

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).

If implementation limitations prevent generating non-deterministic random numbers, the implementation may employ a random number engine.

```cpp
namespace std {
    class random_device {
        public:
            // types
            typedef unsigned int result_type;

            // generator characteristics
            static constexpr result_type min() { return \textit{see below}; }
            static constexpr result_type max() { return \textit{see below}; }

            // constructors
            explicit random_device(const string& token = implementation-defined);

            // generating functions
            result_type operator()();

            // property functions
            double entropy() const;

            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.

```cpp
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.\(^{273}\)

**Throws:** Nothing.\(^{274}\)

```cpp
double entropy() const;
```

**Returns:** If the implementation employs a random number engine, returns 0.0. Otherwise, returns an entropy estimate\(^{274}\) for the random numbers returned by `operator()`, in the range `min()` to `log_2(max() + 1)`.

**Throws:** Nothing.

\(^{273}\)The parameter is intended to allow an implementation to differentiate between different sources of randomness.

\(^{274}\)If a device has \(n\) states whose respective probabilities are \(P_0, \ldots, P_{n-1}\), the device entropy \(S\) is defined as \(S = -\sum_{i=0}^{n-1} P_i \cdot \log_2 P_i\).
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 \( \text{exception} \) if a random number could not be obtained.

26.4.7 Utilities

26.4.7.1 Class seed_seq

An object of type \( \text{seed_seq} \) 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 \( \text{seed_seq} \) shall meet the requirements of \( \text{CopyConstructible} \) (34) and of \( \text{Assignable} \) (23.1).

namespace std {
    class seed_seq {
    public:
    // types
    typedef uint_least32_t result_type;

    // constructors
    seed_seq();
    template<class InputIterator, size_t u = numeric_limits<iterator_traits<InputIterator>::value_type>::digits>
    seed_seq(InputIterator begin, InputIterator end);

    // generating functions
    template<class RandomAccessIterator>
    void generate(RandomAccessIterator begin, RandomAccessIterator end) const;

    // property functions
    size_t size() const;
    template<class OutputIterator> void param(OutputIterator dest) const;

    private:
    vector<result_type> v; // exposition only
    }
}

explicit seed_seq();

Effects: Constructs a \( \text{seed_seq} \) object as if by default-constructing its member \( v \).

Throws: Nothing.
26.4 Random number generation

template<class InputIterator,
         size_t u = numeric_limits<iterator_traits<InputIterator>::value_type>::digits>
seed_seq(InputIterator begin, InputIterator end);

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 some or all of the bits of the supplied sequence [begin,end) of \( w \)-bit quantities into 32-bit units, as if by the following:

First extract the rightmost \( u \) bits from each of the \( n = \text{end} - \text{begin} \) elements of the supplied sequence and concatenate all the extracted bits to initialize a single (possibly very large) unsigned binary number, 

\[
b = \sum_{i=0}^{n-1} (\text{begin}[i]) \mod 2^u \cdot 2^{w-i}\]

(in which the bits of each \( \text{begin}[i] \) are treated as denoting an unsigned quantity). Then carry out the following algorithm:

\[
v.clear(); \quad \text{if} \quad (w < 32) \quad v.push_back(w); \quad \text{for} \quad (; \quad w > 0; \quad --w) \quad v.push_back(b \mod 2^{32}), \quad b /= 2^{32};
\]

```cpp
template<class RandomAccessIterator>
void generate(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: Does nothing if \( \text{begin} == \text{end} \). Otherwise, with \( s = v.size() \) 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 \( n \), and \( T(x) \) is defined as \( x \oplus (x \ll 27) \):

a) By way of initialization, set each element of the range to the value \( 0x8b8b8b8b \). Additionally, for use in subsequent steps, let \( p = (n-t)/2 \) and let \( q = p + t \), where

\[
t = (n \geq 623) \times 11 : (n \geq 68) \times 7 : (n \geq 39) \times 5 : (n \geq 7) \times 3 : (n-1)/2;
\]

b) With \( m \) as the larger of \( s + 1 \) and \( n \), transform the elements of the range: iteratively for \( k = 0, \ldots, m - 1 \), calculate values

\[
\begin{align*}
    r_1 &= 1664525 \cdot T(\text{begin}[k] \oplus \text{begin}[k+p] \oplus \text{begin}[k-1]) \\
    r_2 &= r_1 + \begin{cases} 
    s & , k = 0 \\
    k \mod n + v[k-1], & 0 < k <= s \\
    k \mod n, & s < k
    \end{cases}
\end{align*}
\]

and, in order, increment \( \text{begin}[k+p] \) by \( r_1 \), increment \( \text{begin}[x+q] \) by \( r_2 \), and set \( \text{begin}[k] \) to \( r_2 \).

c) Transform the elements of the range three more times, beginning where the previous step ended: iteratively for \( k = m, \ldots, m+n-1 \), calculate values
\[ r_3 = 1566083941 \cdot T(\text{begin}[k] + \text{begin}[k+p] + \text{begin}[k-1]) \]
\[ r_4 = r_3 - (k \mod n) \]

and, in order, update \( \text{begin}[k+p] \) by xoring it with \( r_4 \), update \( \text{begin}[k+q] \) by xoring it with \( r_3 \), and set \( \text{begin}[k] \) to \( r_4 \).

9 \hspace{1em} \text{Throws: Nothing.}

size_t size() const;

10 \hspace{1em} \text{Returns: The number of 32-bit units that would be returned by a call to \text{param}().}

template<class OutputIterator> void param(OutputIterator dest) const;

11 \hspace{1em} \text{Requires: OutputIterator shall satisfy the requirements of an output iterator (24.1.2) such that \text{iterator_traits<OutputIterator>::value_type} shall be assignable from result_type.}

12 \hspace{1em} \text{Effects: Copies the sequence of prepared 32-bit units to the given destination, as if by executing the following statement:}

\[
\text{copy(v.begin(), v.end(), dest);}
\]

26.4.7.2 Function template generate_canonical \[\text{[rand.util.canonical]}\]

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.

2 \hspace{1em} \text{[Note: Obtaining a value in this way can be a useful step in the process of transforming a value generated by a uniform random number generator into a value that can be delivered by a random number distribution. – end note]} \]

\[
\text{template<class RealType, size_t bits, class UniformRandomNumberGenerator>}
\]

\[
\text{RealType generate_canonical(UniformRandomNumberGenerator& g);}\]

3 \hspace{1em} \text{Complexity: Exactly} \( k = \max(1, \lceil b/\log_2 R \rceil) \) \text{invocations of} \( g \), \text{where} \( b^{275} \) \text{is the lesser of} \text{numeric_limits<RealType>::digits} \text{and} \text{bits}, \text{and} \( R \) \text{is the value of} g.\text{max} - g.\text{min} + 1.

4 \hspace{1em} \text{Required behavior: Invokes} g() \text{\( k \) times to obtain values} g_0, \ldots, g_{k-1}, \text{respectively. Calculates a quantity}

\[
S = \sum_{i=0}^{k-1} (g_i - g.\text{min}) \cdot R^i
\]

\text{using arithmetic of type} \text{RealType}.

5 \hspace{1em} \text{Returns:} \( S/R^k \).

6 \hspace{1em} \text{Throws:} \text{What and when} g \text{throws.}

\[275\] \text{\( b \) is introduced to avoid any attempt to produce more bits of randomness than can be held in \text{RealType}.}
26.4 Random number generation Numerics library

26.4.8 Random number distribution class templates [rand.dist]

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 [rand.dist.uni]

26.4.8.1.1 Class template uniform_int_distribution [rand.dist.uni.int]

A uniform_int_distribution random number distribution produces random integers \( i, a \leq i \leq b \), distributed according to the constant discrete probability function

\[
    P(i|a,b) = \frac{1}{b-a+1}.
\]

namespace std {
    template <class IntType = int>
    class uniform_int_distribution {
    public:
        // types
        typedef IntType result_type;
        typedef unspecified param_type;

        // constructors and reset functions
        explicit uniform_int_distribution(IntType a = 0, IntType b = numeric_limits<IntType>::max());
        explicit uniform_int_distribution(const param_type& parm);
        void reset();

        // generating functions
        template <class 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_int_distribution(IntType a = 0, IntType b = numeric_limits<IntType>::max());

Draft
2 \textit{Requires:} \(a \leq b\).
3 \textit{Effects:} Constructs a \texttt{uniform\_int\_distribution} object; \(a\) and \(b\) correspond to the respective parameters of the distribution.

\begin{verbatim}
result_type a() const;
\end{verbatim}
4 \textit{Returns:} The value of the \(a\) parameter with which the object was constructed.

\begin{verbatim}
result_type b() const;
\end{verbatim}
5 \textit{Returns:} The value of the \(b\) parameter with which the object was constructed.

### 26.4.8.1.2 Class template \texttt{uniform\_real\_distribution} [rand.dist.uni.real]

A \texttt{uniform\_real\_distribution} random number distribution produces random numbers \(x, a \leq x < b\), distributed according to the constant probability density function

\[ p(x|a,b) = \frac{1}{b-a}. \]

```
namespace std {
    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;
    };
}
```

\begin{verbatim}
explicit uniform_real_distribution(RealType a = 0.0, RealType b = 1.0);
\end{verbatim}
26.4 Random number generation Numerics library

2

Requires: \( a \leq b \) and \( b - a \leq \text{numeric\_limits<\text{RealType}>::max()} \).

3

Effects: Constructs a uniform\_real\_distribution object; \( a \) and \( b \) correspond to the respective parameters of the distribution.

\[
\text{result\_type } a() \text{ const;}
\]

4

Returns: The value of the \( a \) parameter with which the object was constructed.

\[
\text{result\_type } b() \text{ const;}
\]

5

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

1 A bernoulli\_distribution random number distribution produces bool values \( b \) distributed according to the discrete probability function

\[
P(b | p) = \begin{cases} 
p & \text{if } b = \text{true} \\
1 - p & \text{if } b = \text{false} 
\end{cases}
\]

namespace std {
    class bernoulli\_distribution {
        public:
            // types
            typedef bool result\_type;
            typedef unspecified param\_type;

            // constructors and reset functions
            explicit bernoulli\_distribution(double p = 0.5);
            explicit bernoulli\_distribution(const param\_type& parm);
            void reset();

            // generating functions
            template <class 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);
\texttt{bernoulli_distribution} object; \( p \) corresponds to the parameter of the distribution.

\begin{verbatim}
double p() const;
\end{verbatim}

\textit{Returns:} The value of the \( p \) parameter with which the object was constructed.

\subsubsection{26.4.8.2.2 Class template binomial_distribution [rand.dist.bern.bin]}

A \texttt{binomial_distribution} random number distribution produces integer values \( i \geq 0 \) distributed according to the discrete probability function

\[ P(i|t,p) = \binom{t}{i} \cdot p^i \cdot (1-p)^{t-i}. \]

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

\begin{verbatim}
    // Requires: 0 \leq p \leq 1.
    // Effects: Constructs a \texttt{binomial_distribution} object; \( p \) corresponds to the parameter of the distribution.
\end{verbatim}

\begin{verbatim}
    double p() const;
\end{verbatim}

\textit{Returns:} The value of the \( p \) parameter with which the object was constructed.

\textit{Effects:} Constructs a \texttt{binomial_distribution} object; \( t \) and \( p \) correspond to the respective parameters of the distribution.
26.4 Random number generation

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

\[
P(i \mid p) = p \cdot (1 - p)^i.
\]

namespace std {
    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>
        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);

Requires: \( 0 < p < 1 \).

Effects: Constructs a geometric_distribution object; p corresponds to the parameter of the distribution.

double p() const;

Returns: The value of the p parameter with which the object was constructed.
A negative_binomial_distribution random number distribution produces random integers \( i \geq 0 \) distributed according to the discrete probability function

\[
P(i|k,p) = \binom{k+i-1}{i} \cdot p^k \cdot (1-p)^i.
\]

```cpp
namespace std {
    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;
        double p() const;
        param_type param() const;
        void param(const param_type& parm);
        result_type min() const;
        result_type max() const;
    };
}
```

```cpp
explicit negative_binomial_distribution(IntType k = 1, double p = 0.5);
```

\textbf{Requires}: \( 0 < p \leq 1 \) and \( 0 < k \).

\textbf{Effects}: Constructs a negative_binomial_distribution object; \( k \) and \( p \) correspond to the respective parameters of the distribution.

```cpp
IntType k() const;
```

\textbf{Returns}: The value of the \( k \) parameter with which the object was constructed.

```cpp
double p() const;
```

\textbf{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

A poisson_distribution random number distribution produces integer values \( i \geq 0 \) distributed according to the discrete probability function

\[
P(i|\mu) = \frac{e^{-\mu} \mu^i}{i!}
\]

The distribution parameter \( \mu \) is also known as this distribution’s mean.

```cpp
namespace std {
    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: \( 0 < \text{mean} \).

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

\[ p(x | \lambda) = \lambda e^{-\lambda x}. \]

namespace std {
    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);

\textit{Requires:} 0 < \lambda.

\textit{Effects:} Constructs an exponential_distribution object; \lambda corresponds to the parameter of the distribution.

RealType lambda() const;

\textit{Returns:} The value of the lambda parameter with which the object was constructed.

26.4.8.3.3 \textbf{Class template} gamma_distribution \hfill [rand.dist.pois.gamma]

A gamma_distribution random number distribution produces random numbers \( x > 0 \) distributed according to the probability density function

\[ p(x | \alpha, \beta) = \frac{e^{-x/\beta}}{\beta^\alpha \Gamma(\alpha)} \cdot x^{\alpha-1}. \]
26.4 Random number generation

namespace std {
    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);

2  Requires: 0 < alpha and 0 < beta.

3  Effects: Constructs a gamma_distribution object; alpha and beta correspond to the parameters of the distribution.

RealType alpha() const;

4  Returns: The value of the alpha parameter with which the object was constructed.

RealType beta() const;

5  Returns: The value of the beta parameter with which the object was constructed.

26.4.8.3.4 Class template weibull_distribution

A weibull_distribution random number distribution produces random numbers \( x \geq 0 \) distributed according to the probability density function

\[
p(x|a,b) = \frac{a}{b} \cdot \left(\frac{x}{b}\right)^{a-1} \cdot \exp\left(-\left(\frac{x}{b}\right)^a\right).
\]

Draft
namespace std {
    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);

// Requires: 0 < a and 0 < b.

// Effects: Constructs a weibull_distribution object; a and b correspond to the respective parameters of the
distribution.

RealType a() const;

// Returns: The value of the a parameter with which the object was constructed.

RealType b() const;

// Returns: The value of the b parameter with which the object was constructed.

26.4.8.3.5 Class template extreme_value_distribution

An extreme_value_distribution random number distribution produces random numbers x distributed according to
the probability density function

\[ p(x|a,b) = \frac{1}{b} \cdot \exp \left( \frac{a - x}{b} - \exp \left( \frac{a - x}{b} \right) \right). \]

namespace std {
    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;
    }
}

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.

---

276) 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.
26.4.8.4 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**.

```cpp
class normal_distribution {
  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;
  };

  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;

  // Additional methods...
};
```
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

\[
p(x|m,s) = \frac{1}{sx\sqrt{2\pi}} \exp\left( -\frac{(\ln x - m)^2}{2s^2} \right).
\]

```cpp
class lognormal_distribution {
public:
    typedef RealType result_type;
    typedef unspecified param_type;

    explicit lognormal_distribution(RealType m = 0.0, RealType s = 1.0);
    explicit lognormal_distribution(const param_type& parm);
    void reset();

    template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    RealType m() const;
    RealType s() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

explicit lognormal_distribution(RealType m = 0.0, RealType s = 1.0);

Requires: \( 0 < s \).

Effects: Constructs a lognormal_distribution object; \( m \) and \( s \) correspond to the respective parameters of the distribution.

Returns: The value of the \( m \) parameter with which the object was constructed.
 Returns: The value of the $n$ 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.

```cpp
namespace std {
  template <class RealType = double>
  class chi_squared_distribution {
  public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    explicit chi_squared_distribution(int n = 1);
    explicit chi_squared_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 chi_squared_distribution(int n = 1);
```

 Requires: $0 < n.$

 Effects: Constructs a `chi_squared_distribution` object; $n$ corresponds to the parameter of the distribution.

 int n() const;

 Returns: The value of the $n$ parameter with which the object was constructed.
26.4 Random number generation

26.4.8.4.4 Class template cauchy_distribution

A cauchy_distribution random number distribution produces random numbers \( x \) distributed according to the probability density function

\[
p(x|a,b) = \left( \pi b \left( 1 + \left( \frac{x-a}{b} \right)^2 \right) \right)^{-1}.
\]

namespace std {
    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);

Requires: \( 0 < b \).

Effects: Constructs a cauchy_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.
26.4.8.4.5 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

\[
p(x|m,n) = \frac{1}{\Gamma(m/2) \Gamma(n/2)} \left( \frac{m}{n} \right)^{m/2} \cdot x^{(m/2)-1} \cdot \left( 1 + m \cdot \frac{x}{n} \right)^{-(m+n)/2},
\]

where \( m \) and \( n \) are positive integers.

```cpp
namespace std {
    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.
A `student_t_distribution` random number distribution produces random numbers \( x \) distributed according to the probability density function

\[
p(x|n) = \frac{1}{\sqrt{n\pi}} \frac{\Gamma((n+1)/2)}{\Gamma(n/2)} \left(1 + \frac{x^2}{n}\right)^{-(n+1)/2},
\]

where \( n \) is a positive integer.

```cpp
namespace std {
    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.

```cpp
int n() const;
```

**Returns:** The value of the \( n \) parameter with which the object was constructed.
A discrete_distribution random number distribution produces random integers $i$, $0 \leq i < n$, distributed according to the discrete probability function

$$P(i | p_0, \ldots, p_{n-1}) = p_i.$$  

namespace std {
    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;
    };
}

discrete_distribution();  

Effects: Constructs a discrete_distribution object with $n = 1$ and $p_0 = 1$. [Note: Such an object will always deliver the value 0. —end note]

template <class InputIterator>
discrete_distribution(InputIterator firstW, InputIterator lastW);

Requires:

a) InputIterator shall satisfy the requirements of an input iterator (24.1.1).
b) If \( \text{firstW} == \text{lastW} \), let the sequence \( w \) have length \( n = 1 \) and consist of the single value \( w_0 = 1 \). Otherwise, \([\text{firstW}, \text{lastW})\) shall form a sequence \( w \) of length \( n > 0 \) and \(*\text{firstW}\) 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

\[
p_k = \frac{w_k}{S} \text{ for } k = 0, \ldots, n-1.
\]

```
vectordouble probabilities() const;
```

Returns: A vector<double> whose size member returns \( n \) and whose operator[] member returns \( p_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n-1 \).

### 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
define std {
    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) [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 $n$, *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

$$\rho_k = \frac{w_k}{S \cdot (b_{k+1} - b_k)} \text{ for } k = 0, \ldots, n - 1.$$ 

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 $k = 0, \ldots, n - 1$. 

Draft
A general_pdf_distribution random number distribution produces random numbers \( x \), \( x_{\text{min}} \leq x < x_{\text{max}} \), distributed according to the probability density function

\[
p(x | x_{\text{min}}, x_{\text{max}}, \rho) = \rho(x) \quad \text{for } x_{\text{min}} \leq x < x_{\text{max}}.
\]

Effects: Constructs a general_pdf_distribution object with \( x_{\text{min}} = 0 \) and \( x_{\text{max}} = 1 \) such that \( p(x) = 1 \) for all \( x_{\text{min}} \leq x < x_{\text{max}} \).

Requires:

a) pdf shall be callable with one argument of type result_type, and shall return values of a type convertible to double;

Draft
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 pdf. [Note: This implies that the user-supplied pdf need not be normalized. — end note]

**Effects:** Constructs a `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 \( \rho(x) = f(x)/z \).

```cpp
template<typename T>
result_type xmin() const;
```

**Returns:** The value of the \( x_{\text{min}} \) parameter with which the object was constructed.

```cpp
template<typename T>
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 `<valarray>` synopsis

```cpp
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 indrected 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>&);

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&);
```

Draft
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 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<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>&);
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.\(^{277}\)

Implementations introducing such replacement types shall provide additional functions and operators as follows:

\(^{277}\) 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.
— 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` (18.5.2.1) 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>&);
        ~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;
```

Draft
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 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>&);
valarray<T>&
operator&= (const valarray<T>&);
valarray<T>&
operator<<= (const valarray<T>&);
valarray<T>&
operator>>= (const valarray<T>&);

// 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());
};

Draft
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.\(^{278}\)

An implementation is permitted to qualify any of the functions declared in `<valarray>` as inline.

### 26.5.2.1 `valarray` constructors

```cpp
valarray();

Effects: Constructs an object of class `valarray<T>`\(^{279}\) which has zero length until it is passed into a library function as a modifiable lvalue or through a non-constant `this` pointer.\(^{280}\)

\[\text{explicit valarray(size_t);}\]

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`.

\[\text{valarray(const T&, size_t);}\]

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.

\[\text{valarray(const T*, size_t);}\]

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.\(^{281}\) If the value of the second argument is greater than the number of values pointed to by the first argument, the behavior is undefined.

\[\text{valarray(const valarray<T>&);}\]

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.\(^{282}\)

\[\text{valarray(valarray<T>&&);}\]

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.

---

\(^{278}\) 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.

\(^{279}\) For convenience, such objects are referred to as “arrays” throughout the remainder of 26.5.

\(^{280}\) This default constructor is essential, since arrays of `valarray` are likely to prove may be useful. There shall also be a way to change the size of an array after initialization; this is supplied by the semantics of `The length of an empty array can be increased after initialization with the resize member function.`

\(^{281}\) This constructor is the preferred method for converting a C array to a `valarray` object.

\(^{282}\) This copy constructor creates a distinct array rather than an alias. Implementations in which arrays share storage are permitted, but they shall implement a copy-on-reference mechanism to ensure that arrays are conceptually distinct.

Draft
These conversion constructors convert one of the four reference templates to a `valarray`.

```
~valarray();
```

The destructor is applied to every element of `*this`; an implementation may return all allocated memory.

### 26.5.2.2 `valarray` assignment

```
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.

```
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>&);
```

**Requires:** The length of the array to which the argument refers equals `size()`.

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

```
const T& operator[](size_t) const;
T& operator[](size_t);
```
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.

Thus, the expression \((a[i] = q, a[i]) == q\) evaluates as true for any non-constant \(\text{valarray}<T>\) \(a\), any \(T\) \(q\), and for any \(\text{size}_t\) \(i\) such that the value of \(i\) is less than the length of \(a\).

The expression \&\(a[i+j] == \&a[i] + j\) evaluates as true for all \(\text{size}_t\) \(i\) and \(\text{size}_t\) \(j\) such that \(i+j\) is less than the length of the non-constant array \(a\).

Likewise, the expression \&\(a[i] != \&b[j]\) evaluates as true for any two non-constant arrays \(a\) and \(b\) and for any \(\text{size}_t\) \(i\) and \(\text{size}_t\) \(j\) 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.

The reference returned by the subscript operator for a non-constant array is guaranteed to be valid until the member function \(\text{resize}()\) \((26.5.2.7)\) is called for that array or until the lifetime of that array ends, whichever happens first.

If the subscript operator is invoked with a \(\text{size}_t\) argument whose value is not less than the length of the array, the behavior is undefined.

### 26.5.2.4 \text{valarray} subset operations

\[
\begin{align*}
\text{valarray}<T> & \quad \text{operator[]}(\text{slice}) \text{ const}; \\
\text{slice_array}<T> & \quad \text{operator[]}(\text{slice}) \text{ const}; \\
\text{valarray}<T> & \quad \text{operator[]}(\text{const gslice&}) \text{ const}; \\
\text{gslice_array}<T> & \quad \text{operator[]}(\text{const gslice&}) \text{ const}; \\
\text{valarray}<T> & \quad \text{operator[]}(\text{const valarray<bool>&}) \text{ const}; \\
\text{mask_array}<T> & \quad \text{operator[]}(\text{const valarray<bool>&}); \\
\text{valarray}<T> & \quad \text{operator[]}(\text{const valarray<\text{size}_t&}) \text{ const}; \\
\text{indirect_array}<T> & \quad \text{operator[]}(\text{const valarray<\text{size}_t&}); \\
\end{align*}
\]

Each of these operations returns a subset of the array. The \(\text{const}\)-qualified versions return this subset as a new \text{valarray}. The \(\text{non-const}\) versions return a class template object which has reference semantics to the original array.

### 26.5.2.5 \text{valarray} unary operators

\[
\begin{align*}
\text{valarray}<T> & \quad \text{operator+() const}; \\
\text{valarray}<T> & \quad \text{operator-() const}; \\
\text{valarray}<T> & \quad \text{operator~() const}; \\
\text{valarray<bool>} & \quad \text{operator!() const}; \\
\end{align*}
\]

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 \(T\) \((\text{bool for operator!})\) or which may be unambiguously implicitly converted to type \(T\) \((\text{bool for operator!})\).

---

283) Compilers may take advantage of inlining, constant propagation, loop fusion, tracking of pointers obtained from \text{operator new}, and other techniques to generate efficient \text{valarrays}.
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

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<T>& operator*= (const T&);
valarray<T>& operator/= (const T&);
valarray<T>& operator%= (const T&);
valarray<T>& operator+= (const T&);
valarray<T>& operator-= (const T&);
valarray<T>& operator^= (const T&);
valarray<T>& operator&= (const T&);
valarray<T>& operator|= (const T&);
valarray<T>& operator<<=(const T&);
valarray<T>& operator>>=(const T&);

Each of these operators may only be instantiated for a type T to which the indicated operator can be applied.

Each of these operators applies the indicated operation to each element of the array and the non-array argument.

The array is then returned by reference.

The appearance of an array on the left-hand side of a computed assignment does not invalidate references or pointers to the elements of the array.

26.5.2.7 valarray member functions
void swap(valarray&& v);

*Effects:* 
- `*this` obtains the value of `v`.
- `v` obtains the value of `*this`.

*Complexity:* Constant.

*Throws:* Nothing.

size_t size() const;

This function returns the number of elements in the array.

T sum() const;

This function may only be instantiated for a type `T` to which `operator+=` can be applied. This function returns the sum of all the elements of the array.

If the array has length 0, the behavior is undefined. If the array has length 1, `sum()` returns the value of element 0. Otherwise, the returned value is calculated by applying `operator+=` to a copy of an element of the array and all other elements of the array in an unspecified order.

T min() const;

This function returns the minimum value contained in `*this`. The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using `operator<`.

T max() const;

This function returns the maximum value contained in `*this`. The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using `operator<`.

valarray<T> shift(int n) const;

This function returns an object of class `valarray<T>` of length `size()`, each of whose elements `I` is `(*this)[I + n]` if `I + n` is non-negative and less than `size()`, otherwise `T()`. Thus if element zero is taken as the leftmost element, a positive value of `n` shifts the elements left `n` places, with zero fill.

[Example: If the argument has the value -2, the first two elements of the result will be 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 ]

valarray<T> cshift(int n) const;

This function returns an object of class `valarray<T>`, of length `size()`, each of whose elements `I` is `(*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.

valarray<T> apply(T func(T)) const;

valarray<T> apply(T func(const T&)) const;

These functions return an array whose length is equal to the array. Each element of the returned array is assigned the value returned by applying the argument function to the corresponding element of the array.
void resize(size_t sz, T c = T());

This member function changes the length of the *this array to sz and then assigns to each element the value of the second argument. Resizing invalidates all pointers and references to elements in the array.

26.5.3 valarray non-member operations [valarray.nonmembers]

26.5.3.1 valarray binary operators [valarray.binary]

template<class T> valarray<T> operator* (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator/ (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator% (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator+ (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator- (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator^ (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator& (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator| (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator<< (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator>> (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator* (const valarray<T>&, const T&);

template<class T> valarray<T> operator* (const T&, const valarray<T>&);

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 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 T or which can be unambiguously implicitly converted to type T.

Each of these operators returns an array whose length is equal to the lengths of the argument arrays. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays.

If the argument arrays do not have the same length, the behavior is undefined.
26.5 Numeric arrays

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 implicitly 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>&);

Each of these operators may only be instantiated for a type T to which the indicated operator can be applied and for which the indicated operator returns a value which is of type bool or which can be unambiguously implicitly converted to type bool.

Each of these operators returns a bool array whose length is equal to the length of the array arguments. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays.

If the two array arguments do not have the same length, the behavior is undefined.
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>&);

Each of these operators may only be instantiated for a type \( T \) to which the indicated operator can be applied and for which the indicated operator returns a value which is of type bool or which can be unambiguously implicitly converted to type bool.

Each of these operators returns a bool array whose length is equal to the length of the array 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 transcendental

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> atanh (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>&);
26.5 Numeric arrays

Each of these functions may only be instantiated for a type T to which a unique function with the indicated name can be applied (unqualified). This function shall return a value which is of type T or which can be unambiguously implicitly converted to type T.

26.5.3.4 valarray specialized algorithms

```cpp
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);
```

Effects: \( x\).swap(\( y \)).

26.5.4 Class slice

```cpp
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;
        }
    }
}
```

The `slice` class represents a BLAS-like slice from an array. Such a slice is specified by a starting index, a length, and a stride.\(^\text{264}\)

26.5.4.1 slice constructors

```cpp
slice();
slice(size_t start, size_t length, size_t stride);
slice(const slice&);
```

The default constructor is equivalent to `slice(0, 0, 0)`. A default constructor is provided only to permit the declaration of arrays of slices. The constructor with arguments for a slice takes a start, length, and stride parameter.

[Example: `slice(3, 8, 2)` constructs a slice which selects elements 3, 5, 7, ... 17 from an array. — end example]

26.5.4.2 slice access functions

\(^\text{264}\) BLAS stands for Basic Linear Algebra Subprograms. C++ programs may instantiate this class. See, for example, Dongarra, Du Croz, Duff, and Hamnerling: 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.
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();
        const slice_array& operator=(const slice_array&) const;
        void operator=(const T&) const;
    slice_array() = delete;  // as implied by declaring copy constructor above
    
};
}  

The slice_array template is a helper template used by the slice subscript operator

slice_array<T> valarray<T>::operator[](slice);

It has reference semantics to a subset of an array specified by a slice object.

[ Example: The expression a[slice(1, 5, 3)] = b; has the effect of assigning the elements of b to a slice of the elements in a. For the slice shown, the elements selected from a are 1, 4, ..., 13. — end example ]

26.5.5.1 slice_array assignment

void operator=(const valarray<T>&) const;

const slice_array& operator=(const slice_array&) const;

These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the valarray<T> object to which the slice_array object refers.
26.5 Numeric arrays

26.5.5.2 slice_array computed assignment

```c++
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 slice_array object refers.

26.5.5.3 slice_array fill function

```c++
void operator=(const T&) const;
```

This function has reference semantics, assigning the value of its argument to the elements of the valarray<T> object to which the slice_array object refers.

26.5.6 The gslice class

```c++
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_j - 1.

[Example: The gslice specification]
start = 3  
length = {2, 4, 3}  
stride = {19, 4, 1}  

yields the sequence of one-dimensional indices 

\[ k = 3 + (0, 1) \times 19 + (0, 1, 2, 3) \times 4 + (0, 1, 2) \times 1 \]

which are ordered as shown in the following table:

\[
(\hat{i}_0, \ \hat{i}_1, \ \hat{i}_2, \ k) = \\
(0, 0, 0, 3), \\
(0, 0, 1, 4), \\
(0, 0, 2, 5), \\
(0, 1, 0, 7), \\
(0, 1, 1, 8), \\
(0, 1, 2, 9), \\
(0, 2, 0, 11), \\
(0, 2, 1, 12), \\
(0, 3, 0, 15), \\
(0, 3, 1, 16), \\
(0, 3, 2, 17), \\
(1, 0, 0, 22), \\
(1, 0, 1, 23), \\
(1, 3, 2, 36)
\]

That is, the highest-ordered index turns fastest. — end example ]

4 It is possible to have degenerate generalized slices in which an address is repeated.

5 [ Example: If the stride parameters in the previous example are changed to \{1, 1, 1\}, the first few elements of the resulting sequence of indices will be

\[
(0, 0, 0, 3), \\
(0, 0, 1, 4), \\
(0, 0, 2, 5), \\
(0, 1, 0, 4), \\
(0, 1, 1, 5), \\
(0, 1, 2, 6), \\
\ldots
\]

— end example ]

6 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 Numeric arrays

26.5.6.1 gslice constructors

```cpp
gslice();
gslice(size_t start, const valarray<size_t>& lengths, const valarray<size_t>& strides);
gslice(const gslice&);
```

The default constructor is equivalent to `gslice(0, valarray<size_t>(), valarray<size_t>())`. The constructor with arguments builds a `gslice` based on a specification of start, lengths, and strides, as explained in the previous section.

26.5.6.2 gslice access functions

```cpp
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

```cpp
namespace std {
    template <class T> class gslice_array {
        public:
            typedef T value_type;

            void operator= (const valarray<T>&) const;
            void operator*= (const valarray<T>&) const;
            void operator/= (const valarray<T>&) const;
            void operator%= (const valarray<T>&) const;
            void operator+= (const valarray<T>&) const;
            void operator-= (const valarray<T>&) const;
            void operator^= (const valarray<T>&) const;
            void operator&= (const valarray<T>&) const;
            void operator|= (const valarray<T>&) const;
            void operator<<=(const valarray<T>&) const;
            void operator>>=(const valarray<T>&) const;
            gslice_array(const gslice_array&);
            ~gslice_array();

            const gslice_array& operator=(const gslice_array&); const
            void operator=(const T&) const;

            gslice_array() = delete;  // as implied by declaring copy constructor above
        };
    }
}
```

This template is a helper template used by the slice subscript operator.
935  Numerics library  

26.5  Numeric arrays

```c
slice_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 assignment

```c
void operator=(const valarray<T>&) const;
const gslice_array& operator=(const gslice_array&) const;
```

These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which the `gslice_array` object refers.

### 26.5.7.2 gslice_array

```c
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.

### 26.5.7.3 gslice_array fill function

```c
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

```c
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 T&) const;
  }
}
```

Draft
void operator\%= (const valarray\textlangle T\rangle & const;
void operator\+= (const valarray\textlangle T\rangle & const;
void operator\-= (const valarray\textlangle T\rangle & const;
void operator\%= (const valarray\textlangle T\rangle & const;
void operator\%= (const valarray\textlangle T\rangle & const;
void operator\%= (const valarray\textlangle T\rangle & const;
void operator\%= (const valarray\textlangle T\rangle & const;
void operator\%= (const valarray\textlangle T\rangle & const;
void operator\%= (const valarray\textlangle T\rangle & const;

mask_array(const mask_array&);
\texttt{\neg}mask_array();
const mask_array& operator=(const mask_array&) \texttt{const};
void operator=(const T&) \texttt{const};
mask_array() = \texttt{delete}; \quad // as implied by declaring copy constructor above
\}

1 This template is a helper template used by the mask subscript operator:

\begin{verbatim}
mask_array\textlangle T\rangle \texttt{valarray\textlangle T\rangle::operator[]}(const valarray\textlangle bool\rangle&).
\end{verbatim}

2 It has reference semantics to a subset of an array specified by a boolean mask. Thus, the expression \texttt{a[mask] = b;} has the effect of assigning the elements of \texttt{b} to the masked elements in \texttt{a} (those for which the corresponding element in \texttt{mask} is true.)

26.5.8.1 mask_array assignment \[\text{mask.array.assign}\]

void operator=(const valarray\textlangle T\rangle& const;
const mask_array& operator=(const mask_array& \texttt{const};

1 These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the \texttt{valarray\textlangle T\rangle} object to which it refers.

26.5.8.2 mask_array computed assignment \[\text{mask.array.comp.assign}\]

void operator\*= (const valarray\textlangle T\rangle& const;
void operator/= (const valarray\textlangle T\rangle& const;
void operator\%= (const valarray\textlangle T\rangle& const;
void operator\%= (const valarray\textlangle T\rangle& const;
void operator\%= (const valarray\textlangle T\rangle& const;
void operator\%= (const valarray\textlangle T\rangle& const;
void operator\%= (const valarray\textlangle T\rangle& const;
void operator\%= (const valarray\textlangle T\rangle& const;

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the \texttt{valarray\textlangle T\rangle} object to which the mask object refers.

Draft
26.5.8.3  mask_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 `mask_array` object refers.

26.5.9  Class template indirect_array

```cpp
namespace std {
    template <class T> class indirect_array {
        public:
            typedef T value_type;
            void operator= (const valarray<T>&) const;
            void operator*= (const valarray<T>&) const;
            void operator/= (const valarray<T>&) const;
            void operator%= (const valarray<T>&) const;
            void operator+= (const valarray<T>&) const;
            void operator-= (const valarray<T>&) const;
            void operator^= (const valarray<T>&) const;
            void operator&= (const valarray<T>&) const;
            void operator|= (const valarray<T>&) const;
            void operator<<=(const valarray<T>&) const;
            void operator>>=(const valarray<T>&) const;
            indirect_array(const indirect_array&);
            const indirect_array& operator=(const indirect_array&) const;
            void operator=(const T&) const;
            indirect_array() = delete;  // as implied by declaring copy constructor above
    };
}
```

This template is a helper template used by the indirect subscript operator

```cpp
indirect_array<T> valarray<T>::operator[](const valarray<
```

2 It has reference semantics to a subset of an array specified by an `indirect_array`. Thus the expression `a[indirect] = b;` has the effect of assigning the elements of `b` to the elements in `a` whose indices appear in `indirect`.

26.5.9.1  indirect_array assignment

```cpp
void operator=(const valarray<T>&) const;
const indirect_array& operator=(const indirect_array&) const;
```
These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which it refers.

If the `indirect_array` specifies an element in the `valarray<T>` object to which it refers more than once, the behavior is undefined.

```cpp
int addr[] = {2, 3, 1, 4, 4};
valarray<size_t> indirect(addr, 5);
valarray<double> a(0., 10), b(1., 5);
a[indirect] = b;
```

results in undefined behavior since element 4 is specified twice in the indirection. —end example

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.

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.

**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,
```
template <class InputIterator1, class InputIterator2, class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, T init, BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);

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) T accumulate(InputIterator first, InputIterator last, T init);
(template) 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
26.6 Generalized numeric operations

order.\textsuperscript{285)

\textit{Requires:} T shall meet the requirements of CopyConstructible (34) and Assignable (23.1) types. In the range [first,last], binary_op shall neither modify elements nor invalidate iterators or subranges.\textsuperscript{286}

26.6.2 Inner product \textit{[inner.product]}

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

\textit{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 [first, last) and iterator i2 in the range [first2, first2 + (last - first)) in order.

\textit{Requires:} T shall meet the requirements of CopyConstructible (34) and Assignable (23.1) types. In the ranges [first, last] and [first2, first2 + (last - first)] binary_op1 and binary_op2 shall neither modify elements nor invalidate iterators or subranges.\textsuperscript{287}

26.6.3 Partial sum \textit{[partial.sum]}

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

\textit{Effects:} Assigns to every element referred to by iterator i in the range [result, result + (last - first)) a value correspondingly equal to

\begin{verbatim}
((...(first + *(first + 1)) + ...) + *(first + (i - result)))
\end{verbatim}

or

\textsuperscript{285) accumulate} is similar to the APL reduction operator and Common Lisp reduce function, but it avoids the difficulty of defining the result of reduction on an empty sequence by always requiring an initial value.

\textsuperscript{286) The use of fully closed ranges is intentional

\textsuperscript{287) The use of fully closed ranges is intentional

Draft
binary_op(binary_op(...,
    binary_op(*first, *(first + 1)),...), *(first + (i - result)))

2 Returns: result + (last - first).
3 Complexity: Exactly (last - first) - 1 applications of binary_op.
4 Requires: In the ranges [first,last] and [result, result + (last - first)] binary_op shall neither modify elements nor invalidate iterators or subranges. 288)
5 Remarks: result may be equal to first.

26.6.4 Adjacent difference [adjacent.difference]

template <class InputIterator, class OutputIterator>
OutputIterator adjacent_difference(
    InputIterator first, InputIterator last,
    OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator adjacent_difference(
    InputIterator first, InputIterator last,
    OutputIterator result,
    BinaryOperation binary_op);

1 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
    binary_op(*(first + (i - result)), *(first + (i - result) - 1)).

result gets the value of *first.
2 Requires: In the ranges [first,last] and [result, result + (last - first)], binary_op shall neither modify elements nor invalidate iterators or subranges. 289)
3 Remarks: result may be equal to first.
4 Returns: result + (last - first).
5 Complexity: Exactly (last - first) - 1 applications of binary_op.

26.7 C Library [c.math]

1 The header <ctgmath> simply includes the headers <ccomplex> and <cmath>.
2 [ Note: The overloads provided in C99 by magic macros are already provided in <ccomplex> and <cmath> by "sufficient" additional overloads. — end note ]

288) The use of fully closed ranges is intentional.
289) The use of fully closed ranges is intentional.
3 The header `<tgmath.h>` effectively includes the headers `<complex.h>` and `<math.h>`.

4 Tables 111 and 112 describe headers `<cmath>` and `<cstdlib>`, respectively.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td></td>
</tr>
<tr>
<td>FP_FAST_FMA</td>
<td>FP_ILOGBNAN</td>
</tr>
<tr>
<td>FP_FAST_FMAF</td>
<td>FP_INFINITE</td>
</tr>
<tr>
<td>FP_FAST_FMAL</td>
<td>FP_NAN</td>
</tr>
<tr>
<td>FP_ILOGBO</td>
<td>FP_NORMAL</td>
</tr>
<tr>
<td>Types:</td>
<td>double_t float_t</td>
</tr>
<tr>
<td>Functions:</td>
<td></td>
</tr>
<tr>
<td>abs</td>
<td>cosh</td>
</tr>
<tr>
<td>acos</td>
<td>erf</td>
</tr>
<tr>
<td>acosh</td>
<td>erfc</td>
</tr>
<tr>
<td>asin</td>
<td>exp</td>
</tr>
<tr>
<td>asinh</td>
<td>exp2</td>
</tr>
<tr>
<td>atan</td>
<td>expm1</td>
</tr>
<tr>
<td>atanh</td>
<td>fahs</td>
</tr>
<tr>
<td>atan2</td>
<td>fahs</td>
</tr>
<tr>
<td>cbrt</td>
<td>floor</td>
</tr>
<tr>
<td>ceil</td>
<td>fma</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>isgreaterequal</td>
</tr>
<tr>
<td>isfinite</td>
<td>isinf</td>
</tr>
<tr>
<td>isgreater</td>
<td>islessequal</td>
</tr>
<tr>
<td>islesssuper</td>
<td>islessgreater</td>
</tr>
</tbody>
</table>

5 The contents of these headers are the same as the Standard C library headers `<math.h>` and `<stdlib.h>` respectively, with the following changes:

6 The `rand` function has the semantics specified in the C standard, except that the implementation may specify that particular library functions may call `rand`.

7 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.

8 The added signatures are:

```c
long abs(long);              // labs()
long long abs(long long);    // llabs()
ldiv_t div(long, long);      // ldiv()
ldiv_t div(long long, long long); // lldiv()
```

290) All macros except `HUGE_VAL`, both types, many functions, and all templates added by TR1.

Draft
Table 112: 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>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>

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.

The added signatures are:

```c
float abs(float);
float acos(float);
float acosh(float);
float asin(float);
float asinh(float);
float atan(float);
float atan2(float, float);
float atanh(float);
float cbrt(float);
float ceil(float);
float copysign(float, float);
float cos(float);
float cosh(float);
float erf(float);
float erfc(float);
float exp(float);
float exp2(float);
float expm1(float);
float fabs(float);
float fdim(float, float);
float floor(float);
float fma(float, float, float);
float fmax(float, float);
float fmin(float, float);
float fmod(float, float);
float frexp(float, int*);
float hypot(float, float);
int ilogb(float);
float ldexp(float, int);
float lgamma(float);
long long llrint(float);
long long llround(float);
float log(float);
```
float log10(float);
float log1p(float);
float log2(float);
float logb(float);
long lrint(float);
long lround(float);
float modf(float, float*);
float nearbyint(float);
float nextafter(float, float);
float nexttoward(float, long double);
float pow(float, float);
float 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);
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:
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 isless(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);
}

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

This clause describes components that C++ programs may use to perform input/output operations.

The following subclauses describe requirements for stream parameters, and components for forward declarations of iostreams, predefined iostreams objects, base iostreams classes, stream buffering, stream formatting and manipulators, string streams, and file streams, as summarized in Table 113.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.1 Requirements</td>
<td></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></td>
<td>&lt;ostream&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;iomanip&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>
<tr>
<td></td>
<td>&lt;cstdio&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cinttypes&gt;</td>
</tr>
</tbody>
</table>

27.1 Iostreams requirements

27.1.1 Imbue Limitations

No function described in clause 27 except for ios_base::imbue and basic_filebuf::pubimbue causes any instance of basic_ios::imbue or basic_streambuf::imbue to be called. If any user function called from a function declared in clause 27 or as an overriding virtual function of any class declared in clause 27 calls imbue, the behavior is undefined.

27.1.2 Positioning Type Limitations

The classes of clause 27 with template arguments charT and traits behave as described if traits::pos_type and traits::off_type are streampos and streamoff respectively. Except as noted explicitly below, their behavior
when traits::pos_type and traits::off_type are other types is implementation-defined.

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;
    template <class charT, class traits = char_traits<charT> >
        class basic_ofstream;
    template <class charT, class traits = char_traits<charT> >
        class basic_fstream;

    template <class charT, class traits = char_traits<charT> >
        class istreambuf_iterator;
    template <class charT, class traits = char_traits<charT> >
        class ostreambuf_iterator;
}
typedef basic_ios<char> ios;
typedef basic_ios<wchar_t> wios;

typedef basic_streambuf<char> streambuf;
typedef basic_istream<char> istream;
typedef basic_ostream<char> ostream;
typedef basic_iostream<char> iostream;

typedef basic_stringbuf<char> stringbuf;
typedef basic_istringstream<char> istringstream;
typedef basic_ostringstream<char> ostringstream;
typedef basic_stringstream<char> stringstream;

typedef basic_filebuf<char> filebuf;
typedef basic_ifstream<char> ifstream;

typedef basic_streambuf<wchar_t> wstreambuf;
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_istringstream<wchar_t> wistringstream;
typedef basic_ostringstream<wchar_t> wostringstream;
typedef basic_stringstream<wchar_t> wstringstream;

typedef basic_filebuf<wchar_t> wfilebuf;
typedef basic_ifstream<wchar_t> wifstream;

template <class state> class fpos;
typedef fpos<char_traits<char>::state_type> streampos;
typedef fpos<char_traits<wchar_t>::state_type> wstreampos;

1  Default template arguments are described as appearing both in `<iosfwd>` and in the synopsis of other headers but it is well-formed to include both `<iosfwd>` and one or more of the other headers.\(^{291}\)

2  \([\textit{Note:}]\) The class template specialization `basic_ios<charT,traits>` serves as a virtual base class for the class templates `basic_istream`, `basic_ostream`, and class templates derived from them. `basic_iostream` is a class template derived from both `basic_istream<charT,traits>` and `basic_ostream<charT,traits>`.

3  The class template specialization `basic_streambuf<charT,traits>` serves as a base class for template classes `basic_stringbuf` and `basic_filebuf`.

\(^{291}\) 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.
4 The class template specialization basic_istream<charT,traits> serves as a base class for template classes basic_iostreamstream and basic_ifstream.

5 The class template specialization basic_ostream<charT,traits> serves as a base class for template classes basic_ostringstream and basic_ofstream.

6 The class template specialization basic_iostream<charT,traits> serves as a base class for template classes basic_stringstream and basic_fstream.

7 Other typedefs define instances of class templates specialized for char or wchar_t types.

8 Specializations of the class template fpos are used for specifying file position information.

9 The types streampos and wstreampos are used for positioning streams specialized on char and wchar_t respectively.

10 This synopsis suggests a circularity between streampos and char_traits<char>. An implementation can avoid this circularity by substituting equivalent types. One way to do this might be

```c
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

```c
#include <ios>
#include <streambuf>
#include <istream>
#include <ostream>

namespace std {
  extern istream cin;
  extern ostream cout;
  extern ostream cerr;
  extern ostream clog;

  extern wistream wcin;
  extern wostream wcout;
  extern wostream wcerr;
  extern wostream wclog;
}
```

1 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.

Draft
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. The objects are not destroyed during program 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

**istream cin**

The object `cin` controls input from a stream buffer associated with the object `stdin`, declared in `<cstdio>`.

**ostream cout**

The object `cout` controls output to a stream buffer associated with the object `stdout`, declared in `<cstdio>`.

**ostream cerr**

The object `cerr` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>`.

### 27.3.2 Wide stream objects

**wistream wcin**

The object `wcin` controls input from a stream buffer associated with the object `stdin`, declared in `<cstdio>`.

**wostream wcout**

The object `wcout` controls output to a stream buffer associated with the object `stdout`, declared in `<cstdio>`.
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);

    ios_base& showpos (ios_base& str);
    ios_base& noshowpos (ios_base& str);

    ios_base& skipws (ios_base& str);
    ios_base& noskipws (ios_base& str);

    ios_base& uppercase (ios_base& str);
    ios_base& nouppercase (ios_base& str);

    ios_base& unitbuf (ios_base& str);
    ios_base& nounitbuf (ios_base& str);

    // 27.4.5.2 adjustfield:

    Draft
ios_base& internal (ios_base& str);
ios_base& left (ios_base& str);
ios_base& right (ios_base& str);

// 27.4.5.3 basefield:
ios_base& dec (ios_base& str);
ios_base& hex (ios_base& str);
ios_base& oct (ios_base& str);

// 27.4.5.4 floatfield:
ios_base& fixed (ios_base& str);
ios_base& scientific (ios_base& str);
ios_base& hexfloat (ios_base& str);
ios_base& defaultfloat(ios_base& str);
}

27.4.1 Types [stream.types]

typedef OFF_T streamoff;

1 The type streamoff is an implementation-defined type that satisfies the requirements of 27.4.3.2.

typedef SZ_T streamsize;

2 The type streamsize is a synonym for one of the signed basic integral types. It is used to represent the number of characters transferred in an I/O operation, or the size of I/O buffers.284)

27.4.2 Class ios_base [ios.base]

namespace std {
    class ios_base {
        public:
            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;

284) streamsize is used in most places where ISO C would use size_t. Most of the uses of streamsize could use size_t, except for the strstreambuf constructors, which require negative values. It should probably be the signed type corresponding to size_t (which is what Posix.2 calls ssize_t).

Draft
27.4 Iostreams base classes

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);

ios_base(const ios_base&) = delete;
ios_base& operator=(const ios_base&) = delete;

static bool sync_with_stdio(bool sync = true);

protected:
  ios_base();

private:
  // static int index;
  // long* iarray;
  // void** parray;
};

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.

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 Iostreams base classes

27.4.2.1 Types

27.4.2.1.1 Class ios_base::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

typedef T1 fmtflags;

The type fmtflags is a bitmask type (17.3.2.1.2). Setting its elements has the effects indicated in Table 114.

Type fmtflags also defines the constants indicated in Table 115.

27.4.2.1.3 Type ios_base::iostate

typedef T2 iostate;

The type iostate is a bitmask type (17.3.2.1.2) that contains the elements indicated in Table 116.

Type iostate also defines the constant:
— goodbit, the value zero.
Table 114: 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</td>
</tr>
<tr>
<td></td>
<td>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>skipws</td>
<td>generates a + sign in non-negative generated numeric output</td>
</tr>
<tr>
<td>unitbuf</td>
<td>flushes output after each output operation</td>
</tr>
<tr>
<td>uppercase</td>
<td>replaces certain lowercase letters with their uppercase equivalents in generated output</td>
</tr>
</tbody>
</table>

Table 115: fmtflags constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Allowable values</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjustfield</td>
<td>left</td>
</tr>
<tr>
<td>basefield</td>
<td>dec</td>
</tr>
<tr>
<td>floatfield</td>
<td>scientific</td>
</tr>
</tbody>
</table>

Table 116: 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>
### 27.4 Iostreams base classes

#### 27.4.2.1.4 Type `ios_base::openmode` \[ios::openmode\]

typedef T3 openmode;

The type `openmode` is a bitmask type (17.3.2.1.2). It contains the elements indicated in Table 117.

<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` \[ios::seekdir\]

typedef T4 seekdir;

The type `seekdir` is an enumerated type (17.3.2.1.1) that contains the elements indicated in Table 118.

<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>

#### 27.4.2.1.6 Class `ios_base::Init` \[ios::Init\]

namespace std {
    class ios_base::Init {
    public:
        Init();
        ~Init();
    private:
        // static int init_cnt;
        exposition only
    };
}

The class `Init` describes an object whose construction ensures the construction of the eight objects declared in `<iostream>` (27.3) that associate file stream buffers with the standard C streams provided for by the functions declared in `<cstdio>` (27.8.2).

For the sake of exposition, the maintained data is presented here as:
— static int init_cnt, counts the number of constructor and destructor calls for class Init, initialized to zero.

Init();

Effects: Constructs an object of class Init. If init_cnt is zero, the function stores the value one in init_cnt, then constructs and initializes the objects cin, cout, cerr, clog (27.3.1), wcin, wcout, wcerr, and wclog (27.3.2). In any case, the function then adds one to the value stored in init_cnt.

~Init();

Effects: Destroys an object of class Init. The function subtracts one from the value stored in init_cnt and, if the resulting stored value is one, calls cout.flush(), cerr.flush(), clog.flush(), wcout.flush(), wcerr.flush(), wclog.flush().

27.4.2.2 ios_base state functions

fmtflags flags() const;

Returns: The format control information for both input and output.

fmtflags flags(fmtflags fmtfl);

Postcondition: fmtfl == flags().

Returns: The previous value of flags().

fmtflags setf(fmtflags fmtfl);

Effects: Sets fmtfl in flags().

Returns: The previous value of flags().

fmtflags setf(fmtflags fmtfl, fmtflags mask);

Effects: Clears mask in flags(), sets fmtfl & mask in flags().

Returns: The previous value of flags().

void unsetf(fmtflags mask);

Effects: Clears mask in flags().

streamsize precision() const;

Returns: The precision to generate on certain output conversions.

streamsize precision(streamsize prec);

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 [ios.baselocales]

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 [ios.members.static]

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

Draft
ungetc(c, f);

is the same as the effect of
   str.rdbuf()->sputbackc(c);

for any sequence of characters.295)

### 27.4.2.5 ios_base storage functions

```c
static int xalloc();

Returns: index ++.
```

```c
long& iword(int idx);

Effects: If iarray is a null pointer, allocates an array of long of unspecified size and stores a pointer to its first element in iarray. The function then extends the array pointed at by iarray as necessary to include the element iarray[idx]. Each newly allocated element of the array is initialized to zero. The reference returned is invalid after any other operations on the object.296) 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 fails297) and *this is a base subobject of a basic_ios<> object or subobject, the effect is equivalent to calling basic_ios<>::setstate(badbit) on the derived object (which may throw failure).

Returns: On success iarray[idx]. On failure, a valid long& initialized to 0.
```

```c
void* & pword(int idx);

Effects: If parray is a null pointer, allocates an array of pointers to void of unspecified size and stores a pointer to its first element in parray. The function then extends the array pointed at by parray as necessary to include the element parray[idx]. Each newly allocated element of the array is initialized to a null pointer. The reference returned is invalid after any other operations on the object. However, the value of the storage referred to is retained, so that until the next call to copyfmt, calling pword with the same index yields another reference to the same value. If the function fails298) and *this is a base subobject of a basic_ios<> object or subobject, the effect is equivalent to calling basic_ios<>::setstate(badbit) on the derived object (which may throw failure).

Returns: On success parray[idx]. On failure a valid void* & initialized to 0.

Remarks: After a subsequent call to pword(int) for the same object, the earlier return value may no longer be valid.
```

### 27.4.2.6 ios_base callbacks

295) 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.

296) An implementation is free to implement both the integer array pointed at by iarray and the pointer array pointed at by parray as sparse data structures, possibly with a one-element cache for each.

297) for example, because it cannot allocate space.

298) for example, because it cannot allocate space.
void register_callback(event_callback fn, int index);

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.

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 [ios.base.cons]

ios_base();

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.

~ios_base()

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 [fpos]

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 [fpos.members]

void state(stateT s);

Effects: Assign s to st.

stateT state() const;

Returns: Current value of st.
27.4.3.2 fpos requirements

Operations specified in Table 119 are permitted. In that table,

— \( P \) refers to an instance of fpos,
— \( p \) and \( q \) refer to values of type \( P \),
— \( O \) refers to type streamoff,
— \( o \) refers to a value of type streamoff,
— \( sz \) refers to a value of type streamsize and
— \( i \) refers to a value of type int.

Table 119: Position type requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>operational semantics</th>
<th>assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P(i) )</td>
<td></td>
<td>( p == P(i) )</td>
<td>note: a destructor is assumed.</td>
</tr>
<tr>
<td>0(p)</td>
<td>OFF_T</td>
<td>converts to offset</td>
<td>( P(O(p)) == p )</td>
</tr>
<tr>
<td>p == q</td>
<td>convertible to bool</td>
<td>( !(p == q) )</td>
<td></td>
</tr>
<tr>
<td>p += q</td>
<td>fpos</td>
<td>+ offset</td>
<td>( q - o == p )</td>
</tr>
<tr>
<td>p -= q</td>
<td>fpos</td>
<td>- offset</td>
<td>( q + o == p )</td>
</tr>
<tr>
<td>o = p - q</td>
<td>OFF_T</td>
<td>distance</td>
<td>( q + o == p )</td>
</tr>
<tr>
<td>streamsize(o)</td>
<td>streamsize</td>
<td>converts</td>
<td>( \text{streamsize}(O(sz)) == sz )</td>
</tr>
<tr>
<td>0(sz)</td>
<td>OFF_T</td>
<td>converts</td>
<td>( \text{streamsize}(O(sz)) == sz )</td>
</tr>
</tbody>
</table>

[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. — end note]

Stream operations that return a value of type traits::pos_type return \( P(O(-1)) \) as an invalid value to signal an error. If this value is used as an argument to any istream, ostream, or streambuf member that accepts a value of type traits::pos_type then the behavior of that function is undefined.

27.4.4 Class template basic_ios

```cpp
namespace std {
    template <class charT, class traits = char_traits<charT> >
    
    Draft
```
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 dfault) const;
   char_type widen(char c) const;

   basic_ios(const basic_ios& ) = delete;
   basic_ios& operator=(const basic_ios&) = delete;

   protected:
   basic_ios();
   void init(basic_streambuf<charT,traits>* sb);

   Draft
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

explicit basic_ios(basic_streambuf<charT, traits>* sb);

Effects: Constructs an object of class basic_ios, assigning initial values to its member objects by calling init(sb).

basic_ios();

Effects: Constructs an object of class basic_ios (27.4.2.7) 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();

Remarks: The destructor does not destroy rdbuf().

void init(basic_streambuf<charT, traits>* sb);

Postconditions: The postconditions of this function are indicated in Table 120.

<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>rdbufstate()</td>
<td>goodbit if sb is not a null pointer, otherwise badbit.</td>
</tr>
<tr>
<td>exceptions()</td>
<td>goodbit</td>
</tr>
<tr>
<td>flags()</td>
<td>skipws</td>
</tr>
<tr>
<td>width()</td>
<td>0</td>
</tr>
<tr>
<td>precision()</td>
<td>6</td>
</tr>
<tr>
<td>fill()</td>
<td>widen(‘ ’);</td>
</tr>
<tr>
<td>getloc()</td>
<td>a copy of the value returned by locale()</td>
</tr>
<tr>
<td>iarray</td>
<td>a null pointer</td>
</tr>
<tr>
<td>parray</td>
<td>a null pointer</td>
</tr>
</tbody>
</table>

27.4.4.2 Member functions

basic_ostream<charT, traits>* tie() const;

Returns: An output sequence that is tied to (synchronized with) the sequence controlled by the stream buffer.
basic_ostream<
character_type,
facet_traits>
* tie(basic_ostream<
character_type,
facet_traits>* tiestr);

Postcondition: tiestr == tie().

Returns: The previous value of tie().

basic_streambuf<
character_type,
facet_traits>* rdbuf() const;

Returns: A pointer to the streambuf associated with the stream.

basic_streambuf<
character_type,
facet_traits>* rdbuf(basic_streambuf<
character_type,
facet_traits>* sb);

Postcondition: sb == rdbuf().

Effects: Calls clear().

Returns: The previous value of rdbuf().

locale imbue(const locale& loc);

Effects: Calls ios_base::imbue(loc) (27.4.2.3) and if rdbuf() != 0 then rdbuf() -> pubimbue(loc) (27.5.2.2.1).

Returns: The prior value of ios_base::imbue().

char narrow(character_type c, character dfault) const;

Returns: use_facet<
cctype<character_type>>(getloc()).narrow(c, dfault)

char_type widen(character c) const;

Returns: use_facet<
cctype<character_type>>(getloc()).widen(c)

char_type fill() const;

Returns: The character used to pad (fill) an output conversion to the specified field width.

char_type fill(char_type fillch);

Postcondition: traits::eq(fillch, fill())

Returns: The previous value of fill().

basic_ios& copyfmt(const basic_ios& rhs);

Effects: If (this == &rhs) does nothing. Otherwise assigns to the member objects of *this the corresponding member objects of rhs, except that:

— rdstate() and rdbuf() are left unchanged;

— 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.299)

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.

299) This suggests an infinite amount of copying, but the implementation can keep track of the maximum element of the arrays that is non-zero.
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;

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.

```cpp
void setstate(iostate state);
```

Effects: Calls clear(rdstate() | state) (which may throw basic_ios::failure (27.4.2.1.1)).

```cpp
bool good() const;
```

Returns: rdstate() == 0

```cpp
bool eof() const;
```

Returns: true if eofbit is set in rdstate().

```cpp
bool fail() const;
```

Returns: true if failbit or badbit is set in rdstate().\(^{300}\)

```cpp
bool bad() const;
```

Returns: true if badbit is set in rdstate().

```cpp
iostate exceptions() const;
```

Returns: A mask that determines what elements set in rdstate() cause exceptions to be thrown.

```cpp
void exceptions(iostate except);
```

Postcondition: except == exceptions().

```cpp
Effects: Calls clear(rdstate()).
```

### 27.4.5 ios_base manipulators

#### 27.4.5.1 fmtflags manipulators

```cpp
ios_base& boolalpha(ios_base& str);
```

Effects: Calls str.setf(ios_base::boolalpha).

```cpp
Returns: str.
```

```cpp
ios_base& noboolalpha(ios_base& str);
```

Effects: Calls str.unsetf(ios_base::boolalpha).

```cpp
Returns: str.
```

```cpp
ios_base& showbase(ios_base& str);
```

\(^{300}\) Checking badbit also for fail() is historical practice.
Effects: Calls `str.setf(ios_base::showbase)`.
Returns: `str`.

`ios_base& noshowbase(ios_base& str);`
Effects: Calls `str.unsetf(ios_base::showbase)`.
Returns: `str`.

`ios_base& showpoint(ios_base& str);`
Effects: Calls `str.setf(ios_base::showpoint)`.
Returns: `str`.

`ios_base& noshowpoint(ios_base& str);`
Effects: Calls `str.unsetf(ios_base::showpoint)`.
Returns: `str`.

`ios_base& showpos(ios_base& str);`
Effects: Calls `str.setf(ios_base::showpos)`.
Returns: `str`.

`ios_base& noshowpos(ios_base& str);`
Effects: Calls `str.unsetf(ios_base::showpos)`.
Returns: `str`.

`ios_base& skipws(ios_base& str);`
Effects: Calls `str.setf(ios_base::skipws)`.
Returns: `str`.

`ios_base& noskipws(ios_base& str);`
Effects: Calls `str.unsetf(ios_base::skipws)`.
Returns: `str`.

`ios_base& uppercase(ios_base& str);`
Effects: Calls `str.setf(ios_base::uppercase)`.
Returns: `str`.

`ios_base& nouppercase(ios_base& str);`
Effects: Calls `str.unsetf(ios_base::uppercase)`.
Returns: `str`.
27.4 Iostreams base classes

ios_base& unitbuf(ios_base& str);

Effects: Calls str.setf(ios_base::unitbuf).
Returns: str.

ios_base& nunitbuf(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\(^{301}\).

ios_base& hex(ios_base& str);

Effects: Calls str.setf(ios_base::hex, ios_base::basefield).
Returns: str.

ios_base& oct(ios_base& str);

Effects: Calls str.setf(ios_base::oct, ios_base::basefield).
Returns: str.

\(^{301}\) 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.

Draft
27.4.5.4 floatfield manipulators

ios_base& fixed(ios_base& str);

   Effects: Calls str.setf(ios_base::fixed, ios_base::floatfield).

   Returns: str.

ios_base& scientific(ios_base& str);

   Effects: Calls str.setf(ios_base::scientific, ios_base::floatfield).

   Returns: str.

ios_base& hexfloat(ios_base& str);

   Effects: Calls str.setf(ios_base::fixed | ios_base::scientific, ios_base::floatfield).

   Returns: str.

   [ Note: The more obvious use of ios_base::hex to specify hexadecimal floating-point format would change the meaning of existing well defined programs. C++2003 gives no meaning to the combination of fixed and scientific.
   — end note ]

ios_base& defaultfloat(ios_base& str);

   Effects: Calls str.unsetf(ios_base::floatfield).

   Returns: str.

27.5 Stream buffers

**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

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.

Draft
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 xbeg here);
- the next pointer, or next element address that is a current candidate for reading or writing (called xnext here);
- the end pointer, or first element address beyond the end of the array (called xend here).

The following semantic constraints shall always apply for any set of three pointers for a sequence, using the pointer names given immediately above:

- If xnext is not a null pointer, then xbeg and xend shall also be non-null pointers into the same charT array, as described above; otherwise, xbeg and xend shall also be null.
- If xnext is not a null pointer and xnext < xend for an output sequence, then a write position is available. In this case, *xnext shall be assignable as the next element to write (to put, or to store a character value, into the sequence).
- If xnext is not a null pointer and xbeg < xnext for an input sequence, then a putback position is available. In this case, xnext[-1] shall have a defined value and is the next (preceding) element to store a character that is put back into the input sequence.
- If xnext is not a null pointer and xnext < xend for an input sequence, then a read position is available. In this case, *xnext shall have a defined value and is the next element to read (to get, or to obtain a character value, from the sequence).

### 27.5.2 Class template basic_streambuf<charchT,traits> [streambuf]

```cpp
namespace std {
    template <class charchT, class traits = char_traits<charchT> >
    class basic_streambuf {
    public:

        // types:
        typedef charchT char_type;
        typedef typename traits::int_type int_type;
        typedef typename traits::pos_type pos_type;
        typedef typename traits::off_type off_type;
        typedef traits traits_type;

        virtual ~basic_streambuf();

        // 27.5.2.2) locales:
        locale pubimbue(const locale& loc);
    }
}
```

Draft
locale getloc() const;

// 27.5.2.2 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* gbeg, char_type* gnext, char_type* gend);

// 27.5.2.3.3 Put area:
char_type* pbase() const;
char_type* pptr() const;
char_type* epptr() const;
void pbump(int n);
void setp(char_type* pbeg, char_type* pend);

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]
975 Input/output library

27.5 Stream buffers

2 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

27.5.2.2.1 Locales

locale pubimbue(const locale& loc);

Postcondition: loc == getloc().

Effects: Calls imbue(loc).

Returns: Previous value of getloc().

locale getloc() const;

Returns: If pubimbue() has ever been called, then the last value of loc supplied, otherwise the current global locale, locale(), in effect at the time of construction. If called after pubimbue() has been called but before pubimbue has returned (i.e. from within the call of imbue()) then it returns the previous value.

27.5.2.2.2 Buffer management and positioning

basic_streambuf<char_type,traits>* pubsetbuf(char_type* s, streamsize n);

Returns: setbuf(s, n).

pos_type pubseekoff(off_type off, ios_base::seekdir way,
   ios_base::openmode which = ios_base::in | ios_base::out);

Draft
27.5 Stream buffers

Returns: seekoff(off, way, which).

pos_type pubseekpos(pos_type sp,  
  ios_base::openmode which = ios_base::in | ios_base::out);

Returns: seekpos(sp, which).

intpubsync();

Returns: sync().

27.5.2.2.3 Get area [streambuf.pub.get]

streamsize in_avail();

Returns: If a read position is available, returns egptr() - gptr(). Otherwise returns showmanyc() (27.5.2.4.3).

int_typesnc();

Effects: Calls sbumpc().

Returns: if that function returns traits::eof(), returns traits::eof(). Otherwise, returns sgetc().

int_typesbumpc();

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_typesungetc();

Returns: If the input sequence read position is not available, returns pbackfail(). Otherwise, returns traits::to_int_type(*gptr()).

streamsize sgetn(char_type* s, streamsize n);

Returns: xgetn(s, n).

27.5.2.2.4 Putback [streambuf.pub.pback]

int_typesputbackc(char_type c);

Returns: If the input sequence putback position is not available, or if traits::eq(c, gptr()[-1]) is false, returns pbackfail(traits::to_int_type(c)). Otherwise, decrements the next pointer for the input sequence and returns traits::to_int_type(*gptr()).

int_typesungetc();

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

```cpp
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)`.

```cpp
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

```cpp
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()`
- `epptr() == rhs.epptr()`
- `getloc() == rhs.getloc()`

Returns: `*this`.

```cpp
void swap(basic_streambuf&& rhs);
```

Effects: Swaps the data members of `rhs` and `*this`.

### 27.5.2.3.2 Get area access

```cpp
char_type* eback() const;
```

Returns: The beginning pointer for the input sequence.

```cpp
char_type* gptr() const;
```

Returns: The next pointer for the input sequence.

```cpp
char_type* egptr() const;
```

Returns: The next pointer for the input sequence.
27.5 Stream buffers

Returns: The end pointer for the input sequence.

```cpp
void gbump(int n);
```

Effects: Adds \( n \) to the next pointer for the input sequence.

```cpp
void setg(char_type* gbeg, char_type* gnext, char_type* gend);
```

Postconditions: \( gbeg == \text{eback}() \), \( gnext == \text{gptr}() \), and \( gend == \text{egptr}() \).

27.5.2.3 Put area access

```cpp
char_type* pbase() const;
```

Returns: The beginning pointer for the output sequence.

```cpp
char_type* pptr() const;
```

Returns: The next pointer for the output sequence.

```cpp
char_type* epptr() const;
```

Returns: The end pointer for the output sequence.

```cpp
void pbump(int n);
```

Effects: Adds \( n \) to the next pointer for the output sequence.

```cpp
void setp(char_type* pbeg, char_type* pend);
```

Postconditions: \( pbeg == \text{pbase}() \), \( pbeg == \text{pptr}() \), and \( pend == \text{epptr}() \).

27.5.2.4 basic_streambuf virtual functions

27.5.2.4.1 Locales

```cpp
void imbue(const locale&);
```

Effects: Change any translations based on locale.

Remarks: Allows the derived class to be informed of changes in locale at the time they occur. Between invocations of this function a class derived from streambuf can safely cache results of calls to locale functions and to members of facets so obtained.

Default behavior: Does nothing.

27.5.2.4.2 Buffer management and positioning

```cpp
basic_streambuf* setbuf(char_type* s, streamsize n);
```
Effects: Influences stream buffering in a way that is defined separately for each class derived from basic_streambuf in this clause (27.7.1.4, 27.8.1.5).

Default behavior: Does nothing. Returns this.

```cpp
pos_type seekoff(off_type off, ios_base::seekdir way,
    ios_base::openmode which
    = ios_base::in | ios_base::out);
```

Effects: Alters the stream positions within one or more of the controlled sequences in a way that is defined separately for each class derived from basic_streambuf in this clause (27.7.1.4, 27.8.1.5).

Default behavior: Returns pos_type(off_type(-1)).

```cpp
pos_type seekpos(pos_type sp,
    ios_base::openmode which
    = ios_base::in | ios_base::out);
```

Effects: Alters the stream positions within one or more of the controlled sequences in a way that is defined separately for each class derived from basic_streambuf in this clause (27.7.1, 27.8.1.1).

Default behavior: Returns pos_type(off_type(-1)).

```cpp
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 [streambuf.virt.get]

```cpp
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().

```cpp
streamsize xsgetn(char_type* s, streamsize n);
```

Effects: Assigns up to n characters to successive elements of the array whose first element is designated by s. The characters assigned are read from the input sequence as if by repeated calls to sbumpc(). Assigning stops when either n characters have been assigned or a call to sbumpc() would return traits::eof().

---

303) The morphemes of showmanyc are “es-how-many-see”, not “show-manic”.
304) underflow or uflow might fail by throwing an exception prematurely. The intention is not only that the calls will not return eof() but that they will return “immediately.”
Returns: The number of characters assigned.

Remarks: Uses traits::eof().

int_type underflow();

Remarks: The public members of basic_streambuf call this virtual function only if gptr() is null or gptr() >= egptr()

Returns: traits::to_int_type(c), where c is the first character of the pending sequence, without moving the input sequence position past it. If the pending sequence is null then the function returns traits::eof() to indicate failure.

The pending sequence of characters is defined as the concatenation of:

a) If gptr() is non-NULL, then the egptr() - gptr() characters starting at gptr(), otherwise the empty sequence.

b) Some sequence (possibly empty) of characters read from the input sequence.

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 eback() is null then empty,

b) Otherwise the gptr() - eback() characters beginning at eback().

Effects: The function sets up the gptr() and egptr() satisfying one of:

a) If the pending sequence is non-empty, egptr() is non-null and egptr() - gptr() characters starting at gptr() are the characters in the pending sequence

b) If the pending sequence is empty, either gptr() is null or gptr() and egptr() are set to the same non-NULL pointer.

If eback() and gptr() are non-null then the function is not constrained as to their contents, but the “usual backup condition” is that either:

a) If the backup sequence contains at least gptr() - eback() characters, then the gptr() - eback() characters starting at eback() agree with the last gptr() - eback() characters of the backup sequence.

b) Or the n characters starting at gptr() - n agree with the backup sequence (where n is the length of the backup sequence)

Default behavior: Returns traits::eof().

int_type uflow();

---

305) Classes derived from basic_streambuf can provide more efficient ways to implement xsgetn() and xsputn() by overriding these definitions from the base class.
27.5.2.4.4 Putback

```c
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

```c
streamsize xputn(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.

```c
int_type overflow(int_type c = traits::eof());
```

Effects: Consumes some initial subsequence of the characters of the pending sequence. The pending sequence is defined as the concatenation of

a) if pbase() is NULL then the empty sequence otherwise, pptr() - pbase() characters beginning at pbase().

Draft
b) if traits::eq_int_type(c,traits::eof()) returns true, then the empty sequence otherwise, the sequence consisting of c.

Remarks: The member functions sputc() and sputn() call this function in case that no room can be found in the put buffer enough to accomodate 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 pbase() and pptr() shall be set so that: pptr() - pbase() == r and the r characters starting at pbase() are the associated output stream. In case r is zero (all characters of the pending sequence have been consumed) then either pbase() is set to NULL, or pbase() and 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 pbase() and pptr() according to the above rules.

Returns: traits::eof() or throws an exception if the function fails.

Otherwise, returns some value other than traits::eof() to indicate success.

Default behavior: Returns traits::eof().

27.6 Formatting and manipulators

Header <istream> synopsis

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 = char_traits<charT> >
    basic_istream<charT,traits>& ws(basic_istream<charT,traits>& is);
}

Header <ostream> synopsis

namespace std {
    template <class charT, class traits = char_traits<charT> >
    class basic_ostream;
	namespace std {
        template <class charT, class traits = char_traits<charT> >
            class basic_ostream;

306] 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.

307] Typically, overflow returns c to indicate success, except when traits::eq_int_type(c,traits::eof()) returns true, in which case it returns traits::not_eof(c).
typedef basic_ostream<char> ostream;
typedef basic_ostream<wchar_t> wostream;

template <class charT, class traits>
basic_ostream<charT,traits>& endl(basic_ostream<charT,traits>& os);
template <class charT, class traits>
basic_ostream<charT,traits>& ends(basic_ostream<charT,traits>& os);
template <class charT, class traits>
basic_ostream<charT,traits>& flush(basic_ostream<charT,traits>& os);

Header <iomanip> synopsis

namespace std {
  // types T1, T2, ... are unspecified implementation types
  T1 resetiosflags(ios_base::fmtflags mask);
  T2 setiosflags (ios_base::fmtflags mask);
  T3 setbase(int base);
  template<charT> T4 setfill(charT c);
  T5 setprecision(int n);
  T6 setw(int n);
}

27.6.1 Input streams

The header <istream> 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);

    // 27.6.1.1.3 Prefix/suffix:

Draft
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_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&));
basic_istream<charT,traits>& operator>>(
    ios_base& (*pf)(ios_base&));
basic_istream<charT,traits>& operator>>(
    bool& n);
basic_istream<charT,traits>& operator>>(
    short& n);
basic_istream<charT,traits>& operator>>(
    unsigned short& n);
basic_istream<charT,traits>& operator>>(
    int& n);
basic_istream<charT,traits>& operator>>(
    unsigned int& n);
basic_istream<charT,traits>& operator>>(
    long& n);
basic_istream<charT,traits>& operator>>(
    unsigned long& n);
basic_istream<charT,traits>& operator>>(
    long long& n);
basic_istream<charT,traits>& operator>>(
    unsigned long long& n);
basic_istream<charT,traits>& operator>>(
    float& f);
basic_istream<charT,traits>& operator>>(
    double& f);
basic_istream<charT,traits>& operator>>(
    long double& f);
basic_istream<charT,traits>& operator>>(
    void*& p);
basic_istream<charT,traits>& operator>>(
    basic_streambuf<char_type,traits>* sb);

// 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);
basic_istream<charT,traits>& unget();
The class `basic_istream` defines a number of member function signatures that assist in reading and interpreting input from sequences controlled by a stream buffer.

Two groups of member function signatures share common properties: the formatted input functions (or extractors) and the unformatted input functions. Both groups of input functions are described as if they obtain (or extract) input characters by calling `rdbuf()->sbumpc()` or `rdbuf()->sgetc()`. They may use other public members of `istream`.

If `rdbuf()->sbumpc()` or `rdbuf()->sgetc()` returns `traits::eof()`, then the input function, except as explicitly noted otherwise, completes its actions and does `setstate(eofbit)`, which may throw `ios_base::failure` (27.4.4.3), before returning.

If one of these called functions throws an exception, then unless explicitly noted otherwise, the input function sets `badbit` in error state. If `badbit` is on in `exceptions()`, the input function rethrows the exception without completing
its actions, otherwise it does not throw anything and proceeds as if the called function had returned a failure indication.

### 27.6.1.11 basic_istream constructors

**explicit basic_istream(basic_streambuf<charT,traits>* sb);**

**Effects:** Constructs an object of class basic_istream, assigning initial values to the base class by calling basic_ios::init(sb) (27.4.4.1).

**Postcondition:** gcount() == 0

**basic_istream(basic_istream&& rhs);**

**Effects:** Move constructs from the rvalue rhs. This is accomplished by default constructing the base class, copying the gcount() from rhs, calling basic_ios<charT, traits>::move(rhs) to initialize the base class, and setting the gcount() for rhs to 0.

**virtual ~basic_istream();**

**Effects:** Destroys an object of class basic_istream.

**Remarks:** Does not perform any operations of rdbuf().

### 27.6.1.12 Class basic_istream assign and swap

**basic_istream& operator=(basic_istream&& rhs);**

**Effects:** swap(rhs);

**Returns:** *this.

**void swap(basic_istream&& rhs);**

**Effects:** Calls basic_ios<charT, traits>::swap(rhs). Exchanges the values returned by gcount() and rhs.gcount().

**template <class charT, class traits>**

**void swap(basic_istream<charT, traits>& x, basic_istream<charT, traits>&& y);**

**template <class charT, class traits>**

**void swap(basic_istream<charT, traits>&& x, basic_istream<charT, traits>& y);**

**template <class charT, class traits>**

**void swap(basic_istream<charT, traits>& x, basic_istream<charT, traits>&& y);**

**Effects:** x.swap(y).

### 27.6.1.13 Class basic_istream::sentry

**namespace std {**

**template <class charT, class traits = char_traits<charT> >**

**class basic_istream<charT,traits>::sentry {**

**typedef traits traits_type;**

// bool ok_;**

exposition only

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.

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`).

[Note: This paragraph is intentionally empty. — end note]

```cpp
~sentry();
```

Effects: None.

```cpp
operator bool() const;
```

Effects: Returns `ok_`.

---

308 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.

309 The `sentry` constructor and destructor can also perform additional implementation-dependent operations.

Draft
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\(^{310}\) 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  

\[
\begin{align*}
&\text{operator>>(unsigned short& val);} \\
&\text{operator>>(unsigned int& val);} \\
&\text{operator>>(long& val);} \\
&\text{operator>>(unsigned long& val);} \\
&\text{operator>>(long long& val);} \\
&\text{operator>>(unsigned long long& val);} \\
&\text{operator>>(float& val);} \\
&\text{operator>>(double& val);} \\
&\text{operator>>(long double& val);} \\
&\text{operator>>(bool& val);} \\
&\text{operator>>(void*& val);} \\
\end{align*}
\]

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:

\[
\begin{align*}
&\text{typedef num_get< charT,istreambuf_iterator<charT,traits> > numget;} \\
&\text{iostate err = 0;} \\
&\text{use_facet<numget>(loc).get(*this, 0, *this, err, val);} \\
&\text{setstate(err);} \\
\end{align*}
\]

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.

\[
\begin{align*}
&\text{operator>>(short& val);} \\
&\text{operator>>(long& val);} \\
&\text{if(err = 0)} \\
&\text{if (err != 0)} \\
&\text{:}
\end{align*}
\]

\(^{310}\) This is done without causing an ios::failure to be thrown.

Draft
```c
if (lval < numeric_limits<short>::min()
    || numeric_limits<short>::max() < lval))
    err = ios_base::failbit;
else
    val = static_cast<short>(lval);
setstate(err);
```
Returns: *this.

```cpp
template<class charT, class traits>
    basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>&& in, charT* s);
```

```cpp
template<class traits>
    basic_istream<char, traits>& operator>>(basic_istream<char, traits>&& in, unsigned char* s);
```

```cpp
template<class traits>
    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.

```cpp
template<class charT, class traits>
    basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>&& in, charT& c);
```

```cpp
template<class traits>
    basic_istream<char, traits>& operator>>(basic_istream<char, traits>&& in, unsigned char& c);
```

```cpp
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.

```cpp
basic_istream<charT, traits>& operator>>(basic_streambuf<charT, traits>* sb);
```
14 Effects: Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1). If \( sb \) is null, calls \( \text{setstate(failbit)} \), which may throw \( \text{ios::badbit} \) (27.4.4.3). After a sentry object is constructed, extracts characters from \( *\text{this} \) 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).

15 If the function inserts no characters, it calls \( \text{setstate(failbit)} \), which may throw \( \text{ios::badbit} \) (27.4.4.3). If it inserted no characters because it caught an exception thrown while extracting characters from \( *\text{this} \) and \( \text{failbit} \) is on in \( \text{exceptions()} \) (27.4.4.3), then the caught exception is rethrown.

16 Returns: \( *\text{this} \).

27.6.1.3 Unformatted input functions

Each unformatted input function begins execution by constructing an object of class \( \text{sentry} \) with the default argument \( \text{noskipws(second)} \) argument \( \text{true} \). If the \( \text{sentry} \) object returns \( \text{true} \), when converted to a value of type \( \text{bool} \), 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 \( \text{bool} \), 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 \( \text{charT()} \)) in the first location of the array. If an exception is thrown during input then \( \text{ios::badbit} \) is turned on in \( *\text{this}'s \) error state. (Exceptions thrown from \( \text{basic_ios<>::clear()} \) are not caught or rethrown.) If \( (\text{exceptions()}\&\text{badbit}) \neq 0 \) 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 \( \text{sentry} \) object is destroyed before leaving the unformatted input function.

\[
\text{streamsize gcount()} \text{ const;}
\]

2 Effects: None. This member function does not behave as an unformatted input function (as described in 27.6.1.3, paragraph 1).

3 Returns: The number of characters extracted by the last unformatted input member function called for the object.

\[
\text{int_type get();}
\]

4 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 \( \text{setstate(failbit)} \), which may throw \( \text{ios::badbit} \) (27.4.4.3).

5 Returns: \( c \) if available, otherwise \( \text{traits::eof()} \).

\[
\text{basic_istream<\text{charT},\text{traits}> & get(\text{char_type} & c);}\]

313) This is done without causing an \( \text{ios::badbit} \) to be thrown.
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.

```
6 basic_istream<charT,traits>& get(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 any of the following occurs:
- $n$ is less than one or $n - 1$ characters are stored;
- end-of-file occurs on the input sequence (in which case the function calls `setstate(eofbit)`);
- `traits::eq(c, delim)` for the next available input character $c$ (in which case $c$ is not extracted).

If the function stores no characters, it calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)). 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.

```
7 basic_istream<charT,traits>& get(char_type* s, streamsize n,
   char_type delim);
```

Effects: Calls `get(s, n, widen('\n'))`

Returns: Value returned by the call.

```
8 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` (27.4.4.3).

Returns: *this.

```
9 basic_istream<charT,traits>& get(basic_streambuf<char_type,traits>& sb,
   char_type delim);
```

Note that this function is not overloaded on types `signed char` and `unsigned char`.

Note that this function is not overloaded on types `signed char` and `unsigned char`.

Draft
Effects: Calls get(ab, 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.\[316\] Characters are extracted and stored until one of the following occurs:

1. end-of-file occurs on the input sequence (in which case the function calls setstate(eofbit));
2. traits::eq(c, delim) for the next available input character c (in which case the input character is extracted but not stored);\[317\]
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.\[318\]

If the function extracts no characters, it calls setstate(failbit) (which may throw ios_base::failure (27.4.4.3)).\[319\]

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;
        }
    }
```

\[316\] Note that this function is not overloaded on types signed char and unsigned char.

\[317\] Since the final input character is “extracted,” it is counted in the gcount(), even though it is not stored.

\[318\] This allows an input line which exactly fills the buffer, without setting failbit. This is different behavior than the historical AT&T implementation.

\[319\] This implies an empty input line will not cause failbit to be set.

Draft


```c++
} cout << " (" << count << " chars): " << buffer << endl;
}

— end example

basic_istream<charT,traits>& getline(char_type* s, streamsize n);

Returns: getline(s,n,widen(\'\n\'))

basic_istream<charT,traits>&
ignore(streamsize n = 1, int_type delim = traits::eof());

Effects: Behaves as an unformatted input function (as described in 27.6.1.3, 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.

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().

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.\footnote{Note
that this function is not overloaded on types signed char and unsigned char.} 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.

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.

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(). 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,
27.6 Formatting and manipulators

ios_base::in). In case of failure, the function calls setstate(failbit) (which may throw ios_base
failure).

42

Returns: *this.

basic_istream<charT,traits>& seekg(off_type off, ios_base::seekdir dir);

43

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).

44

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);
}

1

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 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.

2

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();
    }
}
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

```cpp
explicit basic_iostream(basic_streambuf<charT,traits>* sb);
```

**Effects:** Constructs an object of class `basic_iostream`, assigning initial values to the base classes by calling `basic_istream<charT,traits>(sb)` (27.6.1.1) 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

```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

```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);
```
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 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

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);

            Draft
basic_ostream<charT,traits>& operator<<(long long n);
basic_ostream<charT,traits>& operator<<(unsigned long long n);
basic_ostream<charT,traits>& operator<<(float f);
basic_ostream<charT,traits>& operator<<(double f);
basic_ostream<charT,traits>& operator<<(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);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&, char);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,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);

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
effects:
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`.

```cpp
virtual ~basic_ostream();
```

**Effects:** Destroys an object of class `basic_ostream`.

**Remarks:** Does not perform any operations on `rdbuf()`.

```cpp
basic_ostream(basic_ostream&& rhs);
```

**Effects:** Move constructs from the rvalue `rhs`. This is accomplished by default constructing the base class and calling `basic_ios<charT, traits>::move(rhs)` to initialize the base class.

### 27.6.2.3 Class basic_ostream assign and swap

```cpp
basic_ostream& operator=((basic_ostream&& rhs);
```

**Effects:** `swap(rhs)`.

**Returns:** `*this`.

```cpp
void swap(basic_ostream&& rhs);
```

**Effects:** Calls `basic_ios<charT, traits>::swap(rhs)`.

```cpp
template <class charT, class traits>
t void swap(basic_ostream<charT, traits>& x, basic_ostream<charT, traits>&& y);
t template <class charT, class traits>
t void swap(basic_ostream<charT, traits>&& x, basic_ostream<charT, traits>& y);
t template <class charT, class traits>
t void swap(basic_ostream<charT, traits>& x, basic_ostream<charT, traits>&& y);
```

**Effects:** `x.swap(y)`.

### 27.6.2.4 Class basic_ostream::sentry

```cpp
draft
```
namespace std {
    template <class charT,class traits = char_traits<charT> >
    class basic_ostream<charT,traits>::sentry {
        // bool ok_; exposition only
    public:
        explicit sentry(basic_ostream<charT,traits>& os);
        ~sentry();
        explicit operator bool() const { return ok_; }
        sentry(const sentry&) = delete;
        sentry& operator=(const sentry&) = delete;
    };
}

The class sentry defines a class that is responsible for doing exception safe prefix and suffix operations.

    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)).

    ~sentry();

    If ((os.flags() & ios_base::unitbuf) && !uncaught_exception()) is true, calls os.flush().

    operator bool() const;

    Effects: Returns ok_.

27.6.2.5 basic_ostream seek members [ostream.seeks]

    pos_type tellp();

    Returns: if fail() != false, returns pos_type(-1) to indicate failure. Otherwise, returns rdbuf()->pubseekoff(0, cur, out).

    basic_ostream<charT,traits>& seekp(pos_type pos);

    Effects: If fail() != true, executes rdbuf()->pubseekpos(pos, ios_base::out). In case of failure, the function calls setstate(failbit) (which may throw ios_base::failure).

    Returns: *this.

    basic_ostream<charT,traits>& seekp(off_type off, ios_base::seekdir dir);

    Effects: If fail() != true, executes rdbuf()->pubseekoff(off, dir, ios_base::out).

---

321) The call os.tie()->flush() does not necessarily occur if the function can determine that no synchronization is necessary.
322) The sentry constructor and destructor can also perform additional implementation-dependent operations.
Returns: \*this.

### 27.6.2.6 Formatted output functions

#### 27.6.2.6.1 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 \(^{323}\) in \*this’s error state. If `(exceptions()&badbit)` \(!= 0` then the exception is rethrown. Whether or not an exception is thrown, the `sentry` object is destroyed before leaving the formatted output function. If no exception is thrown, the result of the formatted output function is \*this.

The descriptions of the individual formatted output 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<<(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:

\[
\begin{align*}
\text{bool failed} &= \text{use_facet<} \\
&\quad \text{num_put<charT,ostreambuf_iterator<charT,traits> >} \\
&\quad \quad \text{getloc()}.put(*\this, *\this, fill(), val).failed();
\end{align*}
\]

When \(val\) is of type `short` the formatting conversion occurs as if it performed the following code fragment:

\[
\begin{align*}
\text{ios_base::fmtflags baseflags} &= \text{ios_base::flags()} \text{ & ios_base::basefield} \\
\text{bool failed} &= \text{use_facet<} \\
&\quad \text{num_put<charT,ostreambuf_iterator<charT,traits> >} \\
&\quad \quad \text{getloc()}.put(*\this, *\this, fill(), \\
&\quad \quad \quad \text{baseflags} \text{=} \text{ios_base::oct} \text{ || baseflag \text{=} ios_base::hex} \\
&\quad \quad \quad \quad \text{? static_cast<long>(static_cast<unsigned short>(val))}
\end{align*}
\]

\(^{323}\) without causing an `ios::failure` to be thrown.

Draft
When `val` is of type `int` the formatting conversion occurs as if it performed the following code fragment:

```cpp
ios_base::fmtflags baseflags = ios_base::flags() & ios_base::basefield;
bool failed = use_facet<
    num_put<charT,ostreambuf_iterator<charT,traits> >
>(getloc()).put(*this, *this, fill(),
    baseflags == ios_base::oct || baseflags == ios_base::hex
    ? static_cast<long>(static_cast<unsigned int>(val))
    : static_cast<long>(val)).failed();
```

When `val` is of type `unsigned short` or `unsigned int` the formatting conversion occurs as if it performed the following code fragment:

```cpp
bool failed = use_facet<
    num_put<charT,ostreambuf_iterator<charT,traits> >
>(getloc()).put(*this, *this, fill(),
    static_cast<unsigned long>(val)).failed();
```

When `val` is of type `float` the formatting conversion occurs as if it performed the following code fragment:

```cpp
bool failed = use_facet<
    num_put<charT,ostreambuf_iterator<charT,traits> >
>(getloc()).put(*this, *this, fill(),
    static_cast<double>(val)).failed();
```

The first argument provides an object of the `ostreambuf_iterator<>` class which is an iterator for class `basic_ostream<>`. It bypasses `ostreams` and uses `streambufs` directly. Class `locale` relies on these types as its interface to `iostreams`, since for flexibility it has been abstracted away from direct dependence on `ostream`. The second parameter is a reference to the base 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<<`  

```cpp
(basic_ostream<charT,traits>& (*pf)(basic_ostream<charT,traits>&))
```

**Effects**: None. Does not behave as a formatted output function (as described in 27.6.2.6.1).

**Returns**: `pf(*this)`.

### 27.6.2.8 `basic_ios::operator<<`  

`basic_ios<charT,traits>& operator<<`  

```cpp
(basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&))
```

See, for example, the function signature `endl(basic_ostream&)` (27.6.2.8).
Effects: Calls pf(*this). This inserter does not behave as a formatted output function (as described in 27.6.2.6.1).

Returns: *this.\(^{325}\)

basic_ostream<charT,traits>& operator<<(ios_base& (*pf)(ios_base&))

Effects: Calls pf(*this). This inserter does not behave as a formatted output function (as described in 27.6.2.6.1).

Returns: *this.

basic_ostream<charT,traits>& operator<<(basic_streambuf<charT,traits>* sb);

Effects: Behaves as an unformatted output function (as described in 27.6.2.7, paragraph 1). After the sentry object is constructed, if sb is null calls setstate(badbit) (which may throw ios_base::failure).

Gets characters from sb and inserts them in *this. Characters are read from sb and inserted until any of the following occurs:

— end-of-file occurs on the input sequence;
— inserting in the output sequence fails (in which case the character to be inserted is not extracted);
— an exception occurs while getting a character from sb.

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

```
template<class charT, class traits>
basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& out, charT c);

template<class charT, class traits>
basic_ostream<charT,traits>& operator<<>{basic_ostream<charT,traits>&& out, charT c);

template<class charT, class traits>
basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& out, char c);

template<class charT, class traits>
basic_ostream<charT,traits>& operator<>{basic_ostream<charT,traits>&& out, char c);

// specialization

template<class traits>
basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out, char c);

template<class traits>
```

\(^{325}\) See, for example, the function signature dec(ios_base&) (27.4.5.3).
basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&& out, char c);

// signed and unsigned
template<class traits>
basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&& out, signed char c);

template<class traits>
basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&& out, unsigned char c);

draft
template<class traits>
basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&& out,
const unsigned char* s);

Requires: s shall not be a null pointer.

Effects: Behaves like a formatted inserter (as described in 27.6.2.6.1) 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<char, 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.327

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.328 Characters are inserted until either of the following occurs:

- n characters are inserted;

326) without causing an ios::failure to be thrown.
327) Note that this function is not overloaded on types signed char and unsigned char.
328) Note that this function is not overloaded on types signed char and unsigned char.

Draft
— inserting in the output sequence fails (in which case the function calls setstate(badbit), which may throw ios_base::failure (27.4.4.3)).

6

Returns: *this.

basic_ostream& flush();

Effects: Behaves as an unformatted output function (as described in 27.6.2.6.1, paragraph 1). If rdbuf() is not a null pointer, constructs a sentry object. If this object returns true when converted to a value of type bool the function calls rdbuf()->pubsync(). If that function returns -1 calls setstate(badbit) (which may throw ios_base::failure (27.4.4.3)). Otherwise, if the sentry object returns false, does nothing. Does not behave as an unformatted output function (as described in 27.6.2.7, paragraph 1).

8

Returns: *this.

27.6.2.8 Standard basic_ostream manipulators [ostream.manip]

namespace std {
    template <class charT, class traits>
    basic_ostream<charT,traits>& endl(basic_ostream<charT,traits>& os);
}

1

Effects: Calls os.put(os.widen('\n')), then os.flush().

2

Returns: os.

namespace std {
    template <class charT, class traits>
    basic_ostream<charT,traits>& ends(basic_ostream<charT,traits>& os);
}

3

Effects: Inserts a null character into the output sequence: calls os.put(charT()).

4

Returns: os.

namespace std {
    template <class charT, class traits>
    basic_ostream<charT,traits>& flush(basic_ostream<charT,traits>& os);
}

5

Effects: Calls os.flush().

6

Returns: os.

27.6.3 Standard manipulators [std.manip]

1

The header <iomanip> defines several functions that support extractors and inserters that alter information maintained by class ios_base and its derived classes.

unspecified resetiosflags(ios_base::fmtflags mask);
Returns: An object of unspecified type such that if out is an object of type basic_ostream<charT, traits>
then the expression out << resetiosflags(mask) behaves as if it called f(out, mask), or if in is an object
of type basic_istream<charT, traits> then the expression in >> resetiosflags(mask) behaves as if it
called f(in, mask), where the function f is defined as:

```cpp
void f(ios_base& str, ios_base::fmtflags mask) {
    // reset specified flags
    str.setf(ios_base::fmtflags(0), mask);
}
```

The expression out << resetiosflags(mask) shall have type basic_ostream<charT, traits>& and value
out. The expression in >> resetiosflags(mask) shall have type basic_istream<charT, traits>& and value in.

 unspecified setiosflags(ios_base::fmtflags mask);

Returns: An object of unspecified type such that if out is an object of type basic_ostream<charT, traits>
then the expression out << setiosflags(mask) behaves as if it called f(out, mask), or if in is an object
of type basic_istream<charT, traits> then the expression in >> setiosflags(mask) behaves as if it
called f(in, mask), where the function f is defined as:

```cpp
void f(ios_base& str, ios_base::fmtflags mask) {
    // set specified flags
    str.setf(mask);
}
```

The expression out << setiosflags(mask) shall have type basic_ostream<charT, traits>& and value
out. The expression in >> setiosflags(mask) shall have type basic_istream<charT, traits>& and value in.

 unspecified setbase(int base);

Returns: An object of unspecified type such that if out is an object of type basic_ostream<charT, traits>
then the expression out << setbase(base) behaves as if it called f(out, base), or if in is an object
of type basic_istream<charT, traits> then the expression in >> setbase(base) behaves as if it called f(in,
base), where the function f is defined as:

```cpp
void f(ios_base& str, int base) {
    // set basefield
    str.setf(base == 8 ? ios_base::oct :
             base == 10 ? ios_base::dec :
             base == 16 ? ios_base::hex :
             ios_base::fmtflags(0), ios_base::basefield);
}
```

The expression out << setbase(base) shall have type basic_ostream<charT, traits>& and value out.
The expression in >> setbase(base) shall have type basic_istream<charT, traits>& and value in.

---

329) 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).
27.6 Formatting and manipulators

unspecified setfill(char_type c);

Returns: An object of unspecified type such that if out is an object of type basic_ostream<charT, traits> and c has type charT then the expression out << setfill(c) behaves as if it called f(out, c), where the function f is defined as:

```cpp
template<class charT, class traits>
void f(basic_ios<charT, traits>& str, charT c) {
    // set fill character
    str.fill(c);
}
```

The expression out << setfill(c) shall have type basic_ostream<charT, traits>& and value out.

unspecified setprecision(int n);

Returns: An object of unspecified type such that if out is an object of type basic_ostream<charT, traits> then the expression out << setprecision(n) behaves as if it called f(out, n), or if in is an object of type basic_istream<charT, traits> then the expression in >> setprecision(n) behaves as if it called f(in, n), where the function f is defined as:

```cpp
void f(ios_base& str, int n) {
    // set precision
    str.precision(n);
}
```

The expression out << setprecision(n) shall have type basic_ostream<charT, traits>& and value out. The expression in >> setprecision(n) shall have type basic_istream<charT, traits>& and value in.

unspecified setw(int n);

Returns: An object of unspecified type such that if out is an instance of basic_ostream<charT, traits> then the expression out << setw(n) behaves as if it called f(out, n), or if in is an object of type basic_istream<charT, traits> then the expression in >> setw(n) behaves as if it called f(in, n), where the function f is defined as:

```cpp
void f(ios_base& str, int n) {
    // set width
    str.width(n);
}
```

The expression out << setw(n) shall have type basic_ostream<charT, traits>& and value out. The expression in >> setw(n) shall have type basic_istream<charT, traits>& and value in.

27.6.4 Extended Manipulators

The header `<iomanip>` defines several functions that support extractors and inserters that allow for the parsing and formatting of sequences and values for money and time.

```cpp
template <class moneyT> unspecified get_money(moneyT& mon, bool intl = false);
```

Draft


2 \textit{Requires:} The type \texttt{moneyT} shall be either \texttt{long double} or a specialization of the \texttt{basic_string} template (Clause 21).

3 \textit{Returns:} An object of unspecified type such that if \texttt{in} is an object of type \texttt{basic_istream<charT, traits>} then the expression \texttt{in >> get\_money(mon, intl)} behaves as if it called \texttt{f(in, mon, intl)}, where the function \texttt{f} is defined as:

\begin{verbatim}
template <class charT, class traits, class moneyT>
void f(basic_istream<charT, traits>& str, moneyT& mon, bool intl) {
    typedef istreambuf_iterator<charT> Iter;
    typedef money_get<charT, Iter> MoneyGet;
    ios_base::iostate err = ios_base::goodbit;
    const MoneyGet &mg = use_facet<MoneyGet>(str.getloc());
    mg.get(Iter(str.rdbuf()), Iter(), intl, str, err, mon);
    if (ios_base::goodbit != err)
        str.setstate(err);
}
\end{verbatim}

The expression \texttt{in >> get\_money(mon, intl)} shall have type \texttt{basic_istream<charT, traits>&} and value \texttt{in}.

\begin{verbatim}
template <class charT, class moneyT> unspecified put\_money(const moneyT& mon, bool intl = false);
\end{verbatim}

4 \textit{Requires:} The type \texttt{moneyT} shall be either \texttt{long double} or a specialization of the \texttt{basic_string} template (Clause 21).

5 \textit{Returns:} An object of unspecified type such that if \texttt{out} is an object of type \texttt{basic_ostream<charT, traits>} then the expression \texttt{out << put\_money(mon, intl)} behaves as if it called \texttt{f(out, mon, intl)}, where the function \texttt{f} is defined as:

\begin{verbatim}
template <class charT, class traits, class moneyT>
void f(basic_ostream<charT, traits>& str, const moneyT& mon, bool intl) {
    typedef ostreambuf_iterator<charT> Iter;
    typedef money_put<charT, Iter> MoneyPut;
    const MoneyPut &mp = use_facet<MoneyPut>(str.getloc());
    const Iter end = mp.put(Iter(str.rdbuf()), intl, str, str.fill(), mon);
    if (end.failed())
        str.setstate(ios::badbit);
}
\end{verbatim}

The expression \texttt{out << put\_money(mon, intl)} shall have type \texttt{basic_ostream<charT, traits>&} and value \texttt{out}.

\begin{verbatim}
template <class charT> unspecified get\_time(struct tm *tmb, const charT *fmt);
\end{verbatim}

6 \textit{Requires:} The argument \texttt{tmb} shall be a valid pointer to an object of type \texttt{struct tm}, and the argument \texttt{fmt} shall be a valid pointer to an array of objects of type \texttt{charT} with \texttt{char_traits<charT>::length(fmt)} elements.

Draft
27.7 String-based streams

Returns: An object of unspecified type such that if \( \text{in} \) is an object of type `basic_istream<charT, traits>` then the expression \( \text{in} >> \) `get_time(tmb, fmt)` behaves as if it called \( f(\text{in}, \text{tmb}, \text{fmt}) \), where the function \( f \) is defined as:

```c++
template <class charT, class traits>
void f(basic_ios<charT, traits>& str, struct tm *tmb, const charT *fmt) {
    typedef streambuf_iterator<charT> Iter;
    typedef time_get<charT, Iter> TimeGet;

    ios_base::iostate err = ios_base::goodbit;
    const TimeGet& tg = use_facet<TimeGet>(str.getloc());

    tm.get(Iter(str.rdbuf()), Iter(), str, err, tmb,
    fmt, fmt + traits::length(fmt));

    if (err != ios_base::goodbit)
        str.setstate(err);
}
```

The expression \( \text{in} >> \) `get_time(tmb, fmt)` shall have type `basic_istream<charT, traits>&` and value \( \text{in} \).

```c++
template <class charT> unspecified put_time(struct tm *tmb, const charT *fmt);
```

Requires: The argument \( \text{tmb} \) shall be a valid pointer to an object of type `struct tm`, and the argument \( \text{fmt} \) shall be a valid pointer to an array of objects of type `charT` with `char_traits<charT>::length(fmt)` elements.

Returns: An object of unspecified type such that if \( \text{out} \) is an object of type `basic_ostream<charT, traits>` then the expression \( \text{out} << \) `put_time(tmb, fmt)` behaves as if it called \( f(\text{out}, \text{tmb}, \text{fmt}) \), where the function \( f \) is defined as:

```c++
template <class charT, class traits>
void f(basic_ios<charT, traits>& str, const struct tm *tmb, const charT *fmt) {
    typedef ostreambuf_iterator<charT> Iter;
    typedef time_put<charT, Iter> TimePut;

    const TimePut& tp = use_facet<TimePut>(str.getloc());
    const Iter end = tp.put(Iter(str.rdbuf()), str, str.fill(), tmb,
    fmt, fmt + traits::length(fmt));

    if (end.failed())
        str.setstate(ios_base::badbit);
}
```

The expression \( \text{out} << \) `put_time(tmb, fmt)` shall have type `basic_istream<charT, traits>&` and value \( \text{out} \).

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;

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`

```cpp
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);
```

Draft
// 27.7.2 Assign and swap:
basic_stringbuf& operator=(basic_stringbuf&& rhs);
void swap(basic_stringbuf& rhs);

// 27.7.3 Get and set:
basic_string<charT,traits,Allocator> str() const;
void str(const basic_string<charT,traits,Allocator>& s);

protected:
// 27.7.4 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;

};

template <class charT, class traits, class Allocator>
void swap(basic_stringbuf<charT, traits, Allocator>& x,
  basic_stringbuf<charT, traits, Allocator>& y);

1 The class basic_stringbuf is derived from basic_streambuf to associate possibly the input sequence and possibly
the output sequence with a sequence of arbitrary characters. The sequence can be initialized from, or made available as,
an object of class basic_string.

2 For the sake of exposition, the maintained data 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

Draft
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() == "".

explicit basic_stringbuf(const basic_string<charT,traits,Allocator>& s,
ios_base::openmode which = ios_base::in | ios_base::out);

Effects: Constructs an object of class basic_stringbuf, initializing the base class with basic_streambuf() (27.5.2.1), and initializing mode with which. Then calls str(s), copies the content of s 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 gpstr() 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(), gpstr(), egptr(), pbase(), pptr(), epptr()) obtain the values which rhs had. Whether they do or not, *this and rhs reference separate buffers (if any at all) after the construction. The openmode, locale and any other state of rhs is also copied.

Postconditions: Let rhs_p refer to the state of rhs just prior to this construction and let rhs_a refer to the state of rhs just after this construction.

- str() == rhs_p.str()
- gpstr() - 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 (gpstr()) gpstr() != 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()
Effects: `swap(rhs)`.

Returns: `*this`.

```cpp
void swap(basic_stringbuf&& rhs);
```

Effects: Exchanges the state of `*this` and `rhs`.

```cpp
template <class charT, class traits, class Allocator>
void swap(basic_stringbuf<charT, traits, Allocator>& x,
          basic_stringbuf<charT, traits, Allocator>& y);
template <class charT, class traits, class Allocator>
void swap(basic_stringbuf<charT, traits, Allocator>&& x,
          basic_stringbuf<charT, traits, Allocator>& y);
template <class charT, class traits, class Allocator>
void swap(basic_stringbuf<charT, traits, Allocator>& x,
          basic_stringbuf<charT, traits, Allocator>&& y);
```

Effects: `x.swap(y)`.

### 27.7.1.3 Member functions

```cpp
basic_string<charT, traits, Allocator> str() const;
```

Returns: A `basic_string` object whose content is equal to the `basic_stringbuf` underlying character sequence. If the `basic_stringbuf` was created only in input mode, the resultant `basic_string` contains the character sequence in the range `[eback(), epgptr())`. If the `basic_stringbuf` was created with which & `ios_base::out` being true then the resultant `basic_string` contains the character sequence in the range `[pbase(), high_mark)`, where `high_mark` represents the position one past the highest initialized character in the buffer. Characters can be initialized by writing to the stream, by constructing the `basic_stringbuf` with a `basic_string`, or by calling the `str(basic_string)` member function. In the case of calling the `str(basic_string)` member function, all characters initialized prior to the call are now considered uninitialized (except for those characters re-initialized by the new `basic_string`). Otherwise the `basic_stringbuf` has been created in neither input nor output mode and a zero length `basic_string` is returned.

```cpp
void str(const basic_string<charT, traits, Allocator>& s);
```

Effects: Copies the content of `s` into the `basic_stringbuf` underlying character sequence and initializes the input and output sequences according to `mode`. If `mode & ios_base::out` is true, initializes the output sequence such that `pbase()` points to the first underlying character, `opptr()` points one past the last underlying character, and `pptr()` is equal to `epptr()` if `mode & ios_base::in` is true, otherwise `pptr()` is equal to `pbase()`.

If `mode & 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.

Postconditions: If `mode & ios_base::out` is true, `pbase()` points to the first underlying character and `epptr() >= pbase() + s.size()` holds; in addition, if `mode & ios_base::in` is true, `pptr() == pbase() + s.data()` holds, otherwise `pptr() == pbase()` is true. If `mode & ios_base::in` is true, `eback()` points to the first underlying character, and both `gptr() == eback()` and `egptr() == eback() + s.size()` hold.
27.7.1.4 Overridden virtual functions

int_type underflow();

Returns: If the input sequence has a read position available, returns traits::to_int_type(*gptr()). Otherwise, returns traits::eof(). Any character in the underlying buffer which has been initialized is considered to be part of the input sequence.

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 input sequence has a putback position available, and if traits::eq(to_char_type(c),gptr()[-1]) returns true, assigns gptr() - 1 to gptr().

  Returns: c.

- If traits::eq_int_type(c,traits::eof()) returns false and if the input sequence has a putback position available, and if mode & ios_base::out is nonzero, assigns c to *--gptr().

  Returns: c.

- If traits::eq_int_type(c,traits::eof()) returns true and if the input sequence has a putback position available, assigns gptr() - 1 to gptr().

  Returns: traits::not_eof(c).

Returns: 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.

int_type overflow(int_type c = traits::eof());

Effects: Appends the character designated by c to the output sequence, if possible, in one of two ways:

- If traits::eq_int_type(c,traits::eof()) returns false and if either the output sequence has a write position available or the function makes a write position available (as described below), the function calls sputc(c).

  Signals success by returning c.

- If traits::eq_int_type(c,traits::eof()) returns true, there is no character to append.

  Signals success by returning a value other than traits::eof().

Remarks: The function can alter the number of write positions available as a result of any call.

Returns: traits::eof() to indicate failure.

The function can make a write position available only if (mode & ios_base::out) != 0. To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements to hold the current array object (if any), plus at least one additional write position. If (mode & ios_base::in) != 0, the function alters the read end pointer egptr() to point just past the new write position.
27.7 String-based streams

```cpp
pos_type seekoff(off_type off, ios_base::seekdir way,
                 ios_base::openmode which
             = ios_base::in | ios_base::out);
```

*Effects:* Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 121.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(which &amp; ios_base::in) == ios_base::in</td>
<td>positions the input sequence</td>
</tr>
<tr>
<td>(which &amp; ios_base::out) == ios_base::out</td>
<td>positions the output sequence</td>
</tr>
<tr>
<td>(which &amp; (ios_base::in</td>
<td>ios_base::out)) == (ios_base::in</td>
</tr>
<tr>
<td>Otherwise</td>
<td>the positioning operation fails.</td>
</tr>
</tbody>
</table>

For a sequence to be positioned, if its next pointer (either gptr() or pptr()) is a null pointer and the new offset newoff is nonzero, the positioning operation fails. Otherwise, the function determines newoff as indicated in Table 122.

<table>
<thead>
<tr>
<th>Condition</th>
<th>newoff Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>way == ios_base::beg</td>
<td>0</td>
</tr>
<tr>
<td>way == ios_base::cur</td>
<td>the next pointer minus the beginning pointer (xnext - xbeg).</td>
</tr>
<tr>
<td>way == ios_base::end</td>
<td>the end high mark pointer minus the beginning pointer (xend high_mark - xbeg).</td>
</tr>
</tbody>
</table>

If (newoff + off) < 0, or if newoff + off refers to an uninitialized character (as defined in 27.7.1.3 paragraph 1), the positioning operation fails. Otherwise, the function assigns xbeg + newoff + off to the next pointer xnext.

*Returns:* pos_type(newoff), constructed from the resultant offset newoff (of type off_type), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is pos_type(off_type(-1)).

```cpp
pos_type seekpos(pos_type sp, ios_base::openmode which
                 = ios_base::in | ios_base::out);
```

Draft
Effects: 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.

```cpp
basic_streambuf<charT,traits>* setbuf(charT* s, streamsize n);
```

Effects: implementation-defined, except that `setbuf(0,0)` has no effect.

Returns: `this`.

### 27.7.2 Class template `basic_istringstream`

```cpp
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_type 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);

        // 27.7.2.3 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
    }
```
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

```cpp
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).

```cpp
explicit basic_istringstream(
    const basic_string<charT,traits,allocator>& str,
    ios_base::openmode which = ios_base::in);
```

*Effects:* Constructs an object of class `basic_istringstream<charT, traits>`, initializing the base class with `basic_istream(&sb)` and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::in)` (27.7.1.1).

```cpp
basic_istringstream(basic_istringstream&& rhs);
```

*Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_stringbuf`. Next `basic_istream<charT,traits>::set_rdbuf(&sb)` is called to install the contained `basic_stringbuf`.

### 27.7.2.2 Assign and swap

```cpp
basic_istringstream& operator=(basic_istringstream&& rhs);
```

*Effects:* `swap(rhs)`.

*Returns:* `*this`.

```cpp
void swap(basic_istringstream&& rhs)
```
Effects: Exchanges the state of *this and rhs by calling basic_istream<charT,traits>::swap(rhs) and sb.swap(rhs).

template <class charT, class traits, class Allocator>
void swap(basic_istringstream<charT, traits, Allocator>& x,
basic_istringstream<charT, traits, Allocator>& y);

Effects: x.swap(y).

27.7.2.3 Member functions

basic_stringbuf<charT,traits,Allocator>* rdbuf() const;

Returns: const_cast<basic_stringbuf<charT,traits,Allocator>*(&sb).

basic_string<charT,traits,Allocator> str() const;

Returns: rdbuf()->str().

void str(const basic_string<charT,traits,Allocator>& s);

Effects: Calls rdbuf()->str(s).

27.7.3 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;
    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);

Draft
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).

**explicit basic_ostringstream(
    const basic_string<charT,traits,Allocator>& str,
    ios_base::openmode which = ios_base::out);**

*Effects:* Constructs an object of class `basic_ostringstream<charT, traits>`, initializing the base class with `basic_ostream(&sb)` and initializing sb with `basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::out)` (27.7.1.1).

```cpp
basic_ostringstream(basic_ostringstream&& rhs);
```
3  

**Effects:** Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_stringbuf`. Next `basic_ostream<charT,traits>::set_rdbuf(&sb)` is called to install the contained `basic_stringbuf`.

### 27.7.3.2 Assign and swap

```cpp
basic_ostringstream& operator=(basic_ostringstream&& rhs);
1  
**Effects:** `swap(rhs)`.
2  
**Returns:** `*this`.
3  
void swap(basic_ostringstream&& rhs);
4  
**Effects:** Exchanges the state of `*this` and `rhs` by calling `basic_ostream<charT,traits>::swap(rhs)` and `sb.swap(rhs.sb)`.
```

```cpp
template <class charT, class traits, class Allocator>
void swap(basic_ostringstream<charT, traits, Allocator>& x,  
  basic_ostringstream<charT, traits, Allocator>& y);
3  
```

```cpp
template <class charT, class traits, class Allocator>
void swap(basic_ostringstream<charT, traits, Allocator>& x,  
  basic_ostringstream<charT, traits, Allocator>&& y);
4  
```

```cpp
template <class charT, class traits, class Allocator>
void swap(basic_ostringstream<charT, traits, Allocator>&& x,  
  basic_ostringstream<charT, traits, Allocator>& y);
5  
```

**Effects:** `x.swap(y)`.

### 27.7.3.3 Member functions

```cpp
basic_stringbuf<charT,traits,Allocator>* rdbuf() const;
1  
**Returns:** `const_cast<basic_stringbuf<charT,traits,Allocator>*(&sb)`.  
```

```cpp
basic_string<charT,traits,Allocator> str() const;
2  
**Returns:** `rdbuf()->str()`.
3  
void str(const basic_string<charT,traits,Allocator>& s);
```

**Effects:** Calls `rdbuf()->str(s)`.

### 27.7.4 Class template `basic_stringstream`

```cpp
namespace std {
  template <class charT, class traits = char_traits<charT>,  
    class Allocator = allocator<charT>>
  class basic_stringstream  
    : public basic_iostream<charT,traits> {
```

Draft
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);
    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
    }

The class template basic_stringstream<charT, traits> supports reading and writing from objects of class basic_string<charT, traits, Allocator>. It uses a basic_stringbuf<charT, traits, Allocator> object to control the associated sequence. For the sake of exposition, the maintained data is presented here as

— sb, the stringbuf object.

27.7.5 basic_stringstream constructors  [stringstream.cons]
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).

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).

basic_stringstream(basic_stringstream&& rhs);

  Effects: Move constructs from the rvalue rhs. This is accomplished by move constructing the base class, and the contained basic_stringbuf. Next basic_istream<charT, traits>::set_rdbuf(&sb) is called to install the contained basic_stringbuf.

27.7.5.1 Assign and swap

basic_stringstream& operator=(basic_stringstream&& rhs);

  Effects: swap(rhs).

Returns: *this.

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).

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).

27.7.6 Member functions

basic_stringbuf<charT, traits, Allocator>* rdbuf() const;

  Returns: const_cast<basic_stringbuf<charT, traits, Allocator>*>(&sb)

basic_string<charT, traits, Allocator> str() const;

Draft
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).\(^\text{330}\)

File A File provides an external source/sink stream whose underlaid character type is char (byte).\(^\text{331}\)

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.

\(^\text{330}\) In C FILE must be a typedef. In C++ it may be a typedef or other type name.

\(^\text{331}\) 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.
27.8.1.1 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();
            virtual void imbue(const locale& loc);
    }
}

template <class charT, class traits>
void swap(basic_filebuf<charT, traits>& x,
    basic_filebuf<charT, traits>& y);

Draft
27.8 File-based streams

The class `basic_filebuf<charT, traits>` associates both the input sequence and the output sequence with a file.

The restrictions on reading and writing a sequence controlled by an object of class `basic_filebuf<charT, traits>` are the same as for reading and writing with the Standard C library `FILE`s.

In particular:
- 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
class codecvt<
    class charT,
    class traits

    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

```cpp
class basic_filebuf;

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(). If an exception occurs during the destruction of the object, including the call to close(), the exception is caught but not rethrown (see 17.4.4.8).

### 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.

```cpp
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

bool is_open() const;

Draft
Returns: true if a previous call to open succeeded (returned a non-null value) and there has been no intervening call to close.

basic_filebuf<charT,traits>* open(const char* s,
ios_base::openmode mode);

Effects: If is_open() != false, returns a null pointer. Otherwise, initializes the filebuf as required. It then opens a file, if possible, whose name is the NTBS s (“as if” by calling std::fopen(s, modstr)). The NTBS modstr is determined from mode & ~ios_base::ate as indicated in Table 123. 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 in out trunc app</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;r&quot;</td>
</tr>
<tr>
<td>+ +</td>
<td>&quot;r+b&quot;</td>
</tr>
<tr>
<td>+</td>
<td>&quot;wb&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;rb&quot;</td>
</tr>
<tr>
<td>+ +</td>
<td>&quot;r+b&quot;</td>
</tr>
<tr>
<td>+</td>
<td>&quot;wb&quot;</td>
</tr>
</tbody>
</table>

If the open operation succeeds and (mode & ios_base::ate) != 0, positions the file to the end (“as if” by calling std::fseek(file, 0, SEEK_END)).

Returns: this if successful, a null pointer otherwise.

basic_filebuf<charT,traits>* open(const string& s,
ios_base::openmode mode);

Returns: open(s.c_str(), mode);

basic_filebuf<charT,traits>* close();

Effects: If is_open() == false, returns a null pointer. If a put area exists, calls overflow(traits::eof()) to flush characters. If the last virtual member function called on *this (between underflow, overflow, seekoff, and seekpos) was overflow then calls a Codecvt::unshift (possibly several times) to determine a termination sequence, inserts those characters and calls overflow(traits::eof()) again. Finally, regardless of whether any of the preceding calls fails or throws an exception, the function closes the file (“as if” by calling

332) 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).
std::fclose(file). If any of the calls to `overflow` or `std::fclose` fails then close fails. If one of these calls throws an exception, the exception is caught and rethrown after closing the file.

### Returns

`std::fclose` returns `true` on success, and `false` otherwise.

### Postcondition

`is_open()` evaluates to `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()`.

_Returns:_ `c`.

---

333) The function signature `fclose(FILE*)` is declared in `<cstdio>` (27.8.2).
— If traits::eq_int_type(c, traits::eof()) returns false and if the function makes a putback position available and if the function is permitted to assign to the putback position, decrements the next pointer for the input sequence, and stores c there.

Returns: c.

— If traits::eq_int_type(c, traits::eof()) returns true, and if either the input sequence has a putback position available or the function makes a putback position available, decrements the next pointer for the input sequence, gptr().

Returns: traits::not_eof(c).

Returns: traits::eof() to indicate failure.

Remarks: If is_open() == false, the function always fails.

The function does not put back a character directly to the input sequence.

If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

int_type overflow(int_type c = traits::eof());

Effects: Behaves according to the description of basic_streambuf<charT, traits>::overflow(c), except that the behavior of “consuming characters” is performed by first converting “as if” by:

```
charT* b = pbase();
charT* p = pptr();
charT* end;
char xbuf[XSIZE];
char* xbuf_end;
codecvt_base::result r =
a_codecvt.out(state, b, p, end, xbuf, xbuf+XSIZE, xbuf_end);
```

and then

— 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.

Returns: traits::not_eof(c) to indicate success, and traits::eof() to indicate failure. If is_open() == false, the function always fails.

basic_streambuf* setbuf(char_type* s, streamsize n);

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 124.

<table>
<thead>
<tr>
<th>way</th>
<th>Value</th>
<th>stdio Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic_ios::beg</td>
<td>SEEK_SET</td>
<td></td>
</tr>
<tr>
<td>basic_ios::cur</td>
<td>SEEK_CUR</td>
<td></td>
</tr>
<tr>
<td>basic_ios::end</td>
<td>SEEK_END</td>
<td></td>
</tr>
</tbody>
</table>

Returns: a newly constructed pos_type object that stores the resultant stream position, if possible. If the positioning operation fails, or if the object cannot represent the resultant stream position, returns pos_type(off_type(-1)).

pos_type seekpos(pos_type sp,
   ios_base::openmode which = ios_base::in | ios_base::out);

Alters the file position, if possible, to correspond to the position stored in sp (as described below). Altering the file position performs as follows:

1. if (om & ios_base::out) != 0, then update the output sequence and write any unshift sequence;
2. set the file position to sp;
3. if (om & ios_base::in) != 0, then update the input sequence;

where om is the open mode passed to the last call to open(). The operation fails if is_open() returns false.

If sp is an invalid stream position, or if the function positions neither sequence, the positioning operation fails. If sp has not been obtained by a previous successful call to one of the positioning functions (seekoff or seekpos) on the same file the effects are undefined.

Returns: sp on success. Otherwise returns pos_type(off_type(-1)).

int sync();

Effects: If a put area exists, calls filebuf::overflow to write the characters to the file. If a get area exists, the effect is implementation-defined.

void imbue(const locale& loc);

Draft
27.8 File-based streams

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

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
            };

        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);
    };

Draft
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();**

*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).

```cpp
effect: 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).
```

```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)`.

```cpp
explicit basic_ifstream(const string& s,
ios_base::openmode mode = ios_base::in);
```

*Effects:*

the same as `basic_ifstream(s.c_str(), mode)`.

```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);**

*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.sb)`.

```cpp
template <class charT, class traits, class Allocator>
void swap(basic_ifstream<charT, traits, Allocator>& x,
```

Draft
27.8 File-based streams

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

basic_filebuf<charT,traits>* rdbuf() const;

Returns: const_cast<basic_filebuf<charT,traits>*>(&sb).

bool is_open() const;

Returns: rdbuf()->is_open().

void open(const char* s, ios_base::openmode mode = ios_base::in);

Effects: Calls rdbuf()->open(s, mode | ios_base::in). If that function does not return a null pointer calls
clear(), otherwise calls setstate(failbit) (which may throw ios_base::failure (27.4.4.3)).

void open(const string& s, ios_base::openmode mode = ios_base::in);

Effects: calls open(s.c_str(), mode).

void close();

Effects: Calls rdbuf()->close() and, if that function returns a null pointer, calls setstate(failbit) (which
may throw ios_base::failure (27.4.4.3)).

27.8.1.10 Class template basic_ofstream

namespace std {

template <class charT, class traits = char_traits<charT> >
class basic_ofstream : public basic_ostream<charT,traits> {
public:
    typedef charT char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits traits_type;

    // 27.8.1.11 Constructors:
    basic_ofstream();
    explicit basic_ofstream(const char* s,
        ios_base::openmode mode = ios_base::out);
    explicit basic_ofstream(const string& s,

Draft
The \texttt{basic\_ofstream} class supports writing to named files. It uses a \texttt{basic\_filebuf} object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

\begin{verbatim}
— sb, the filebuf object.
\end{verbatim}

\subsection{basic\_ofstream constructors} \hfill \textit{[ofstream.cons]}

\begin{verbatim}
basic\_ofstream();

Effects: Constructs an object of class \texttt{basic\_ofstream<charT,traits>}, initializing the base class with \texttt{basic\_ostream(&sb)} and initializing \texttt{sb} with \texttt{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 \texttt{basic\_ofstream<charT,traits>}, initializing the base class with \texttt{basic\_ostream(&sb)} and initializing \texttt{sb} with \texttt{basic\_filebuf<charT,traits>()} (27.6.2.2, 27.8.1.2), then calls \texttt{rdbuf()->open(s, mode|ios_base::out)}. If that function returns a null pointer, calls \texttt{setstate(fail-bit)}.

explicit basic\_ofstream(const string& s, 
\end{verbatim}
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);

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;

Returns: rdbuf()->is_open().

void open(const char* s, ios_base::openmode mode = ios_base::out);

Effects: Calls rdbuf()->open(s, mode | ios_base::out). If that function does not return a null pointer
calls clear(), otherwise calls setstate(failbit) (which may throw ios_base::failure(27.4.4.3)).

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);
        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,
            ios_base::openmode mode = ios_base::in|ios_base::out);
        void close();

    private:
        // basic_filebuf<charT,traits> sb; exposition only
    };

    template <class charT, class traits, 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);
}
```

Draft
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

```cpp
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>().

```cpp
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_istream<&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).

```cpp
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);

```cpp
basic_fstream(basic_fstream&& rhs);
```

**Effects:** Move constructs from the rvalue rhs. This is accomplished by move constructing the base class, and the contained basic_filebuf. Next basic_istream<charT, traits>::set_rdbuf(&sb) is called to install the contained basic_filebuf.

### 27.8.1.16 Assign and swap

```cpp
basic_fstream& operator=(basic_fstream&& rhs);
```

**Effects:** swap(rhs).

**Returns:** *this.

```cpp
void swap(basic_fstream&& rhs);
```

**Effects:** Exchanges the state of *this and rhs by calling basic_istream<charT, traits>::swap(rhs) and sb.swap(rhs.sb).
template <class charT, class traits, class Allocator>
void swap(basic_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

```cpp
basic_filebuf<charT,traits>* rdbuf() const;

Returns: \(\text{const\_cast<basic\_filebuf<charT,traits>*>(sb)}\).
```

```cpp
bool is_open() const;

Returns: \(\text{rdbuf() \text{\texttt{->}} is\_open()}\).
```

```cpp
void open(const char* s,
           ios_base::openmode mode = ios_base::in|ios_base::out);

Effects: Calls \(\text{rdbuf() \text{\texttt{->}} open(s,mode)}\). If that function does not return a null pointer calls \(\text{clear()}\), otherwise calls \(\text{setstate(failbit)}\), (which may throw \(\text{ios\_base::failure}\))\.(27.4.4.3).
```

```cpp
void open(const string& s,
           ios_base::openmode mode = ios_base::in|ios_base::out);

Effects: \(\text{calls open(s.c\_str(), mode)}\);
```

```cpp
void close();

Effects: Calls \(\text{rdbuf() \text{\texttt{->}} close()}\) and, if that function returns returns a null pointer, calls \(\text{setstate(failbit)}\)\.(27.4.4.3) (which may throw \(\text{ios\_base::failure}\)).
```

### 27.8.2 C Library files

Table 125 describes header `<cstdio>`.

SEE ALSO: ISO C subclause 7.9, Amendment 1 subclause 4.6.2.

Table 126 describes header `<cinttypes>`.

---

Draft
### 27.8 File-based streams

#### Table 125: 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>BUFSIZ</td>
<td>FOPEN_MAX SEEK_CUR TMP_MAX _IONBF stdout</td>
</tr>
<tr>
<td>EOF</td>
<td>L_tmpnam SEEK_END _IOFBF stderr</td>
</tr>
<tr>
<td>FILENAME_MAX</td>
<td>NULL <code>&lt;stdio&gt;</code> SEEK_SET _IOLBF stdin</td>
</tr>
<tr>
<td>Types:</td>
<td>FILE fpos_t size_t <code>&lt;stdio&gt;</code></td>
</tr>
</tbody>
</table>
| Functions: | |}

- clearerr
- fopen
- fsetpos
- putc
- setbuf
- vprintf
- fclose
- fprintf
- ftell
- putchar
- setvbuf
- vsprintf
- feof
- fputc
- fwrite
- puts
- snprintf
- vsnprintf
- ferror
- fputs
- getc
- rename
- sprintf
- vsprintf
- fflush
- fread
- getchar
- remove
- tmpfile
- vsscanf
- fgetc
- freopen
- getpos
- rename
- tmpnam
- fgets
- fseek
- printf
- sscanf
- vfprintf

### Table 126: 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>PRI{d i o u x}[FAST LEAST]{8 16 32 64}</td>
<td></td>
</tr>
<tr>
<td>PRI{d i o u x}{MAX PTR}</td>
<td></td>
</tr>
<tr>
<td>SCN{d i o u x}[FAST LEAST]{8 16 32 64}</td>
<td></td>
</tr>
<tr>
<td>SCN{d i o u x}{MAX PTR}</td>
<td></td>
</tr>
<tr>
<td>Types:</td>
<td>imaxdiv_t</td>
</tr>
<tr>
<td>Functions:</td>
<td></td>
</tr>
<tr>
<td>abs</td>
<td>imaxabs strtoimax wcstoiimax</td>
</tr>
<tr>
<td>div</td>
<td>imaxdiv strtoimax wcstoiimax</td>
</tr>
</tbody>
</table>
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:

- **collating element**
  - a sequence of one or more characters within the current locale that collate as if they were a single character.

- **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.

- **format specifier**
  - a sequence of one or more characters that is to be replaced with some part of a regular expression match.

- **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.

- **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.
**28.2 Requirements**

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 127 `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><code>X::char_type</code></td>
<td><code>charT</code></td>
<td>The character container type used in the implementation of class template <code>basic_regex</code>.</td>
</tr>
<tr>
<td><code>X::string_type</code></td>
<td><code>std::basic_string&lt;charT&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>X::locale_type</code></td>
<td>A copy constructible type</td>
<td>A type that represents the locale used by the traits class.</td>
</tr>
<tr>
<td><code>X::char_class_type</code></td>
<td>A bitmask type [17.3.2.1.2].</td>
<td>A bitmask type representing a particular character classification.</td>
</tr>
<tr>
<td><code>X::length(p)</code></td>
<td><code>std::size_t</code></td>
<td>Yields the smallest <code>i</code> such that <code>p[i] == 0</code>. Complexity is linear in <code>i</code>.</td>
</tr>
<tr>
<td><code>v.translate(c)</code></td>
<td><code>X::char_type</code></td>
<td>Returns a character such that for any character <code>d</code> that is to be considered equivalent to <code>c</code> then <code>v.translate(c) == v.translate(d)</code>.</td>
</tr>
<tr>
<td><code>v.translate_nocase(c)</code></td>
<td><code>X::char_type</code></td>
<td>For all characters <code>C</code> that are to be considered equivalent to <code>c</code> when comparisons are to be performed without regard to case, then <code>v.translate_nocase(c) == v.translate_nocase(C)</code>.</td>
</tr>
<tr>
<td>expression</td>
<td>Return Type</td>
<td>Assertion / Note / Pre / Post condition</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>v.transform(F1, F2)</td>
<td>X::string_type</td>
<td>Returns a sort key for the character sequence designated by the iterator range [F1,F2) such that if the character sequence [G1,G2) sorts before the character sequence [H1,H2) then v.transform(G1, G2) &lt; v.transform(H1, H2).</td>
</tr>
<tr>
<td>v.transform_primary(F1, F2)</td>
<td>X::string_type</td>
<td>Returns a sort key for the character sequence designated by the iterator range [F1,F2) such that if the character sequence [G1,G2) sorts before the character sequence [H1,H2) when character case is not considered then v.transform_primary(G1, G2) &lt; v.transform_primary(H1, H2).</td>
</tr>
<tr>
<td>v.lookup_collatename(F1, F2)</td>
<td>X::string_type</td>
<td>Returns a sequence of characters that represents the collating element consisting of the character sequence designated by the iterator range [F1,F2). Returns an empty string if the character sequence is not a valid collating element.</td>
</tr>
<tr>
<td>v.lookup_classname(F1, F2, b)</td>
<td>X::char_class_type</td>
<td>Converts the character sequence designated by the iterator range [F1,F2) into a value of a bitmask type that can subsequently be passed to isctype. Values returned from lookup_classname can be bitwise or'ed together; the resulting value represents membership in either of the corresponding character classes. If b is true, the returned bitmask is suitable for matching characters without regard to their case. Returns 0 if the character sequence is not the name of a character class recognized by X. The value returned shall be independent of the case of the characters in the sequence.</td>
</tr>
<tr>
<td>v.isctype(c, cl)</td>
<td>bool</td>
<td>Returns true if character c is a member of one of the character classes designated by cl; false otherwise.</td>
</tr>
<tr>
<td>v.value(c, I)</td>
<td>int</td>
<td>Returns the value represented by the digit c in base I if the character c is a valid digit in base I; otherwise returns -1. [ Note: the value of I will only be 8, 10, or 16. — end note ]</td>
</tr>
<tr>
<td>u.imbue(loc)</td>
<td>X::locale_type</td>
<td>Imbues u with the locale loc and returns the previous locale used by u if any.</td>
</tr>
<tr>
<td>v.getloc()</td>
<td>X::locale_type</td>
<td>Returns the current locale used by v, if any.</td>
</tr>
</tbody>
</table>

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. ]
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 string::const_iterator> sssub_match;
    typedef sub_match<const 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);
template <typename BiIter, class ST, class SA>
bool operator!=(
    typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

template <typename BiIter, class ST, class SA>
bool operator<(
    typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

template <typename BiIter, class ST, class SA>
bool operator>(
    typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

template <typename BiIter, class ST, class SA>
bool operator>=(
    typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

template <typename BiIter, class ST, class SA>
bool operator<=(
    typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

template <typename BiIter, class ST, class SA>
bool operator==(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const* rhs);

template <typename BiIter, class ST, class SA>
bool operator!=(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const* rhs);

Draft
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);

typename iterator_traits<BiIter>::value_type const* rhs);

bool operator==(const sub_match<BiIter>& lhs,  
                typename iterator_traits<BiIter>::value_type const& rhs);

bool operator!=(typename iterator_traits<BiIter>::value_type const& lhs,  
                const sub_match<BiIter>& rhs);

bool operator>(typename iterator_traits<BiIter>::value_type const& lhs,  
               const sub_match<BiIter>& rhs);

bool operator>=(typename iterator_traits<BiIter>::value_type const& lhs,  
                const sub_match<BiIter>& rhs);

bool operator!=(const sub_match<BiIter>& lhs,  
                typename iterator_traits<BiIter>::value_type const& rhs);

bool operator<(const sub_match<BiIter>& lhs,  
               typename iterator_traits<BiIter>::value_type const& rhs);

bool operator>(const sub_match<BiIter>& lhs,  
               typename iterator_traits<BiIter>::value_type const& rhs);

bool operator>=(const sub_match<BiIter>& lhs,  
                typename iterator_traits<BiIter>::value_type const& rhs);

bool operator<=(const sub_match<BiIter>& lhs,  
                typename iterator_traits<BiIter>::value_type const& rhs);

Draft
28.4 Header `<regex>` synopsis

Regular expressions library

```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> > >

// match_results comparisons

template <class BidirectionalIterator, class Allocator = allocator<sub_match<BidirectionalIterator> > >
bool operator==(const match_results<BidirectionalIterator, Allocator>& m1,
const match_results<BidirectionalIterator, Allocator>& m2);

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;

// 28.10.6, match_results swap:

template <class BidirectionalIterator, class Allocator = allocator<sub_match<BidirectionalIterator> > >
void swap(match_results<BidirectionalIterator, Allocator>& m1,
match_results<BidirectionalIterator, Allocator>& m2);

// 28.11.2, function template regex_match:

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);

// 28.11.2, function template regex_match:

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);

bool regex_match(const basic_string<charT, ST, SA>& s,
match_results<
Draft
typename basic_string<charT, ST, SA>::const_iterator, 
    Allocator>& m,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags = 
    regex_constants::match_default); 

template <class charT, class 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,
    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,
    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

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 std {
    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;
    }
}
```

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 128. 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 std {
    namespace regex_constants {
        typedef bitmask_type 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;
    }
}
```

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 129 for any bitmask elements set.
## Table 128: `syntax_option_type` effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Effect(s) if set</th>
</tr>
</thead>
<tbody>
<tr>
<td>icase</td>
<td>Specifies that matching of regular expressions against a character container sequence shall be performed without regard to case.</td>
</tr>
<tr>
<td>nosubs</td>
<td>Specifies that 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>
Table 129: `regex_constants::match_flag_type` effects when obtaining a match against a character container sequence `[first,last)`.

<table>
<thead>
<tr>
<th>Element</th>
<th>Effect(s) if set</th>
</tr>
</thead>
<tbody>
<tr>
<td>match_not_bol</td>
<td>The first character in the sequence <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;$</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 ECMAScript 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` [re.err]

```cpp
namespace std {
    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;
    }
}
```
28.6 Class regex_error

The type error_type is an implementation defined enumeration type ([17.3.2.1.1]). Values of type error_type represent the error conditions described in table 130:

Table 130: 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

class regex_error : public std::runtime_error {
public:
    explicit regex_error(regex_constants::error_type ecode);
    regex_constants::error_type code() const;
};

The class regex_error defines the type of objects thrown as exceptions to report errors from the regular expression library.

regex_error(regex_constants::error_type ecode);

Effects: Constructs an object of class regex_error.

Postcondition: ecode == code()
regex_constants::error_type code() const;

4   Returns: The error code that was passed to the constructor.

28.7 Class template regex_traits

namespace std {
    template <class charT>
    struct regex_traits {
        public:
            typedef charT char_type;
            typedef std::basic_string<char_type> string_type;
            typedef std::locale locale_type;
            typedef bitmask_type char_class_type;

            regex_traits();
            static std::size_t length(const char_type* p);
            charT translate(charT c) const;
            charT translate_nocase(charT c) const;
            template <class ForwardIterator>
                string_type transform(ForwardIterator first, ForwardIterator last) const;
            template <class ForwardIterator>
                string_type transform_primary(
                    ForwardIterator first, ForwardIterator last) const;
            template <class ForwardIterator>
                string_type lookup_collatename(
                    ForwardIterator first, ForwardIterator last) const;
            template <class ForwardIterator>
                char_class_type lookup_classname(
                    ForwardIterator first, ForwardIterator last, bool icase = false) const;
            bool isctype(charT c, char_class_type f) const;
            int value(charT ch, int radix) const;
            locale_type imbue(locale_type l);
            locale_type getloc() const;
        };
    }
}

1   The specializations regex_traits<char> and regex_traits<wchar_t> shall be valid and shall satisfy the requirements for a regular expression traits class (28.2).

    typedef bitmask_type char_class_type;

2   The type char_class_type is used to represent a character classification and is capable of holding an implementation specific set returned by lookup_classname.

    static std::size_t length(const char_type* p);

3   Returns: char_traits<charT>::length(p);

    charT translate(charT c) const;

Draft
4  Returns: (c).

    charT translate_nocase(charT c) const;

    Returns: use_facet<ctype<charT> >(getloc()).tolower(c).

5  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());

6  template <class ForwardIterator>
    string_type transform_primary(ForwardIterator first, ForwardIterator last) const;

    Effects: if typeid(use_facet<collate<charT> >(getloc())) == 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.

7  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.

8  template <class ForwardIterator>
    char_class_type lookup_classname(
        ForwardIterator first, ForwardIterator last, bool icase = false)) const;

    Returns: an unspecified value that represents the character classification named by the character
    sequence designated by the iterator range [first,last). If the parameter icase is true then the
    returned mask identifies the character classification without regard to the case of the characters being matched,
    otherwise it does honor the case of the characters being matched. 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.

9  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.

10 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'd

334 For example, if the parameter icase is true then 
[[[:lower:]]] is the same as 
[[[:alpha:]].
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 radix shall be 8, 10, or 16.

Returns: the value represented by the digit ch in base radix if the character ch is a valid digit in base radix;
otherwise returns -1.

locale_type imbue(locale_type loc);

Effects: Imbues this with a copy of the locale loc. [Note: calling imbue with a different locale than the one
currently in use invalidates all cached data held by *this. — end note]

Returns: if no locale has been previously imbued then a copy of the global locale in effect at the time of construc-
tion of *this, otherwise a copy of the last argument passed to imbue.

Postcondition: getloc() == loc.

locale_type getloc() const;

Returns: if no locale has been imbued then a copy of the global locale in effect at the time of construction of
*this, otherwise a copy of the last argument passed to imbue.

28.8 Class template basic_regex

For a char-like type charT, specializations of class template basic_regex represent regular expressions constructed
from character sequences of charT characters. In the rest of 28.8, charT denotes a given char-like type. Storage for a
regular expression is allocated and freed as necessary by the member functions of class basic_regex.

Objects of type specialization of basic_regex are responsible for converting the sequence of charT objects to an
internal representation. It is not specified what form this representation takes, nor how it is accessed by algorithms that
operate on regular expressions. [Note: implementations will typically declare some function templates as friends of
basic_regex to achieve this — end note]

The functions described in this clause report errors by throwing exceptions of type regex_error.

namespace std {
  template <class charT,
    class traits = regex_traits<charT> >
  class basic_regex {
    public:
      // types:
      typedef charT value_type;
      typedef 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

Draft
28.8 Class template basic_regex

...
flag_type f = regex_constants::ECMAScript);

// 28.8.4, const operations:
unsigned mark_count() const;
flag_type flags() const;

// 28.8.5, locale:
locale_type imbue(locale_type loc);
locale_type getloc() const;

// 28.8.6, swap:
void swap(basic_regex&);
}
}

28.8.1 basic_regex constants

static 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;

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.
28.8 Class template `basic_regex`

Effects: Constructs an object of class `basic_regex`; the object’s internal finite state machine is constructed from the regular expression contained in the array of `charT` of length `char_traits<charT>::length(p)` whose first element is designated by `p`, and interpreted according to the flags `f`.

Postconditions: `flags()` returns `f`. `mark_count()` returns the number of marked sub-expressions within the expression.

```
basic_regex(const charT* p, size_t len, flag_type f);
```

Requires: `p` shall not be a null pointer.

Throws: `regex_error` if `p` is not a valid regular expression.

Effects: Constructs an object of class `basic_regex`; the object’s internal finite state machine is constructed from the regular expression contained in the sequence of characters `[p,p+len)`, and interpreted according the flags specified in `f`.

Postconditions: `flags()` returns `f`. `mark_count()` returns the number of marked sub-expressions within the expression.

```
basic_regex(const basic_regex& e);
```

Effects: Constructs an object of class `basic_regex` as a copy of the object `e`.

Postconditions: `flags()` and `mark_count()` return `e.flags()` and `e.mark_count()`, respectively.

```
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.

Effects: Constructs an object of class `basic_regex`; the object’s internal finite state machine is constructed from the regular expression contained in the string `s`, and interpreted according to the flags specified in `f`.

Postconditions: `flags()` returns `f`. `mark_count()` returns the number of marked sub-expressions within the expression.

```
template <class ForwardIterator>
basic_regex(ForwardIterator first, ForwardIterator last,
    flag_type f = regex_constants::ECMAScript);
```

Throws: `regex_error` if the sequence `[first,last)` is not a valid regular expression.

Effects: Constructs an object of class `basic_regex`; the object’s internal finite state machine is constructed from the regular expression contained in the sequence of characters `[first,last)`, and interpreted according to the flags specified in `f`.

Postconditions: `flags()` returns `f`. `mark_count()` returns the number of marked sub-expressions within the expression.

```
basic_regex& operator=(const basic_regex& e);
```

Effects: Returns the result of `assign(e)`.

Draft
basic_regex& operator=(const charT* ptr);
  
  Requires: ptr shall not be a null pointer.
  Effects: Returns the result of assign(ptr).

  template <class ST, class SA>
  basic_regex& operator=(const basic_string<charT, ST, SA>& p);
  
  Effects: Returns the result of assign(p).

### 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 shall satisfy the requirements for an Input Iterator (24.1.1).

  Returns: assign(string_type(first, last), f).

### 28.8.4 basic_regex constant operations

unsigned mark_count() const;
28.9  **Class template** sub_match

Effects: Returns the number of marked sub-expressions within the regular expression.

    flag_type flags() const;

Effects: Returns a copy of the regular expression syntax flags that were passed to the object’s constructor or to the last call to assign.

**28.8.5  basic_regex locale**

locale_type imbue(locale_type loc);

Effects: Returns the result of traits_inst.imbue(loc) where traits_inst is a (default initialized) instance of the template type argument traits stored within the object. After a call to imbue the basic_regex object does not match any character sequence.

locale_type getloc() const;

Effects: Returns the result of traits_inst.getloc() where traits_inst is a (default initialized) instance of the template parameter traits stored within the object.

**28.8.6  basic_regex swap**

void swap(basic_regex& e);

Effects: Swaps the contents of the two regular expressions.

Postcondition: *this contains the regular expression that was in e, e contains the regular expression that was in *this.

Complexity: constant time.

**28.8.7  basic_regex non-member functions**

**28.8.7.1  basic_regex non-member swap**

    template <class charT, class traits>
    void swap(basic_regex<charT, traits>& lhs, basic_regex<charT, traits>& rhs);

Effects: Calls lhs.swap(rhs).

**28.9  Class template** sub_match

Class template sub_match denotes the sequence of characters matched by a particular marked sub-expression.

    namespace std {
        template <class BidirectionalIterator>
        class sub_match : public std::pair<BidirectionalIterator, BidirectionalIterator> {
            public:
                typedef typename iterator_traits<BidirectionalIterator>::
                    value_type value_type;
    
Draft
```cpp
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: \((\text{matched} \ ? \ \text{distance}(\text{first}, \text{second}) : 0)\).

- **operator basic_string<value_type>() const;**

  Returns: \(\text{matched} \ ? \ \text{basic_string<value_type>}(\text{first}, \text{second}) : \text{basic_string<value_type>()}\).

- **basic_string<value_type> str() const;**

  Returns: \(\text{matched} \ ? \ \text{basic_string<value_type>}(\text{first}, \text{second}) : \text{basic_string<value_type>()}\).

- **int compare(const sub_match& s) const;**

  Returns: \(\text{str()}.\text{compare}(s.\text{str()})\).

- **int compare(const basic_string<value_type>& s) const;**

  Returns: \(\text{str()}.\text{compare}(s)\).

- **int compare(const value_type* s) const;**

  Returns: \(\text{str()}.\text{compare}(s)\).

### 28.9.2 sub_match non-member operators

```cpp
template <class BiIter>
bool operator==(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

Returns: \(\text{lhs.\text{compare}(\text{rhs})} == 0\).

```cpp
template <class BiIter>
bool operator!=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

Returns: \(\text{lhs.\text{compare}(\text{rhs})} != 0\).

Draft
template <class BiIter>
bool operator<(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: \( \text{lhs.compare(rhs)} < 0 \).

template <class BiIter>
bool operator<=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: \( \text{lhs.compare(rhs)} \leq 0 \).

template <class BiIter>
bool operator>=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: \( \text{lhs.compare(rhs)} \geq 0 \).

template <class BiIter>
bool operator>(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: \( \text{lhs.compare(rhs)} > 0 \).

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: \( \text{lhs == rhs.str()} \).

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: \( \text{lhs != rhs.str()} \).

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: \( \text{lhs < rhs.str()} \).

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: \( \text{lhs > rhs.str()} \).
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 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.
28.9 Class template sub_match

Returns: lhs.str() <= rhs.

```cpp
template <class BiIter>
bool operator==(typename iterator_traits<BiIter>::value_type const* lhs,
               const sub_match<BiIter>& rhs);
```

Returns: lhs == rhs.str().

```cpp
template <class BiIter>
bool operator!=(typename iterator_traits<BiIter>::value_type const* lhs,
               const sub_match<BiIter>& rhs);
```

Returns: lhs != rhs.str().

```cpp
template <class BiIter>
bool operator<(typename iterator_traits<BiIter>::value_type const* lhs,
               const sub_match<BiIter>& rhs);
```

Returns: lhs < rhs.str().

```cpp
template <class BiIter>
bool operator<=(typename iterator_traits<BiIter>::value_type const* lhs,
                const sub_match<BiIter>& rhs);
```

Returns: lhs <= rhs.str().

```cpp
template <class BiIter>
bool operator==(const sub_match<BiIter>& lhs,
               typename iterator_traits<BiIter>::value_type const* rhs);
```

Returns: lhs.str() == rhs.

```cpp
template <class BiIter>
bool operator!=(const sub_match<BiIter>& lhs,
               typename iterator_traits<BiIter>::value_type const* rhs);
```

Returns: lhs.str() != rhs.

```cpp
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.

template <class BiIter>
bool operator==(typename iterator_traits<BiIter>::value_type const& lhs,
    const sub_match<BiIter>& rhs);

Returns: basic_string<typename iterator_traits<BiIter>::value_type>(1, lhs) == rhs.str().

template <class BiIter>
bool operator!=(typename iterator_traits<BiIter>::value_type const& lhs,
    const sub_match<BiIter>& rhs);

Returns: basic_string<typename iterator_traits<BiIter>::value_type>(1, lhs) != rhs.str().

template <class BiIter>
bool operator<(typename iterator_traits<BiIter>::value_type const& lhs,
    const sub_match<BiIter>& rhs);

Returns: basic_string<typename iterator_traits<BiIter>::value_type>(1, lhs) < rhs.str().

template <class BiIter>
bool operator>(typename iterator_traits<BiIter>::value_type const& lhs,
    const sub_match<BiIter>& rhs);

Returns: basic_string<typename iterator_traits<BiIter>::value_type>(1, lhs) > rhs.str().

template <class BiIter>
bool operator>=(typename iterator_traits<BiIter>::value_type const& lhs,
    const sub_match<BiIter>& rhs);

Returns: basic_string<typename iterator_traits<BiIter>::value_type>(1, lhs) >= rhs.str().

template <class BiIter>
bool operator<=(typename iterator_traits<BiIter>::value_type const& lhs,
    const sub_match<BiIter>& rhs);

Returns: basic_string<typename iterator_traits<BiIter>::value_type>(1, lhs) <= rhs.str().
28.10 Class template match_results

bool operator==(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

    Returns: lhs.str() == basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs).

template <class BiIter>
bool operator!=(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

    Returns: lhs.str() != basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs).

template <class BiIter>
bool operator<(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

    Returns: lhs.str() < basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs).

template <class BiIter>
bool operator>(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

    Returns: lhs.str() > basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs).

template <class BiIter>
bool operator>=(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

    Returns: lhs.str() >= basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs).

template <class BiIter>
bool operator<=(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

    Returns: lhs.str() <= basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs).

template <class charT, class ST, class BiIter>
basic_ostream<charT, ST>&
    operator<<(basic_ostream<charT, ST>& os, const sub_match<BiIter>& m);

    Returns: (os << m.str()).

28.10 Class template match_results

Class template match_results denotes a collection of character sequences representing the result of a regular expression match. Storage for the collection is allocated and freed as necessary by the member functions of class template match_results.

The class template match_results shall satisfy the requirements of a sequence container, as specified in [23.1.1], except that only operations defined for const-qualified sequence containers 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]

namespace std {
    template <class BidirectionalIterator,
             class Allocator = allocator<sub_match<BidirectionalIterator> >
    class match_results {
    public:
        typedef sub_match<BidirectionalIterator> value_type;
        typedef typename Allocator::const_reference const_reference;
        typedef const_reference reference;
        typedef const_iterator iterator;
        typedef typename iterator_traits<BidirectionalIterator>::
            implementation_defined difference_type;
        typedef typename Allocator::size_type size_type;
        typedef Allocator allocator_type;
        typedef typename iterator_traits<BidirectionalIterator>::
            value_type char_type;
        typedef basic_string<char_type> string_type;

        // 28.10.1, construct/copy/destroy:
        explicit match_results(const Allocator& a = Allocator());
        match_results(const match_results& m);
        match_results& 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;
        const_iterator cbegin() const;
        const_iterator cend() const;

        // 28.10.4, format:
        template <class OutputIter>
        OutputIter
28.10 Class template match_results

```cpp
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;

// 28.10.5, allocator:
allocator_type get_allocator() const;

// 28.10.6, swap:
void swap(match_results& that);
```
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 [re.results.acc]

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;
const_iterator cbegin() const;

Returns: A starting iterator that enumerates over all the sub-expressions stored in *this.

const_iterator end() const;
const_iterator cend() const;

Draft
28.10 Class template `match_results`  

Regular expressions library 1074

Returns: A terminating iterator that enumerates over all the sub-expressions stored in `*this`.

### 28.10.4 `match_results formatting`

```cpp
template <class OutputIter>
OutputIter format(OutputIter out,
const string_type& fmt,
regex_constants::match_flag_type flags =
regex_constants::format_default) const;
```

Requires: The type `OutputIter` shall satisfy the requirements for an Output Iterator (24.1.2).

Effects: Copies the character sequence `[fmt.begin(), fmt.end())` to `OutputIter` `out`. Replaces each format specifier or escape sequence in `fmt` with either the character(s) it represents or the sequence of characters within `*this` to which it refers. The bitmasks specified in `flags` determines what format specifiers and escape sequences are recognized.

Returns: `out`.

```cpp
string_type format(const string_type& fmt,
regex_constants::match_flag_type flags =
regex_constants::format_default) const;
```

Effects: Returns a copy of the string `fmt`. Replaces each format specifier or escape sequence in `fmt` with either the character(s) it represents or the sequence of characters within `*this` to which it refers. The bitmasks specified in `flags` determines what format specifiers and escape sequences are recognized.

### 28.10.5 `match_results allocator`

```cpp
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`

```cpp
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.

```cpp
template <class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1,
match_results<BidirectionalIterator, Allocator>& m2);
```

Effects: `m1.swap(m2)`.
28.10.7 match-results non-member functions [re.results.nonmember]

template <class BidirectionalIterator, class Allocator>
bool operator==(const match_results<BidirectionalIterator, Allocator>& m1,
               const match_results<BidirectionalIterator, Allocator>& m2);

Returns: true only if the two objects refer to the same match.

template <class BidirectionalIterator, class Allocator>
bool operator!=(const match_results<BidirectionalIterator, Allocator>& m1,
               const match_results<BidirectionalIterator, Allocator>& m2);

Returns: !(m1 == m2).

28.11 Regular expression algorithms [re.alg]

28.11.1 exceptions [re.except]

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 [re.alg.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 132.

Table 132: 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>
</tbody>
</table>
28.11 Regular expression algorithms

<table>
<thead>
<tr>
<th>m.suffix().second</th>
<th>last</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

| m[n].first | For all integers n < 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. |
| m[n].second | For all integers n < 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. |
| m[n].matched | For all integers n < m.size(), true if sub-expression n participated in the match, false otherwise. |

```
template <class BidirectionalIterator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags =
                 regex_constants::match_default);
```

**Effects:** Behaves “as if” by constructing an instance of match_results<BidirectionalIterator> what, and then returning the result of regex_match(first, last, what, e, flags).

```
template <class charT, class Allocator, class traits>
bool regex_match(const charT* str,
                 match_results<const charT*, Allocator>& m,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags =
                 regex_constants::match_default);
```

**Returns:** regex_match(str, str + char_traits<charT>::length(str), m, e, flags).

```
template <class ST, class SA, class Allocator, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
                 match_results<typename basic_string<charT, ST, SA>::const_iterator,
                 Allocator>& m,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags =
                 regex_constants::match_default);
```

**Returns:** regex_match(s.begin(), s.end(), m, e, flags).

```
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);
```

Draft
template <class ST, class SA, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags =
                 regex_constants::match_default);  

Returns: regex_match(s.begin(), s.end(), e, flags).

28.11.3 regex_search

Table 133: Effects of regex_search algorithm

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>m.size()</td>
<td>1 + e.mark_count()</td>
</tr>
<tr>
<td>m.empty()</td>
<td>false</td>
</tr>
<tr>
<td>m.prefix().first</td>
<td>first</td>
</tr>
<tr>
<td>m.prefix().second</td>
<td>m[0].first</td>
</tr>
<tr>
<td>m.prefix().matched</td>
<td>m.prefix().first != m.prefix().second</td>
</tr>
<tr>
<td>m.suffix().first</td>
<td>m[0].second</td>
</tr>
<tr>
<td>m.suffix().second</td>
<td>last</td>
</tr>
<tr>
<td>m.suffix().matched</td>
<td>m.suffix().first != m.suffix().second</td>
</tr>
<tr>
<td>m[0].first</td>
<td>The start of the sequence of characters that matched the regular expression</td>
</tr>
<tr>
<td>m[0].second</td>
<td>The end of the sequence of characters that matched the regular expression</td>
</tr>
<tr>
<td>m[0].matched</td>
<td>true 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>
</tbody>
</table>
For all integers \( n < m.\text{size()} \), the end of the sequence that matched sub-expression \( n \). Alternatively, if sub-expression \( n \) did not participate in the match, then \( \text{last} \).

For all integers \( n < m.\text{size()} \), true if sub-expression \( n \) participated in the match, false otherwise.

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 \( \text{regex_search(str, str + \text{char_traits<charT>::length(str)}, m, e, flags)} \).

template <class ST, class SA, class Allocator, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
    match_results<typename basic_string<charT, ST, SA>::const_iterator,
    Allocator>& m,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags =
    regex_constants::match_default);

Returns: The result of \( \text{regex_search(s.begin(), s.end(), m, e, flags)} \).

template <class BidirectionalIterator, class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags =
    regex_constants::match_default);

Effects: Behaves “as if” by constructing an object what of type match_results<BidirectionalIterator> and then returning the result of \( \text{regex_search(first, last, what, e, flags)} \).

template <class charT, class traits>
bool regex_search(const charT* str,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags =
    regex_constants::match_default);

Returns: \( \text{regex_search(str, str + \text{char_traits<charT>::length(str)}, e, flags)} \).

template <class ST, class SA, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags =
    regex_constants::match_default);

Returns: \( \text{regex_search(s.begin(), s.end(), e, flags)} \).
28.11.4 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);

Effects: Constructs a regex_iterator object \(i\) as if by
regex_iterator<BidirectionalIterator, charT, traits> i(first, last, e, flags), and uses \(i\) to enumerate through all of the matches \(m\) of type match_results<BidirectionalIterator> that occur within the sequence \([first, last)\). If no such matches are found and !(flags & regex_constants::format_no_copy) then calls 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.

Returns: out.

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

28.12.1 Class template regex_iterator

The class template regex_iterator is an iterator adaptor. It represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence. A regex_iterator uses regex_search to find successive regular expression matches within the sequence from which it was constructed. After the iterator is constructed, and every time operator++ is used, the iterator finds and stores a value of match_results<BidirectionalIterator>. If the end of the sequence is reached (regex_search returns false), the iterator becomes equal to the end-of-sequence iterator value. The default constructor constructs an end-of-sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of operator* on an end-of-sequence iterator is not defined. For any other iterator value a const match_results<BidirectionalIterator>& is returned. The result of operator-> on an end-of-sequence iterator is not defined. For any other iterator value a const
match_results<BidirectionalIterator>* is returned. It is impossible to store things into regex_iterators. Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.

namespace std {
  template <class BidirectionalIterator,
           class charT = typename iterator_traits<BidirectionalIterator>::value_type,
           class traits = regex_traits<charT> >
  class regex_iterator {
    public:
      typedef basic_regex<charT, traits> regex_type;
      typedef match_results<BidirectionalIterator> value_type;
      typedef std::ptrdiff_t difference_type;
      typedef const value_type* pointer;
      typedef const value_type& reference;
      typedef std::forward_iterator_tag iterator_category;

      regex_iterator();
      regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
                     const regex_type& re,
                     regex_constants::match_flag_type m =
                     regex_constants::match_default);
      regex_iterator(const regex_iterator&);
      regex_iterator& operator=(const regex_iterator&);
      bool operator==(const regex_iterator&) const;
      bool operator!=(const regex_iterator&) const;
      const value_type& operator*() const;
      const value_type* operator->() const;
      regex_iterator& operator++();
      regex_iterator operator++(int);
    }
  private:
    // 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;
  };
}

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) const;

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) const;

Returns: !(const regex_iterator& right) const;

28.12.1.3 regex_iterator dereference

const value_type& operator*() const;

Returns: match.

const value_type* operator->() const;

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]

```cpp
regex_iterator operator++(int);
```

Effects:
```
regex_iterator tmp = *this;
++(*this);
return tmp;
```

28.12.2 Class template `regex_token_iterator` [re.tokiter]

The class template `regex_token_iterator` is an iterator adaptor; that is to say it represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence, and presenting one or more sub-expressions for each match found. Each position enumerated by the iterator is a `sub_match` class template instance that represents what matched a particular sub-expression within the regular expression.

When class `regex_token_iterator` is used to enumerate a single sub-expression with index `-1` the iterator performs field splitting: that is to say it enumerates one sub-expression for each section of the character container sequence that does not match the regular expression specified.

After it is constructed, the iterator finds and stores a value `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.

If the end of sequence is reached (position is equal to the end of sequence iterator), the iterator becomes equal to the end-of-sequence iterator value, unless the sub-expression being enumerated has index `-1`, in which case the iterator enumerates one last sub-expression that contains all the characters from the end of the last regular expression match to the end of the input sequence being enumerated, provided that this would not be an empty sub-expression.

The default constructor constructs an end-of-sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end-of-sequence iterator is not defined. For any other iterator value a `const sub_match<BidirectionalIterator>&` is returned. The result of `operator->` on an end-of-sequence iterator is not defined. For any other iterator value a `const sub_match<BidirectionalIterator>&` is returned.

It is impossible to store things into `regex_iterators`. Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.
namespace std {
    template <class BidirectionalIterator,
             class charT = typename iterator_traits<BidirectionalIterator>::value_type,
             class traits = regex_traits<charT> >
    class regex_token_iterator {
public:
    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);

    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&) const;
    bool operator!=(const regex_token_iterator&) const;
    const value_type& operator*() const;
    const value_type* operator->() const;
    regex_token_iterator& operator++();
    regex_token_iterator operator++(int);

    private: // 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.
In a suffix iterator the member `result` holds a pointer to the data member `suffix`, the value of the member `suffix.match` is true, `suffix.first` points to the beginning of the final sequence, and `suffix.second` points to the end of the final sequence.

[Note: for a suffix iterator, data member `suffix.first` is the same as the end of the last match found, and `suffix.second` is the same as the end of the target sequence — end note]

The current match is (`*position`).prefix() if `subs[N] == -1`, or (`*position)[subs[N]]` for any other value of `subs[N]`.

### 28.12.2.1 regex_token_iterator constructors

```cpp
regex_token_iterator();
```

**Effects:** Constructs the end-of-sequence iterator.

```cpp
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b, const regex_type& re, int submatch = 0, regex_constants::match_flag_type m = regex_constants::match_default);
```

```cpp
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b, const regex_type& re, const std::vector<int>& submatches, regex_constants::match_flag_type m = regex_constants::match_default);
```

```cpp
template <std::size_t N>
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b, const regex_type& re, const int (&submatches)[N], regex_constants::match_flag_type m = regex_constants::match_default);
```

**Requires:** Each of the initialization values of `submatches` shall be `>= -1`.

**Effects:** The first constructor initializes the member `subs` to hold the single value `submatch`. The second constructor initializes the member `subs` to hold a copy of the argument `submatches`. The third 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

```cpp
bool operator==(const regex_token_iterator& right) const;
```

Draft
Returns: true if *this and right are both end-of-sequence iterators, or if *this and right are both suffix iterators and suffix == right.suffix; otherwise returns false if *this or right is an end-of-sequence iterator or a suffix iterator. Otherwise returns true if position == right.position, N == right.N, and subs == right.subs. Otherwise returns false.

bool operator!=(const regex_token_iterator& right) const;

Returns: !(this == right).

28.12.2.3 regex_token_iterator dereference

const value_type& operator*() const;

Returns: *result.

const value_type* operator->() const;

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 subs is equal to -1 and prev->suffix().length() is not 0 the operator sets *this to a suffix iterator that points to the range [prev->suffix().first, prev->suffix().second). Otherwise, sets *this to an end-of-sequence iterator.

Returns: *this

regex_token_iterator& operator++(int);

Effects: Constructs a copy tmp of *this, then calls ++(*this).

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.

3 The following productions within the ECMAScript grammar are modified as follows:

```
CharacterClass ::
    [ [lookahead \{]* ClassRanges ]
    [ ~ ClassRanges ]

ClassAtom ::
    -
    ClassAtomNoDash
    ClassAtomExClass
    ClassAtomCollatingElement
    ClassAtomEquivalence
```

4 The following new productions are then added:

```
ClassAtomExClass ::
    [: ClassName :]

ClassAtomCollatingElement ::
    [ . ClassName .]

ClassAtomEquivalence ::
    [= ClassName =]

ClassName ::
    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 128.

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 ClassNames are determined by the type of the traits class, but at least the following names shall be recognized: alnum, alpha, blank, cntrl, digit, graph, lower, print, punct, space, upper, xdigit, d, s, w. In addition the following expressions shall be equivalent:

\d and [:digit:]

\D and [^[:digit:]]

\s and [:space:]

Draft
\S and `^[[:space:]]`
\w and `_[[:alnum:]]`
\W and `[^_[:alnum:]]`

A ClassName production when used in a ClassAtomCollatingElement production is not valid if the value returned by \texttt{traits\_inst.lookup\_collatename} for that name is an empty string.

The results from multiple calls to \texttt{traits\_inst.lookup\_classname} can be bitwise OR'ed together and subsequently passed to \texttt{traist\_inst.istctype}.

A ClassName production when used in a ClassAtomEquivalence production is not valid if the value returned by \texttt{traits\_inst.lookup\_collatename} for that name is an empty string or if the value returned by \texttt{traits\_inst.transform\_primary} for the result of the call to \texttt{traits\_inst.lookup\_collatename} is an empty string.

When the sequence of characters being transformed to a finite state machine contains an invalid class name the translator shall throw an exception object of type \texttt{regex\_error}.

If the CV of a \texttt{UnicodeEscapeSequence} is greater than the largest value that can be held in an object of type \texttt{charT} the translator shall throw an exception object of type \texttt{regex\_error}. \textit{Note:} this means that values of the form "uxxxx" that do not fit in a character are invalid. — end note

When the regular expression grammar requires the conversion of a sequence of characters to an integral value, this is accomplished by calling \texttt{traits\_inst.value}.

The behavior of the internal finite state machine representation when used to match a sequence of characters is as described in ECMA-262. The behavior is modified according to any match_flag_type flags specified when using the regular expression object in one of the regular expression algorithms. 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 \((\text{flags()} \& \text{regex\_constants::icase})\) the two characters are equal if \(\text{traits\_inst.translate\_nocase}(c) == \text{traits\_inst.translate\_nocase}(d)\);
2. otherwise, if \(\text{flags()} \& \text{regex\_constants::collate}\) the two characters are equal if \(\text{traits\_inst.translate}(c) == \text{traits\_inst.translate}(d)\);
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 \(\text{flags()} \& \text{regex\_constants::collate}\) is false then the character \(c\) is matched if \(c1 <= c \&\& c <= c2\), otherwise \(c\) is matched in accordance with the following algorithm:

```
string\_type str1 = string\_type(1,
flags() \& icase ?
    traits\_inst.translate\_nocase(c1) : traits\_inst.translate(c1);
```

```Draft```
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, flags() & icase)) is true.
Chapter 29   Atomic operations library  [atomics]

1 This clause describes components for fine-grained atomic access. This access is provided via operations on atomic objects.\textsuperscript{335}

2 The following subclauses describe atomics requirements and components for types and operations, as summarized below.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.1 Order and Consistency</td>
<td></td>
</tr>
<tr>
<td>29.2 Lock-free Property</td>
<td></td>
</tr>
<tr>
<td>29.3 Flag Type and Operations</td>
<td></td>
</tr>
<tr>
<td>29.4.1 Integral Types</td>
<td>\texttt{&lt;cstdatomic&gt;}, \texttt{&lt;stdatomic.h&gt;}</td>
</tr>
<tr>
<td>29.4.1 Address Type</td>
<td></td>
</tr>
<tr>
<td>29.4.3 Generic Types</td>
<td></td>
</tr>
<tr>
<td>29.4.4 Operations</td>
<td></td>
</tr>
</tbody>
</table>

\textbf{Header} \texttt{<cstdatomic> synopsis}

namespace std {
    // 29.1. order and consistency
    enum memory_order;

    // 29.2. lock-free property
    #define ATOMIC_INTEGRAL_LOCK_FREE unspecified
    #define ATOMIC_ADDRESS_LOCK_FREE unspecified

    // 29.3. flag type and operations
    struct atomic_flag;
    bool atomic_flag_test_and_set(volatile atomic_flag*);
    bool atomic_flag_test_and_set_explicit(volatile atomic_flag*, memory_order);
    void atomic_flag_clear(volatile atomic_flag*);
    void atomic_flag_clear_explicit(volatile atomic_flag*, memory_order);
    void atomic_flag_fence(const volatile atomic_flag*, memory_order);

    #define ATOMIC_FLAG_INIT unspecified
    extern const atomic_flag atomic_global_fence_compatibility;

335) Atomic objects are neither active nor radioactive.
// 29.4.1, integral types
struct atomic_bool;
bool atomic_is_lock_free(const volatile atomic_bool*);
void atomic_store(volatile atomic_bool*, bool);
void atomic_store_explicit(volatile atomic_bool*, bool, memory_order);
bool atomic_load(volatile atomic_bool*);
bool atomic_load_explicit(volatile atomic_bool*, memory_order);
bool atomic_swap(volatile atomic_bool*);
bool atomic_swap_explicit(volatile atomic_bool*, bool);
bool atomic_compare_swap(volatile atomic_bool*, bool*, bool);
bool atomic_compare_swap_explicit(volatile atomic_bool*, bool*, bool, memory_order, memory_order);
void atomic_fence(const volatile atomic_bool*, memory_order);

// For each of the integral types:
struct atomic_itype;
bool atomic_is_lock_free(const volatile atomic_itype*);
void atomic_store(volatile atomic_itype*, integral);
void atomic_store_explicit(volatile atomic_itype*, integral, memory_order);
integral atomic_load(volatile atomic_itype*);
integral atomic_load_explicit(volatile atomic_itype*, memory_order);
integral atomic_swap(volatile atomic_itype*, integral);
integral atomic_swap_explicit(volatile atomic_itype*, integral, memory_order);
bool atomic_compare_swap(volatile atomic_itype*, integral*, integral);
bool atomic_compare_swap_explicit(volatile atomic_itype*, integral*, integral, memory_order, memory_order);
void atomic_fence(const volatile atomic_itype*, memory_order);
integral atomic_fetch_add(volatile atomic_itype*, integral);
integral atomic_fetch_add_explicit(volatile atomic_itype*, integral, memory_order);
integral atomic_fetch_sub(volatile atomic_itype*, integral);
integral atomic_fetch_sub_explicit(volatile atomic_itype*, integral, memory_order);
integral atomic_fetch_and(volatile atomic_itype*, integral);
integral atomic_fetch_and_explicit(volatile atomic_itype*, integral, memory_order);
integral atomic_fetch_or(volatile atomic_itype*, integral);
integral atomic_fetch_or_explicit(volatile atomic_itype*, integral, memory_order);
integral atomic_fetch_xor(volatile atomic_itype*, integral);
integral atomic_fetch_xor_explicit(volatile atomic_itype*, integral, memory_order);

// 29.4.2, address types
struct atomic_address;
bool atomic_is_lock_free(const volatile atomic_address*);
void atomic_store(volatile atomic_address*, void*);
void atomic_store_explicit(void*, memory_order);  
void* atomic_load(void*);  
void* atomic_load_explicit(void*, memory_order);  
void* atomic_swap(void*);  
void* atomic_swap_explicit(void*, void*, memory_order);  
bool atomic_compare_swap(void**, void*, memory_order, memory_order);  
void atomic_fence(void*, memory_order);  
void* atomic_fetch_add(void*, ptrdiff_t);  
void* atomic_fetch_add_explicit(void*, ptrdiff_t, memory_order);  
void* atomic_fetch_sub(void*, ptrdiff_t);  
void* atomic_fetch_sub_explicit(void*, ptrdiff_t, memory_order);

// 29.4.3, generic types
template<class T> struct atomic;

namespace std {  
    typedef enum memory_order {  
        memory_order_relaxed, memory_order_acquire, memory_order_release,  
        memory_order_acq_rel, memory_order_seq_cst  
    } memory_order;
}

29.1 Order and Consistency [atomics.order]

namespace std {

    typedef enum memory_order {  
        memory_order_relaxed, memory_order_acquire, memory_order_release,  
        memory_order_acq_rel, memory_order_seq_cst  
    } memory_order;
}

The enumeration memory_order specifies the detailed regular (non-atomic) memory synchronization order as defined in clause 1.7 and may provide for operation ordering. Its enumerated values and their meanings are as follows:

<table>
<thead>
<tr>
<th>Element</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory_order_relaxed</td>
<td>the operation does not order memory</td>
</tr>
<tr>
<td>memory_order_release</td>
<td>the operation performs a release operation on the affected memory location, thus making regular memory writes visible to other threads through the atomic variable to which it is applied</td>
</tr>
<tr>
<td>memory_order_acquire</td>
<td>the operation performs an acquire operation on the affected memory location, thus making regular memory writes in other threads released through the atomic variable to which is applied visible to the current thread</td>
</tr>
<tr>
<td>memory_order_acq_rel</td>
<td>the operation has both acquire and release semantics</td>
</tr>
<tr>
<td>memory_order_seq_cst</td>
<td>the operation has both acquire and release semantics, and, in addition, has sequentially-consistent operation ordering</td>
</tr>
</tbody>
</table>

Draft
The `memory_order_seq_cst` operations that load a value are acquire operations on the affected locations. The `memory_order_seq_cst` operations that store a value are release operations on the affected locations. In addition, in a consistent execution, there must be a single total order $S$ on all `memory_order_seq_cst` operations, consistent with the happens before order and modification orders for all affected locations, such that each `memory_order_seq_cst` operation observes either the last preceding modification according to this order $S$, or the result of an operation that is not `memory_order_seq_cst`. [Note: Although it is not explicitly required that $S$ include locks, it can always be extended to an order that does include lock and unlock operations, since the ordering between those is already included in the happens before ordering. — end note]

An atomic store shall only store a value that has been computed from constants and program input values by a finite sequence of program evaluations, such that each evaluation observes the values of variables as computed by the last prior assignment in the sequence. The ordering of evaluations in this sequence shall be such that:

- if an evaluation $B$ observes a value computed by $A$ in a different thread, then $B$ must not happen before $A$, and
- if an evaluation $A$ is included in the sequence, then all evaluations that assign to the same variable and are sequenced before or happens-before $A$ must be included.

[Note: The second requirement disallows “out-of-thin-air” or “speculative” stores of atomics when relaxed atomics are used. Since unordered operations are involved, evaluations may appear in this sequence out of thread order. For example, with $x$ and $y$ initially zero,

```
// Thread 1:
r1 = y.load(memory_order_relaxed);
x.store(r1, memory_order_relaxed);
// Thread 2:
r2 = x.load(memory_order_relaxed);
y.store(42, memory_order_relaxed);
```

is allowed to produce $r1 = r2 = 42$. The sequence of evaluations justifying this consists of:

```
y.store(42, memory_order_relaxed);
r1 = y.load(memory_order_relaxed);
x.store(r1, memory_order_relaxed);
r2 = x.load(memory_order_relaxed);
```

On the other hand,

```
// Thread 1:
r1 = y.load(memory_order_relaxed);
x.store(r1, memory_order_relaxed);
// Thread 2:
r2 = x.load(memory_order_relaxed);
y.store(r2, memory_order_relaxed);
```

may not produce $r1 = r2 = 42$, since there is no sequence of evaluations that results in the computation of 42. In the absence of “relaxed” operations and read-modify-write operations with weaker than `memory_order_acq_rel` ordering, the second requirement has no impact. — end note]
[Note: The requirements do allow r1 == r2 == 42 in the following example, with x and y initially zero:

// Thread 1:
r1 = x.load(memory_order_relaxed);
if (r1 == 42) y.store(r1, memory_order_relaxed);

// Thread 2:
r2 = y.load(memory_order_relaxed);
if (r2 == 42) x.store(42, memory_order_relaxed);

However, implementations should not allow such behavior. —end note]

Implementations should make atomic stores visible to atomic loads within a reasonable amount of time. Implementations shall not move an atomic operation out of an unbounded loop.

29.2 Lock-free Property

namespace std {
    #define ATOMIC_INTEGRAL_LOCK_FREE unspecified
    #define ATOMIC_ADDRESS_LOCK_FREE unspecified
}

The macros ATOMIC_INTEGRAL_LOCK_FREE and ATOMIC_ADDRESS_LOCK_FREE indicate the general lock-free property of integral and address atomic types. The properties also apply to the corresponding specializations of the atomic template. A value of 0 indicates that the types are never lock-free. A value of 1 indicates that the types are sometimes lock-free. A value of 2 indicates that the types are always lock-free.

The function atomic_is_lock_free (29.4.4) indicates whether the object is lock-free. The result of a lock-free query on one object cannot be inferred from the result of a lock-free query on another object.

[Note: Operations that are lock-free should also be address-free. That is, atomic operations on the same memory location via two different addresses will communicate atomically. The implementation should not depend on any per-process state. This restriction enables communication via memory that is mapped into a process more than once and by memory that is shared between two processes. —end note]

29.3 Flag Type and Operations

namespace std {
    typedef struct atomic_flag {
        bool test_and_set(memory_order = memory_order_seq_cst) volatile;
        void clear(memory_order = memory_order_seq_cst) volatile;
        void fence(memory_order) const volatile;
        atomic_flag() = default;
        atomic_flag(const atomic_flag&) = delete;
        atomic_flag& operator=(const atomic_flag&) = delete;
    } atomic_flag;

    bool atomic_flag_test_and_set(volatile atomic_flag*);
    bool atomic_flag_test_and_set_explicit(volatile atomic_flag*, memory_order);
    void atomic_flag_clear(volatile atomic_flag*);
}

Draft
The atomic_flag type provides the classic test-and-set functionality. It has two states, set and clear.

Operations on an object of type atomic_flag shall be lock-free. [Note: Hence the operations should also be address-free. No other type requires lock-free operations, so the atomic_flag type is the minimum hardware-implemented type needed to conform to this International standard. The remaining types can be emulated with atomic_flag, though with less than ideal properties. —end note]

The atomic_flag type shall have standard layout. It shall have a trivial default constructor, a deleted copy constructor, a deleted copy assignment operator, and a trivial destructor.

The macro ATOMIC_FLAG_INIT shall be defined in such a way that it can be used to initialize an object of type atomic_flag to the clear state. For a static-duration object, that initialization shall be static. A program that uses an object of type atomic_flag without initializing it with the macro ATOMIC_FLAG_INIT is ill-formed. [Example:

atomical_flag guard = ATOMIC_FLAG_INIT;
—end example]

Returns: Atomically, the value of the object immediately before the effects.

Requires: The order argument shall not be memory_order_acquire nor memory_order_acq_rel.

Effects: Atomically sets the value pointed to by object or by this to true. Memory is affected according to the value of order. These operations are read-modify-write operations in the sense of the “synchronizes with” definition (1.10), so both such an operation and the evaluation that produced the input value synchronize with any evaluation that reads the updated value.

Requires: The order argument shall not be memory_order_relaxed.

Effects: Memory is affected according to the value of order. These operations are read-modify-write operations in the sense of the “synchronizes with” definition (1.10), so both such an operation and the evaluation that produced the input value synchronize with any evaluation that reads the updated value.

Draft
[Note: The static-duration variable atomic_global_fence_compatibility of type atomic_flag may be used to emulate global fences. [Example:

    atomic_flag_fence(&atomic_global_fence_compatibility, memory_order_acquire);
    atomic_global_fence_compatibility.fence(memory_order_release);

    — end example] — end note]

29.4 Other Types and Operations

29.4.1 Integral Types

namespace std {

    typedef struct atomic_bool {
        bool is_lock_free() const volatile;
        void store(bool, memory_order = memory_order_seq_cst) volatile;
        bool load(memory_order = memory_order_seq_cst) volatile;
        bool swap(bool, memory_order = memory_order_seq_cst) volatile;
        bool compare_swap(bool&, bool, memory_order, memory_order) volatile;
        bool compare_swap(bool&, bool, memory_order = memory_order_seq_cst) volatile;
        void fence(memory_order) const volatile;
    
    atomic_bool() = default;
    constexpr explicit atomic_bool(bool);
    atomic Bool(const atomicBool&) = delete;
    bool operator=(bool) volatile;

    } atomicBool;

    bool atomic_is_lock_free(const volatile atomicBool*);
    void atomic_store(volatile atomicBool*, bool);
    void atomic_store_explicit(volatile atomicBool*, bool, memory_order);
    bool atomic_load(volatile atomicBool*);
    bool atomic_load_explicit(volatile atomicBool*, memory_order);
    bool atomic_swap(volatile atomicBool*);
    bool atomic_swap_explicit(volatile atomicBool*, bool);
    bool atomic_compare_swap(volatile atomicBool*, bool*, bool);
    bool atomic_compare_swap_explicit(volatile atomicBool*, bool*, bool, memory_order, memory_order);
    void atomic_fence(const volatile atomicBool*, memory_order);

    // For each of the integral types listed below:
    typedef struct atomic_iType {
        bool is_lock_free() const volatile;
        void store(integral, memory_order = memory_order_seq_cst) volatile;
        integral load(memory_order = memory_order_seq_cst) volatile;
        integral swap(integral,
                        memory_order = memory_order_seq_cst) volatile;
        bool compare_swap(integral&, integral,
                           memory_order, memory_order) volatile;

    } atomic_iType;

}
bool compare_swap(integral&, integral,
    memory_order = memory_order_seq_cst) volatile;
void fence(memory_order) const volatile;
integral fetch_add(integral,
    memory_order = memory_order_seq_cst) volatile;
integral fetch_sub(integral,
    memory_order = memory_order_seq_cst) volatile;
integral fetch_and(integral,
    memory_order = memory_order_seq_cst) volatile;
integral fetch_or(integral,
    memory_order = memory_order_seq_cst) volatile;
integral fetch_xor(integral,
    memory_order = memory_order_seq_cst) volatile;

atomic_itype() = default;
constexpr explicit atomic_itype(integral);
atomic_itype(const atomic_itype&) = delete;
atomic_itype& operator=(const atomic_itype&) = delete;
integral operator=(integral) volatile;
integral operator++(int) volatile;
integral operator--(int) volatile;
integral operator++() volatile;
integral operator--() volatile;
integral operator+=(integral) volatile;
integral operator-=(integral) volatile;
integral operator^=(integral) volatile;

bool atomic_is_lock_free(const volatile atomic_itype*);
void atomic_store(volatile atomic_itype*, integral);
void atomic_store_explicit(volatile atomic_itype*, integral,
    memory_order);
integral atomic_load(volatile atomic_itype*);
integral atomic_load_explicit(volatile atomic_itype*, memory_order);
integral atomic_swap(volatile atomic_itype*, integral);
integral atomic_swap_explicit(volatile atomic_itype*, integral,
    memory_order);
bool atomic_compare_swap(volatile atomic_itype*, integral*, integral);
bool atomic_compare_swap_explicit(volatile atomic_itype*, integral*,
    integral*,
    integral, memory_order, memory_order);
void atomic_fence(const volatile atomic_itype*, memory_order);
integral atomic_fetch_add(volatile atomic_itype*, integral);
integral atomic_fetch_add_explicit(volatile atomic_itype*, integral,
    memory_order);
integral atomic_fetch_sub(volatile atomic_itype*, integral);
integral atomic_fetch_sub_explicit(volatile atomic_itype*, integral,
    memory_order);
integral atomic_fetch_and(volatile atomic_itype*, integral);
The name `atomic_itype` and the functions operating on it in the preceding synopsis are placeholders for a set of classes and functions. Throughout the preceding synopsis, `atomic_itype` should be replaced by each of the class names in table 136 and table 137, and `integral` should be replaced by the integral type corresponding to the class name.

<table>
<thead>
<tr>
<th>Class name</th>
<th>Integral type</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_char</td>
<td>char</td>
</tr>
<tr>
<td>atomic_schar</td>
<td>signed char</td>
</tr>
<tr>
<td>atomic_uchar</td>
<td>unsigned char</td>
</tr>
<tr>
<td>atomic_short</td>
<td>short</td>
</tr>
<tr>
<td>atomic_ushort</td>
<td>unsigned short</td>
</tr>
<tr>
<td>atomic_int</td>
<td>int</td>
</tr>
<tr>
<td>atomic_uint</td>
<td>unsigned int</td>
</tr>
<tr>
<td>atomic_long</td>
<td>long</td>
</tr>
<tr>
<td>atomic_ulong</td>
<td>unsigned long</td>
</tr>
<tr>
<td>atomic_llong</td>
<td>long long</td>
</tr>
<tr>
<td>atomic_ullong</td>
<td>unsigned long long</td>
</tr>
<tr>
<td>atomic_char16_t</td>
<td>char16_t</td>
</tr>
<tr>
<td>atomic_char32_t</td>
<td>char32_t</td>
</tr>
<tr>
<td>atomic_wchar_t</td>
<td>wchar_t</td>
</tr>
</tbody>
</table>

The atomic integral types shall have standard layout. They shall each have a trivial default constructor, a constexpr explicit value constructor, a deleted copy constructor, a deleted copy assignment operator, and a trivial destructor. They shall each support aggregate initialization syntax.

The semantics of the operations on these types are defined in 29.4.4.

The `atomic_bool` type provides an atomic boolean.

[Note: The representation of atomic integral types need not have the same size as their corresponding regular types. They should have the same size whenever possible, as it eases porting existing code. — end note]

29.4.2 Address Type

```cpp
namespace std {
    typedef struct atomic_address {
```
### Table 137: Atomics for standard typedef types

<table>
<thead>
<tr>
<th>Class name</th>
<th>Integral type</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_int_least8_t</td>
<td>int_least8_t</td>
</tr>
<tr>
<td>atomic_uint_least8_t</td>
<td>uint_least8_t</td>
</tr>
<tr>
<td>atomic_int_least16_t</td>
<td>int_least16_t</td>
</tr>
<tr>
<td>atomic_uint_least16_t</td>
<td>uint_least16_t</td>
</tr>
<tr>
<td>atomic_int_least32_t</td>
<td>int_least32_t</td>
</tr>
<tr>
<td>atomic_uint_least32_t</td>
<td>uint_least32_t</td>
</tr>
<tr>
<td>atomic_int_least64_t</td>
<td>int_least64_t</td>
</tr>
<tr>
<td>atomic_uint_least64_t</td>
<td>uint_least64_t</td>
</tr>
<tr>
<td>atomic_int_fast8_t</td>
<td>int_fast8_t</td>
</tr>
<tr>
<td>atomic_uint_fast8_t</td>
<td>uint_fast8_t</td>
</tr>
<tr>
<td>atomic_int_fast16_t</td>
<td>int_fast16_t</td>
</tr>
<tr>
<td>atomic_uint_fast16_t</td>
<td>uint_fast16_t</td>
</tr>
<tr>
<td>atomic_int_fast32_t</td>
<td>int_fast32_t</td>
</tr>
<tr>
<td>atomic_uint_fast32_t</td>
<td>uint_fast32_t</td>
</tr>
<tr>
<td>atomic_int_fast64_t</td>
<td>int_fast64_t</td>
</tr>
<tr>
<td>atomic_uint_fast64_t</td>
<td>uint_fast64_t</td>
</tr>
<tr>
<td>atomicintptr_t</td>
<td>intptr_t</td>
</tr>
<tr>
<td>atomicuintptr_t</td>
<td>uintptr_t</td>
</tr>
<tr>
<td>atomic_size_t</td>
<td>size_t</td>
</tr>
<tr>
<td>atomic_ssize_t</td>
<td>ssize_t</td>
</tr>
<tr>
<td>atomicptrdiff_t</td>
<td>ptrdiff_t</td>
</tr>
<tr>
<td>atomicintmax_t</td>
<td>intmax_t</td>
</tr>
<tr>
<td>atomicuintmax_t</td>
<td>uintmax_t</td>
</tr>
</tbody>
</table>

```cpp
bool is_lock_free() const volatile;
void store(void*, memory_order = memory_order_seq_cst) volatile;
void* load(memory_order = memory_order_seq_cst) volatile;
void* swap(void*, memory_order = memory_order_seq_cst) volatile;
bool compare_swap(void*&, void*, memory_order, memory_order) volatile;
bool compare_swap(void*&, void*, memory_order = memory_order_seq_cst) volatile;
void fence(memory_order) const volatile;
void* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst) volatile;
void* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst) volatile;
atomic_address() = default;
constexpr explicit atomic_address(void*);
atomic_address(const atomic_address&) = delete;
atomic_address& operator=(const atomic_address&) = delete;
void* operator=(void*) volatile;
```

Draft
void* operator+=(ptrdiff_t) volatile;
void* operator-=(ptrdiff_t) volatile;
} atomic_address;

bool atomic_is_lock_free(const volatile atomic_address*);
void atomic_store(const volatile atomic_address*, void*);
void* atomic_store_explicit(const volatile atomic_address*, void*, memory_order);
void* atomic_load(volatile atomic_address*);
void* atomic_load_explicit(volatile atomic_address*, memory_order);
void* atomic_swap(volatile atomic_address*);
void* atomic_swap_explicit(volatile atomic_address*, void*, memory_order);
bool atomic_compare_swap(volatile atomic_address*, void**, void*);
bool atomic_compare_swap_explicit(volatile atomic_address*, void**, void*,
                                 memory_order, memory_order);
bool atomic_fence(const volatile atomic_address*, memory_order);
void atomic_fetch_add(volatile atomic_address*, ptrdiff_t);
void* atomic_fetch_add_explicit(volatile atomic_address*, ptrdiff_t,
                                 memory_order);
void* atomic_fetch_sub(volatile atomic_address*, ptrdiff_t);
void* atomic_fetch_sub_explicit(volatile atomic_address*, ptrdiff_t,
                                 memory_order);

1 The type atomic_address shall have standard layout. It shall have a trivial default constructor, a constexpr explicit
value constructor, a deleted copy constructor, a deleted copy assignment operator, and a trivial destructor. It shall support
aggregate initialization syntax.

2 The semantics of the operations on this type are defined in 29.4.4.

3 The atomic_address type provides atomic void* operations. The unit of addition/subtraction shall be one byte.

4 [Note: The representation of the atomic address type need not have the same size as its corresponding regular type. It
should have the same size whenever possible, as it eases effort required to port existing code. — end note ]

29.4.3 Generic Types

namespace std {

    template <class T> struct atomic {
        bool is_lock_free() const volatile;
        void store(T, memory_order = memory_order_seq_cst) volatile;
        T load(memory_order = memory_order_seq_cst) volatile;
        T swap(T, memory_order = memory_order_seq_cst) volatile;
        bool compare_swap(T&, T, memory_order, memory_order) volatile;
        void fence(memory_order) const volatile;

        atomic() = default;
        constexpr explicit atomic(T);
        atomic(const atomic&) = delete;
        atomic& operator=(const atomic&) = delete;
        T operator=(T) volatile;
    }
}
There is a generic class template `atomic<T>`. The type of the template argument `T` shall be trivially copy assignable and bitwise equality comparable. [Note: Type arguments that are not also statically initializable and trivially destructable may be difficult to use. — end note]

Specializations of the `atomic` template shall have a deleted copy constructor, a deleted copy assignment operator, and a constexpr explicit value constructor.

There are full specializations over the integral types on the `atomic` class template. For each integral type `integral` in the second column of table 136 or table 137, the specialization `atomic<integral>` shall be publicly derived from the corresponding atomic integral type in the first column of the table. These specializations shall have trivial default constructors and trivial destructors.

There are pointer partial specializations on the `atomic` class template. These specializations shall be publicly derived from `atomic_address`. The unit of addition/subtraction for these specializations shall be the size of the referenced type. These specializations shall have trivial default constructors and trivial destructors.

### 29.4.4 Operations

There are only a few kinds of operations on atomic types, though there are many instances on those kinds. This section specifies each general kind. The specific instances are defined in 29.4.1, 29.4.2, and 29.4.3.

In the following operation definitions:
— an $A$ refers to one of the atomic types
— a $C$ refers to its corresponding non-atomic type. The *atomic_address* atomic type corresponds to the $void^*$ non-atomic type
— an $M$ refers to type of the other argument for arithmetic operations. For integral atomic types, $M$ is $C$. For atomic address types, $M$ is `std::ptrdiff_t`
— the free functions not ending in *explicit* have the semantics of their corresponding *explicit* with *memory_order* arguments of `memory_order_seq_cst`.

[Note: Many operations are volatile-qualified. The “volatile as device register” semantics have not changed in the standard. This qualification means that volatility is preserved when applying these operations to volatile objects. It does not mean that operations on non-volatile objects become volatile. Thus, volatile qualified operations on non-volatile objects may be merged under some conditions. — end note]

```cpp
bool atomic_is_lock_free(const volatile A *object);
void A::is_lock_free() const volatile;
```

Returns: True if the object’s operations are lock-free, false otherwise.

```cpp
void atomic_store(volatile A *object, C desired);
void atomic_store_explicit(volatile A *object, C desired, memory_order order);
void A::store(C desired, memory_order order = memory_order_seq_cst) volatile;
```

Requires: The order argument shall not be `memory_order_acquire` nor `memory_order_acq_rel`.

Effects: Atomically replaces the value pointed to by `object` or by this with the value of `desired`. Memory is affected according to the value of `order`.

```cpp
C A::operator=(C desired) volatile;
```

Effects: `store(desired)`

Returns: `desired`

```cpp
C atomic_load(volatile A* object);
C atomic_load_explicit(volatile A* object, memory_order);
C A::load(memory_order order = memory_order_seq_cst) volatile;
```

Requires: The order argument shall not be `memory_order_acquire` nor `memory_order_acq_rel`.

Effects: Memory is affected according to the value of `order`.

Returns: Atomically returns the value pointed to by `object` or by this.

```cpp
C atomic_swap(volatile A *object, C desired);
C atomic_swap_explicit(volatile A *object, C desired, memory_order);
C A::swap(C desired, memory_order order = memory_order_seq_cst) volatile;
```

Effects: Atomically replaces the value pointed to by `object` or by this with `desired`. Memory is affected according to the value of `order`. These operations are read-modify-write operations in the sense of the “synchronizes with” definition (1.10), so both such an operation and the evaluation that produced the input value synchronize with any evaluation that reads the updated value.
### 29.4 Other Types and Operations

**Atomic operations library**

#### Returns:
Atomically returns the value pointed to by `object` or by `this` immediately before the effects.

```cpp
bool atomic_compare_swap(volatile A *object, C *expected, C desired);
bool atomic_compare_swap_explicit(volatile A *object, C *expected, C desired,
                                  memory_order success, memory_order failure);
bool A::compare_swap(Ck expected, C desired,
                     memory_order order = memory_order_seq_cst) volatile;
```

**Requires:** The failure argument shall not be `memory_order_release` nor `memory_order_acq_rel`. The failure argument shall be no stronger than the success argument.

**Effects:** Atomically, compares the value pointed to by `object` or by `this` for equality with that in `expected`, and if true, replaces the value pointed to by `object` or by `this` with `desired`, and if false, updates the value in `expected` with the value pointed to by `object` or by `this`. Further, if the comparison is true, memory is affected according to the value of `success`, and if the comparison is false, memory is affected according to the value of `failure`. When only one `memory_order` argument is supplied, the value of `success` is `order`, and the value of `failure` is `order` except that a value of `memory_order_acq_rel` shall be replaced by the value `memory_order_require` and a value of `memory_order_release` shall be replaced by the value `memory_order_relaxed`. These operations are read-modify-write operations in the sense of the “synchronizes with” definition (1.10), so both such an operation and the evaluation that produced the input value synchronize with any evaluation that reads the updated value.

**Returns:** The result of the comparison.

[Note: The effect of the compare-and-swap operations is]

```cpp
if (*object == *expected)
  *object = desired;
else
  *expected = *object;
```

[--- end note]

**Remark:** The compare-and-swap operations may fail spuriously, that is, return false while leaving the value pointed to by `expected` unchanged. [Note: This spurious failure enables implementation of compare-and-swap on a broader class of machines, e.g. load-locked store-conditional machines. — end note] [Example: A consequence of spurious failure is that nearly all uses of compare-and-swap will be in a loop.]

```cpp
expected = current.load();
do desired = function(expected);
while (!current.compare_swap(expected, desired));
```

[--- end example]

```cpp
void atomic_fence(const volatile A *object, memory_order order);
void A::fence(memory_order order) const volatile;
```

**Requires:** The order argument shall not be `memory_order_relaxed`. 

Draft
Effects: Memory is affected according to the value of order. These operations are read-modify-write operations in the sense of the “synchronizes with” definition (1.10), so both such an operation and the evaluation that produced the input value synchronize with any evaluation that reads the updated value.

The following operations perform arithmetic computations. The key, operator, and computation correspondence is:

<table>
<thead>
<tr>
<th>key</th>
<th>op</th>
<th>computation</th>
<th>key</th>
<th>op</th>
<th>computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>+</td>
<td>addition</td>
<td>sub</td>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>or</td>
<td></td>
<td>bitwise inclusive or</td>
<td>xor</td>
<td>^</td>
<td>bitwise exclusive or</td>
</tr>
<tr>
<td>and</td>
<td>&amp;</td>
<td>bitwise and</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ C \text{atomic_fetch_key} (\text{volatile } A \ast \text{object}, M \text{operand}); \]
\[ C \text{atomic_fetch_key_explicit} (\text{volatile } A \ast \text{object}, M \text{operand}, \text{memory_order order}); \]
\[ C A::\text{fetch_key} (M \text{operand}, \text{memory_order order} = \text{memory_order_seq_cst}) \text{ volatile}; \]

Effects: Atomically replaces the value pointed to by object or by this with the result of the computation applied to the value pointed to by object or by this and the given operand. Memory is affected according to the value of order. These operations are read-modify-write operations in the sense of the “synchronizes with” definition (1.10), so both such an operation and the evaluation that produced the input value synchronize with any evaluation that reads the updated value.

Returns: Atomically, the value pointed to by object or by this immediately before the effects.

Remark: For signed integral types, arithmetic is defined to use two’s complement representation. There are no undefined results. For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior.

\[ C A::\text{operator op} = (M \text{operand}) \text{ volatile}; \]

Effects: \text{fetch_key} (operand)

Returns: \text{fetch_key} (operand) op operand

\[ C A::\text{operator}++(\text{int}) \text{ volatile}; \]

Returns: \text{fetch_add}(1)

\[ C A::\text{operator}--(\text{int}) \text{ volatile}; \]

Returns: \text{fetch_sub}(1)

\[ C A::\text{operator}++() \text{ volatile}; \]

Effects: \text{fetch_add}(1)

Returns: \text{fetch_add}(1) + 1

\[ C A::\text{operator}--() \text{ volatile}; \]

Effects: \text{fetch_sub}(1)

Returns: \text{fetch_sub}(1) - 1

Draft
Chapter 30  Thread support library  [thread]

The following subclauses describe components to create and manage threads (1.10), perform mutual exclusion, and communicate conditions between threads, as summarized in Table 139.

Table 139: Thread support library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.2 Threads</td>
<td>&lt;thread&gt;</td>
</tr>
<tr>
<td>30.3 Mutual exclusion</td>
<td>&lt;mutex&gt;</td>
</tr>
<tr>
<td>30.4 Condition variables</td>
<td>condition_variable</td>
</tr>
</tbody>
</table>

30.1 Requirements  [thread.req]

30.1.1 Template parameter names  [thread.req.paramname]

Throughout this clause, the names of template parameters are used to express type requirements.

2 The requirements for Duration parameters are specified in ??.

3 If a parameter is Predicate, operator() applied to the actual template argument shall return a value that is convertible to bool.

30.1.2 Exceptions  [thread.req.exception]

1 Implementations of functions described in this clause are permitted to call operating system or other low-level applications program interfaces (API's). Some functions described in this clause are specified to throw exceptions of type system_error (19.4.5). Such exceptions shall be thrown if such a call results in an error that prevents the library function from satisfying its postconditions or from returning a meaningful value.

2 The error_category (19.4.1.1) of the error_code reported by such an exception’s code() member function is implementation-defined. [Note: The category is typically system_category (19.4.1.1) since these error codes usually originate from the underlying operating system application program interface (API). — end note]

30.1.3 Native handles  [thread.req.native]

1 Several classes described in this clause have members native_handle_type and native_handle. The presence of these members and their semantics is implementation defined. [Note: These members allow implementations to provide
access to implementation details. Their names are specified to facilitate portable compile-time detection. Actual use of these members is inherently non-portable. — end note ]

30.1.4 Timing specifications [thread.req.timing]
1 Several functions described in this clause take an argument to specify a timeout. These timeouts are specified as either a Duration or a Time Point type as specified in Chapter??.
2 The resolution of timing provided by an implementation depends on both operating system and hardware. The finest resolution provided by an implementation is called the native resolution.

30.2 Threads [thread.threads]
1 30.2 describes components that can be used to create and manage threads. [ Note: These threads are intended to map one-to-one with operating system threads. — end note ]

Header <thread> synopsis

namespace std {
    class thread;

    void swap(thread& x, thread& y);
    void swap(thread&& x, thread& y);
    void swap(thread& x, thread&& y);

    namespace this_thread {
        thread::id get_id();
        void yield();
        void sleep(const system_time& abs_time);
        template <class Duration>
            void sleep(const Duration& rel_time);
    }
}

30.2.1 Class thread [thread.thread.class]
1 The class thread provides a mechanism to create a new thread of execution, to join with a thread (i.e. wait for a thread to complete), and to perform other operations that manage and query the state of a thread. A thread object uniquely represents a particular thread of execution. That representation may be transferred to other thread objects in such a way that no two thread objects simultaneously represent the same thread of execution. A thread of execution is detached when no thread object represents that thread. Objects of class thread can be in a state that does not represent a thread of execution. [ Note: A thread object does not represent a thread of execution after default construction, after being moved from, or after a successful call to detach or join. — end note ]

namespace std {
    class thread {
    public:
        // types:
        class id;
        typedef implementation-defined native_handle_type; // See 30.1.3
// construct/copy/destroy:
thread();
template <class F> explicit thread(F f);
template <class F, class ...Args> thread(F&& f, Args&&... args);
~thread();
thread(const thread&) = delete;
thread(thread&);
thread& operator=(const thread&) = delete;
thread& operator=(thread&);

// members:
void swap(thread&);
bool joinable() const;
void join();
void detach();
id get_id() const;
native_handle_type native_handle(); // See 30.1.3

// static members:
static unsigned hardware_concurrency();

30.2.1.1 Class thread::id
namespace std {
    class thread::id {
    public:
        id();
    }

    bool operator==(thread::id x, thread::id y);
    bool operator!=(thread::id x, thread::id y);
    bool operator<(thread::id x, thread::id y);
    bool operator<=(thread::id x, thread::id y);
    bool operator>(thread::id x, thread::id y);
    bool operator>=(thread::id x, thread::id y);

    template<class charT, class traits>
    basic_ostream<charT, traits>&
        operator<< (basic_ostream<charT, traits>&& out, thread::id id);
}

1 An object of type thread::id provides a unique identifier for each thread of execution and a single distinct value for all thread objects that do not represent a thread of execution (30.2.1). Each thread of execution has an associated thread::id object that is not equal to the thread::id object of any other thread of execution and that is not equal to the thread::id object of any std::thread object that does not represent threads of execution.

2 [Note: Relational operators allow thread::id objects to be used as keys in associative containers. — end note]
id();

Effects: Constructs an object of type id.

Throws: Nothing.

Postconditions: The constructed object does not represent a thread of execution.

bool operator==(thread::id x, thread::id y);

Returns: true only if x and y represent the same thread of execution or neither x nor y represents a thread of execution.

Throws: Nothing.

bool operator!=(thread::id x, thread::id y);

Returns: !(x == y)

Throws: Nothing.

bool operator<(thread::id x, thread::id y);

Returns: A value such that operator< is a total ordering as described in 25.3.

Throws: Nothing.

bool operator<=(thread::id x, thread::id y);

Returns: !(y < x)

Throws: Nothing.

bool operator>(thread::id x, thread::id y);

Returns: y < x

Throws: Nothing.

bool operator>=(thread::id x, thread::id y);

Returns: !(x < y)

Throws: Nothing.

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<< (basic_ostream<charT, traits>&& out, thread::id id);

Effects: Inserts an unspecified text representation of id into out. For two objects of type thread::id x and y, if x == y the thread::id objects shall have the same text representation and if x != y the thread::id objects shall have distinct text representations.

Returns: out
### 30.2.1.2 thread constructors

```cpp
thread();
```

**Effects:** Constructs a thread object that does not represent a thread of execution.

**Postcondition:** `get_id() == id()`

**Throws:** Nothing.

```cpp
template <class F> explicit thread(F f);
template <class F, class ...Args> thread(F&& f, Args&&... args);
```

**Requires:** `F` and each `Ti` in `Args` shall be CopyConstructible if an lvalue and otherwise MoveConstructible.

**Effects:** Constructs an object of type `thread` and executes `INVOKE(f, t1, t2, ..., tN)` in a new thread of execution, where `t1, t2, ..., tN` are the values in `args...`. Any return value from `f` is ignored. If `f` terminates with an uncaught exception, `std::terminate()` shall be called.

**Synchronization:** The invocation of the constructor happens before the invocation of `f`.

Every thread shall ensure that each of its uses of static-duration variables happens before their destruction and that all calls to the standard library happen before completion of destruction of static-duration variables and execution of functions registered by `std::atexit` (18.4). [Note: Terminating the thread before a call to `std::exit` or the exit from main is sufficient, but not necessary, to satisfy this requirement. This requirement permits the implementation of thread managers as static-duration objects. —end note]

**Postconditions:** `get_id() != id()`. 

**Throws:** `std::system_error` or `std::bad_alloc` if unable to start the new thread.

```cpp
thread(thread&& x);
```

**Effects:** Constructs an object of type `thread` from `x`, and sets `x` to a default constructed state.

**Postconditions:** `x.get_id() == id()` and `get_id()` returns the value of `x.get_id()` prior to the start of construction.

**Throws:** Nothing.

### 30.2.1.3 thread destructor

```cpp
"thread();
```

**Effects:** If `joinable()` then `detach()`, otherwise no effects. [Note: Destroying a joinable thread can be unsafe if the thread accesses objects or the standard library unless the thread performs explicit synchronization to ensure that it does not access the objects or the standard library past their respective lifetimes. Terminating the process with `_exit` or `quick_exit` removes some of these obligations. —end note]

**Throws:** Nothing.
30.2.1.4 thread assignment

thread& operator=(thread&& x);

Effects: If joinable(), calls detach(). Then assigns the state of x to *this and sets x to a default constructed state.

Postconditions: x.get_id() == id() and get_id() returns the value of x.get_id() prior to the assignment.

Throws: Nothing.

30.2.1.5 thread members

void swap(thread&& x);

Effects: Swaps the state of *this and x.

Throws: Nothing.

bool joinable() const;

Returns: get_id() != id()

Throws: Nothing.

void join();

Precondition: joinable() is true.

Synchronization: The completion of the thread represented by *this happens before (1.10) join() returns. [Note: Operations on *this are not synchronized. — end note]

Postconditions: If join() throws an exception, the value returned by get_id() is unchanged. Otherwise, get_id() == id().

Throws: std::system_error when the postconditions cannot be achieved.

void detach();

Precondition: joinable() is true.

Effects: The thread represented by *this continues execution without the calling thread blocking. When detach() returns, *this no longer represents the possibly continuing thread of execution. When the thread previously represented by *this ends execution, the implementation shall release any owned resources.

Postcondition: get_id() == id().

Throws: std::system_error when the effects or postconditions cannot be achieved.

id get_id() const;

Returns: A default constructed id object if *this does not represent a thread, otherwise this_thread::get_id() for the thread of execution represented by *this.

Throws: Nothing.
30.2.1.6 thread static members

```cpp
unsigned hardware_concurrency();
```

**Returns:** The number of hardware thread contexts. [Note: This value should only be considered to be a hint. — end note ] If this value is not computable or well defined an implementation should return 0.

30.2.1.7 thread specialized algorithms

```cpp
void swap(thread& x, thread& y);
void swap(thread&& x, thread& y);
void swap(thread& x, thread&& y);
```

**Effects:** x.swap(y)

30.2.2 Namespace this_thread

```cpp
namespace std {
    namespace this_thread {
        thread::id get_id();
    
        void yield();
        void sleep(const system_time& abs_time);
        template <class Duration> void sleep(const Duration& rel_time);
    }
}
```

```cpp
thread::id this_thread::get_id();
```

**Returns:** An object of type thread::id that uniquely identifies the current thread of execution. No other thread of execution shall have this id and this thread of execution shall always have this id. The object returned shall not compare equal to a default constructed thread::id.

**Throws:** Nothing.

```cpp
void this_thread::yield();
```

**Effects:** Offers the operating system the opportunity to schedule another thread.

**Synchronization:** None.

**Throws:** Nothing.

```cpp
void this_thread::sleep(const system_time& abs_time);
```

**Effects:** Blocks the calling thread at least until the time specified by abs_time.

**Synchronization:** None.

**Throws:** Nothing.
template <class Duration>
void sleep(const Duration& rel_time);

Effects: Blocks the calling thread for at least the time specified by rel_time.

Synchronization: None.

Throws: Nothing.

30.3 Mutual exclusion [thread.mutex]

This section provides mechanisms for mutual exclusion: mutexes, locks, and call once. These mechanisms ease the production of race-free programs (1.10).

Header <mutex> synopsis

namespace std {
    class mutex;
    class recursive_mutex;
    class timed_mutex;
    class recursive_timed_mutex;

    struct defer_lock_t;
    struct try_to_lock_t;
    struct adopt_lock_t;

    extern const defer_lock_t defer_lock;
    extern const try_to_lock_t try_to_lock;
    extern const adopt_lock_t adopt_lock;

    class lock_error;

    template <class Mutex> class lock_guard;
    template <class Mutex> class unique_lock;

    template <class Mutex>
    void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y);
    template <class Mutex>
    void swap(unique_lock<Mutex>&& x, unique_lock<Mutex>& y);
    template <class Mutex>
    void swap(unique_lock<Mutex>& x, unique_lock<Mutex>&& y);

    template <class L1, class L2, class... L3>
    int try_lock(L1&, L2&, L3&...);
    template <class L1, class L2, class... L3>
    void lock(L1&, L2&, L3&...);

    struct once_flag {
        constexpr once_flag();
        once_flag(const once_flag&) = delete;
        once_flag& operator=(const once_flag&) = delete;
    };

Draft
template<class Callable, class ...Args>
    void call_once(once_flag& flag, Callable func, Args&&... args);
}

30.3.1 Mutex requirements

A mutex object facilitates protection against data races and allows thread-safe synchronization of data between threads. A thread owns a mutex from the time it successfully calls one of the lock functions until it calls unlock. Mutexes may be either recursive or non-recursive, and may grant simultaneous ownership to one or many threads. The mutex types supplied by the Standard Library provide exclusive ownership semantics: only one thread may own the mutex at a time. Both recursive and non-recursive mutexes are supplied.

This section describes requirements on template argument types used to instantiate templates defined in the C++ Standard Library. The template definitions in the C++ Standard Library refer to the named Mutex requirements whose details are set out below. In this description, m is an object of a Mutex type.

A Mutex type shall be DefaultConstructible and Destructible. If initialization of an object of a Mutex type fails, an exception of type std::system_error shall be thrown. A Mutex type shall not be copyable nor movable.

The implementation shall provide lock and unlock operations, as described below. The implementation shall serialize those operations. [Note: Construction and destruction of an object of a Mutex type need not be thread-safe; other synchronization should be used to ensure that Mutex objects are initialized and visible to other threads. —end note]

The expression m.lock() shall be well-formed and have the following semantics:

- **Effects:** Blocks the calling thread until ownership of the mutex can be obtained for the calling thread.
- **Postcondition:** The calling thread owns the mutex.
- **Return type:** void
- **Synchronization:** Prior unlock() operations on the same object shall synchronize with (1.10) this operation.
- **Throws:** std::system_error when the effects or postcondition cannot be achieved.

The expression m.try_lock() shall be well-formed and have the following semantics:

- **Effects:** Attempts to obtain ownership of the mutex for the calling thread without blocking. If ownership is not obtained, there is no effect and try_lock() immediately returns. An implementation may fail to obtain the lock even if it is not held by any other thread. [Note: This spurious failure is normally uncommon, but allows interesting implementations based on a simple compare_swap (29). —end note]
- **Return type:** bool
- **Returns:** true if ownership of the mutex was obtained for the calling thread, otherwise false.
- **Synchronization:** If try_lock() returns true, prior unlock() operations on the same object synchronize with (1.10) this operation. [Note: Since lock() does not synchronize with a failed subsequent try_lock(), the visibility rules are weak enough that little would be known about the state after a failure, even in the absence of spurious failures. —end note]
- **Throws:** Nothing.

Draft
The expression `m.unlock()` shall be well-formed and have the following semantics:

**Precondition:** The calling thread shall own the mutex.

**Effects:** Releases the calling thread’s ownership of the mutex.

**Return type:** `void`

**Synchronization:** This operation synchronizes with (1.10) subsequent lock operations that obtain ownership on the same object.

**Throws:** Nothing.

### 30.3.1 Class `mutex` [thread.mutex.class]

```cpp
namespace std {
    class mutex {
        public:
            mutex();
            ~mutex();

            mutex(const mutex&) = delete;
            mutex& operator=(const mutex&) = delete;

            mutex(mutex&) = delete;
            mutex(mutex&&) = delete;

            void lock();
            bool try_lock();
            void unlock();

            typedef implementation-defined native_handle_type; // See 30.1.3
            native_handle_type native_handle(); // See 30.1.3
    };
}
```

The class `mutex` provides a non-recursive mutex with exclusive ownership semantics. If one thread owns a mutex object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()`) until the owning thread has released ownership with a call to `unlock()`.

The class `mutex` shall satisfy all the Mutex requirements (30.3.1). It shall be a standard-layout class (9).

The behavior of a program is undefined if:

- it destroys a mutex object owned by any thread,
- a thread that owns a mutex object calls `lock()` or `try_lock()` on that object, or
- a thread terminates while owning a mutex object.

### 30.3.1.2 Class `recursive_mutex` [thread.mutex.recursive]

```cpp
namespace std {
    class recursive_mutex {
        public:
    }
```

Draft
The class `recursive_mutex` provides a recursive mutex with exclusive ownership semantics. If one thread owns a `recursive_mutex` object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()`) until the first thread has completely released ownership.

The class `recursive_mutex` shall satisfy all the Mutex requirements (30.3.1). It shall be a standard-layout class (9).

A thread that owns a `recursive_mutex` object may acquire additional levels of ownership by calling `lock()` or `try_lock()` on that object. It is unspecified how many levels of ownership may be acquired by a single thread. If a thread has already acquired the maximum level of ownership for a `recursive_mutex` object, additional calls to `try_lock()` shall fail, and additional calls to `lock()` shall throw an exception of type `std::system_error`. A thread shall call `unlock()` once for each level of ownership acquired by calls to `lock()` and `try_lock()`. Only when all levels of ownership have been released may ownership be acquired by another thread.

The behavior of a program is undefined if:
- it destroys a `recursive_mutex` object owned by any thread or
- a thread terminates while owning a `recursive_mutex` object.

### 30.3.2 TimedMutex requirements

A `TimedMutex` type shall meet the requirements for a `Mutex` type. In addition, it shall meet the requirements set out in this clause 30.3.2, where `rel_time` denotes a value of a type `RelTime` that meets the `Duration` requirements and `abs_time` denotes a value of type `std::system_time`.

The expression `m.timed_lock(rel_time)` shall be well-formed and have the following semantics:

**Precondition:** If the resolution of `RelTime` is finer than the native resolution, the time shall be rounded up to the nearest native resolution.

**Effects:** The function attempts to obtain ownership of the mutex within the time specified by `rel_time`. If the time specified by `rel_time` is less than or equal to 0, the function attempts to obtain ownership without blocking (as if by calling `try_lock()`). The function shall return within the time specified by `rel_time` only if it has obtained ownership of the mutex object. [Note: As with `try_lock()`, there is no guarantee that ownership will...]

Draft
be obtained if the lock is available, but implementations are expected to make a strong effort to do so. —end note

Return type: bool

Returns: true if ownership was obtained, otherwise false.

Synchronization: If timed_lock() returns true, prior unlock() operations on the same object synchronize with (1.10) this operation.

Throws: Nothing.

The expression m.timed_lock(abs_time) shall be well-formed and have the following semantics:

Effects: The function attempts to obtain ownership of the mutex by the time specified by abs_time. If abs_time has already passed, the function attempts to obtain ownership without blocking (as if by calling try_lock()). The function shall return before the time specified by abs_time only if it has obtained ownership of the mutex object. [Note: As with try_lock(), there is no guarantee that ownership will be obtained if the lock is available, but implementations are expected to make a strong effort to do so. —end note]

Return type: bool

Returns: true if ownership was obtained, otherwise false.

Synchronization: If timed_lock() returns true, prior unlock() operations on the same object synchronize with (1.10) this operation.

Throws: Nothing.

### 30.3.2.1 Class timed_mutex

```cpp
namespace std {
    class timed_mutex {
    public:
        timed_mutex();
        ~timed_mutex();
        timed_mutex(const timed_mutex&) = delete;
        timed_mutex& operator=(const timed_mutex&) = delete;

        void lock();
        bool try_lock();
        template <class Duration>
        bool timed_lock(const Duration& rel_time);
        bool timed_lock(const system_time& abs_time);
        void unlock();

        typedef implementation-defined native_handle_type; // See 30.1.3
        native_handle_type native_handle(); // See 30.1.3
    }
};
```

Draft
The class `timed_mutex` provides a non-recursive mutex with exclusive ownership semantics. If one thread owns a `timed_mutex` object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()` and `timed_lock()`) until the owning thread has released ownership with a call to `unlock()` or the call to `timed_lock()` times out (having failed to obtain ownership).

The class `timed_mutex` shall satisfy all of the `TimedMutex` requirements (30.3.2). It shall be a standard-layout class (9).

The behavior of a program is undefined if:

- it destroys a `timed_mutex` object owned by any thread,
- a thread that owns a `timed_mutex` object calls `lock()`, `try_lock()` or either overload of `timed_lock()` on that object, or
- a thread terminates while owning a `timed_mutex` object.

### 30.3.2.2 Class `recursive_timed_mutex`

```cpp
namespace std {
    class recursive_timed_mutex {
    public:
        recursive_timed_mutex();
        ~recursive_timed_mutex();
        recursive_timed_mutex(const recursive_timed_mutex&) = delete;
        recursive_timed_mutex& operator=(const recursive_timed_mutex&) = delete;
        void lock();
        bool try_lock();
        template <class Duration>
            bool timed_lock(const Duration& rel_time);
        bool timed_lock(const system_time& abs_time);
        void unlock();
        typedef implementation-defined native_handle_type; // See 30.1.3
        native_handle_type native_handle(); // See 30.1.3
    };
}
```

The class `recursive_timed_mutex` provides a recursive mutex with exclusive ownership semantics. If one thread owns a `recursive_timed_mutex` object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()` or `timed_lock()`) until the owning thread has completely released ownership or the call to `timed_lock()` times out (having failed to obtain ownership).

The class `recursive_timed_mutex` shall satisfy all of the `TimedMutex` requirements (30.3.2). It shall be a standard-layout class (9).

A thread that owns a `recursive_timed_mutex` object may acquire additional levels of ownership by calling `lock()`, `try_lock()` or `timed_lock()` on that object. It is unspecified how many levels of ownership may be acquired by a single thread. If a thread has already acquired the maximum level of ownership for a `recursive_timed_mutex` object,
additional calls to `try_lock()` or `timed_lock()` shall fail, and additional calls to `lock()` shall throw an exception of type `std::system_error`. A thread shall call `unlock()` once for each level of ownership acquired by calls to `lock()`, `try_lock()` and `timed_lock()`. Only when all levels of ownership have been released may ownership of the object be acquired by another thread.

The behavior of a program is undefined if:

- it destroys a `recursive_timed_mutex` object owned by any thread, or
- a thread terminates while owning a `recursive_timed_mutex` object.

### 30.3.3 Locks

A lock is an object that holds a reference to a mutex and may unlock the mutex during the lock's destruction (such as when leaving block scope). A thread of execution may use a lock to aid in managing mutex ownership in an exception safe manner. A lock is said to own a mutex if it is currently managing the ownership of that mutex for a thread of execution. A lock does not manage the lifetime of the mutex it references. [Note: Locks are intended to ease the burden of unlocking the mutex under both normal and exceptional circumstances. — end note]

Some lock constructors take tag types which describe what should be done with the mutex object during the lock’s construction.

```cpp
namespace std {
    struct defer_lock_t {};// do not acquire ownership of the mutex
    struct try_to_lock_t {};// try to acquire ownership of the mutex
    // without blocking
    struct adopt_lock_t {};// assume the calling thread has already
    // obtained mutex ownership and manage it

    extern const defer_lock_t defer_lock;
    extern const try_to_lock_t try_to_lock;
    extern const adopt_lock_t adopt_lock;
}
```

An exception class `lock_error` is derived from `std::exception` and is used to indicate improper usage of locks such as locking a mutex that the lock already owns or unlocking a mutex that the lock does not own.

```cpp
namespace std {
    class lock_error : public std::exception {
        public:
            virtual const char* what() const throw();
    };
}
```

### 30.3.3.1 Class template lock_guard

```cpp
namespace std {
    template <class Mutex>
    class lock_guard {
        public:
            typedef Mutex mutex_type;
```

Draft
An object of type `lock_guard` controls the ownership of a mutex object within a scope. A `lock_guard` object maintains ownership of a mutex object throughout the `lock_guard` object's lifetime. The behavior of a program is undefined if the mutex referenced by `pm` does not exist for the entire lifetime (3.8) of the `lock_guard` object.

```cpp
explicit lock_guard(mutex_type& m);
lock_guard(mutex_type& m, adopt_lock_t);
~lock_guard();
lock_guard(lock_guard const&) = delete;
lock_guard& operator=(lock_guard const&) = delete;

private:
    // exposition only:
    mutex_type& pm;
};
```

1. An object of type `lock_guard` controls the ownership of a mutex object within a scope. A `lock_guard` object maintains ownership of a mutex object throughout the `lock_guard` object's lifetime. The behavior of a program is undefined if the mutex referenced by `pm` does not exist for the entire lifetime (3.8) of the `lock_guard` object.

```cpp
explicit lock_guard(mutex_type& m);
```

2. **Precondition:** If `mutex_type` is not a recursive mutex, the calling thread does not own the mutex `m`.

3. **Effects:** `m.lock()`

4. **Postcondition:** `&pm == &m`

```cpp
lock_guard(mutex_type& m, adopt_lock_t);
```

5. **Precondition:** The calling thread owns the mutex `m`.

6. **Postcondition:** `&pm == &m`

7. **Throws:** Nothing.

```cpp
~lock_guard();
```

8. **Effects:** `pm.unlock()`

9. **Throws:** Nothing.

### 30.3.3.2 Class template `unique_lock` [thread.lock.unique]

```cpp
namespace std {
    template <class Mutex>
    class unique_lock {
    public:
        typedef Mutex mutex_type;

        // 30.3.3.2.1 construct/copy/destroy
        unique_lock();
        explicit unique_lock(mutex_type& m);
        unique_lock(mutex_type& m, defer_lock_t);
    };
```
30.3 Mutual exclusion

An object of type `unique_lock` controls the ownership of a mutex within a scope. Mutex ownership may be acquired at construction or after construction, and may be transferred, after acquisition, to another `unique_lock` object. Objects of type `unique_lock` are not copyable but are movable. The behavior of a program is undefined if the contained pointer

Draft
pm is not null and the mutex pointed to by pm does not exist for the entire remaining lifetime (3.8) of the unique_lock object.

30.3.3.2.1 unique_lock constructors, destructor, and assignment

unique_lock();

Effects: Constructs an object of type unique_lock.
Postconditions: pm == 0 and owns == false.
Throws: Nothing.

explicit unique_lock(mutex_type& m);

Precondition: If mutex_type is not a recursive mutex the calling thread does not own the mutex.
Effects: Constructs an object of type unique_lock and calls m.lock().
Postconditions: pm == &m and owns == true.

unique_lock(mutex_type& m, defer_lock_t);

Precondition: If mutex_type is not a recursive mutex the calling thread does not own the mutex.
Effects: Constructs an object of type unique_lock.
Postconditions: pm == &m and owns == false.
Throws: Nothing.

unique_lock(mutex_type& m, try_to_lock_t);

Precondition: If mutex_type is not a recursive mutex the calling thread does not own the mutex.
Effects: Constructs an object of type unique_lock and calls m.try_lock().
Postconditions: pm == &m and owns == res, where res is the value returned by the call to m.try_lock().
Throws: Nothing.

unique_lock(mutex_type& m, adopt_lock_t);

Precondition: The calling thread own the mutex.
Effects: Constructs an object of type unique_lock.
Postconditions: pm == &m and owns == true.
Throws: Nothing.

unique_lock(mutex_type& m, const system_time& abs_time);

Precondition: If mutex_type is not a recursive mutex the calling thread does not own the mutex.
Effects: Constructs an object of type unique_lock and calls m.timed_lock(abs_time).
30.3 Mutual exclusion

Postconditions: pm == &m and owns == res, where res is the value returned by the call to m.timed_lock(abs_time).

Throws: Nothing.

template <class Duration>
unique_lock(mutex_type& m, const Duration& rel_time);

Remark: The implementation shall ensure that only Duration types (»)will bind to this constructor.

Precondition: If mutex_type is not a recursive mutex the calling thread does not own the mutex.

Effects: Constructs an object of type unique_lock and calls m.timed_lock(rel_time).

Postconditions: pm == &m and owns == res, where res is the value returned by the call to m.timed_lock(rel_time).

Throws: Nothing.

unique_lock(unique_lock&& u);

Postconditions: pm == u_p.pm and owns == u_p.owns (where u_p is the state of u just prior to this construction), u.pm == 0 and u.owns == false.

Throws: Nothing.

unique_lock& operator=(unique_lock&& u);

Effects: If owns calls pm->unlock().

Postconditions: pm == u_p.pm and owns == u_p.owns (where u_p is the state of u just prior to this construction), u.pm == 0 and u.owns == false.

Throws: Nothing.

[ Note: With a recursive mutex it is possible for both *this and u to own the same mutex before the assignment. In this case, *this will own the mutex after the assignment and u will not. — end note ]

~unique_lock();

Effects: If owns calls pm->unlock().

Throws: Nothing.

30.3.3.2.2 unique_lock locking

void lock();

Effects: pm->lock()

Postcondition: owns == true

Throws: lock_error if on entry owns is true or pm is null.

bool try_lock();
1123  Thread support library  30.3  Mutual exclusion

4  Effects: pm->try_lock()
5  Returns: The value returned by the call to try_lock().
6  Postcondition: owns == res, where res is the value returned by the call to try_lock().
7  Throws: lock_error if on entry owns is true or pm is null.

bool timed_lock(const system_time& abs_time);
8  Effects: pm->timed_lock(abs_time)
9  Returns: The value returned by the call to timed_lock(abs_time).
10  Postcondition: owns == res, where res is the value returned by the call to timed_lock(abs_time).
11  Throws: lock_error if on entry owns is true or pm is null.

12  template <class Duration>
13    bool timed_lock(const Duration& rel_time);
14  Effects: pm->timed_lock(rel_time)
15  Returns: The value returned by the call to timed_lock(rel_time).
16  Postcondition: owns == res, where res is the value returned by the call to timed_lock(rel_time).
17  Throws: lock_error if on entry owns is true or pm is null.

void unlock();
18  Effects: pm->unlock()
19  Postcondition: owns == false
20  Throws: lock_entry if on entry owns is false.

30.3.3.2.3  unique_lock modifiers  [thread.lock.unique.mod]

void swap(unique_lock&& u);
1  Effects: Swaps the data members of *this and u.
2  Throws: Nothing.

mutex_type *release();
3  Returns: The previous value of pm.
4  Postconditions: pm == 0 and owns == false.
5  Throws: Nothing.

template <class Mutex>
void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y);

template <class Mutex>
void swap(unique_lock<Mutex>&& x, unique_lock<Mutex>&& y);
template <class Mutex>
    void swap(unique_lock<Mutex>& x, unique_lock<Mutex>&& y);

    Effects: x.swap(y)

    Throws: Nothing.

30.3.3.2.4 unique_lock observers

    bool owns_lock() const;
    
    Returns: owns
    
    Throws: Nothing.

    explicit operator bool() const;
    
    Returns: owns
    
    Throws: Nothing.

    mutex_type *mutex() const;
    
    Returns: pm
    
    Throws: Nothing.

30.3.4 Generic locking algorithms

    template <class L1, class L2, class... L3> int try_lock(L1&, L2&, L3&...);

    Requires: Each template parameter type shall meet the Mutex requirements, except that a call to try_lock() may throw an exception. [Note: The unique_lock class template meets these requirements when suitably instantiated. — end note]

    Effects: Calls try_lock() for each argument in order beginning with the first until all arguments have been processed or a call to try_lock() fails, either by returning false or by throwing an exception. If a call to try_lock() fails, unlock() shall be called for all prior arguments. 

    Returns: -1 if all calls to try_lock() returned true, otherwise a 0-based index value that indicates the argument for which try_lock() returned false. [Note: On return, either all arguments will be locked or none will be locked. — end note]

    template <class L1, class L2, class... L3> void lock(L1&, L2&, L3&...);

    Requires: Each template parameter type shall meet the Mutex requirements, except that a call to try_lock() may throw an exception. [Note: The unique_lock class template meets these requirements when suitably instantiated. — end note]

    Effects: All arguments are locked via a sequence of calls to lock(), try_lock(), or unlock() on each argument. The sequence of calls shall not result in deadlock, but is otherwise unspecified. [Note: A deadlock avoidance algorithm such as try-and-back-off must be used, but the specific algorithm is not specified to avoid over-
constraining implementations. — end note] If a call to `lock()` or `try_lock()` throws an exception, `unlock()` shall be called for any argument that had been locked by a call to `lock()` or `try_lock()`.

### 30.3.5 Call once

The class `once_flag` is an opaque data structure that `call_once` uses to initialize data without causing a data race or deadlock.

#### 30.3.5.1 Struct `once_flag`

```cpp
constexpr once_flag();
```

**Effects:** Constructs an object of type `once_flag`.

**Synchronization:** The construction of a `once_flag` object is not synchronized.

**Postcondition:** The object’s internal state is set to indicate to an invocation of `call_once` with the object as its initial argument that no function has been called.

#### 30.3.5.2 Function `call_once`

```cpp
template<class Callable, class ...Args>
void call_once(once_flag& flag, Callable func, Args&&... args);
```

**Requires:** The template parameters `Callable` and each `Ti` in `Args` shall be CopyConstructible if an lvalue and otherwise MoveConstructible. `INVOKE(func, w1, w2, ..., wN)` (20.5.2) shall be a valid expression for some values `w1, w2, ..., wN`, where `N == sizeof...(Args)`.

**Effects:** Calls to `call_once` on the same `once_flag` object are serialized. If there has been a prior effective call to `call_once` on the same `once_flag` object, the call to `call_once` returns without invoking `func`. If there has been no prior effective call to `call_once` on the same `once_flag` object, the argument `func` (or a copy thereof) is called as if by invoking `func(args)`. The call to `call_once` is effective if and only if `func(args)` returns without throwing an exception. If an exception is thrown it is propagated to the caller.

**Synchronization:** The completion of an effective call to `call_once` on a `once_flag` object synchronizes with (1.10) all subsequent calls to `call_once` on the same `once_flag` object.

** Throws:** `std::system_error` when the effects cannot be achieved, or any exception thrown by `func`.

**Example:**

```cpp
// global flag, regular function
void init();
std::once_flag flag;

void f() {
    std::call_once(flag, init);
}

// function static flag, function object
struct initializer {
```
void operator()();
};

void g() {
    static std::once_flag flag2;
    std::call_once(flag2, initializer);
}

// object flag, member function
class information {
    std::once_flag verified;
    void verifier();
public:
    void verify() { std::call_once(verified, verifier); }
};

— end example —

### 30.4 Condition variables

Condition variables provide synchronization primitives used to block a thread until notified by some other thread that some condition is met or until a system time is reached. Class `condition_variable` provides a condition variable that can only wait on a `Lock`, allowing maximum efficiency on some platforms. Class `condition_variable_any` provides a general condition variable that can wait on user-supplied lock types.

Condition variables permit concurrent invocation of the `wait`, `timed_wait`, `notify_one` and `notify_all` member functions.

The execution of `notify_one` and `notify_all` shall be atomic. The execution of `wait` and `timed_wait` shall be performed in three atomic parts:

1. the release of the mutex, and entry into the waiting state;
2. the unblocking of the wait; and
3. the reacquisition of the lock.

The implementation shall behave as if `notify_one`, `notify_all`, and each part of the `wait` and `timed_wait` executions are executed in some unspecified total order.

Condition variable construction and destruction need not be synchronized.

#### Header condition_variable synopsis

```cpp
namespace std {
    class condition_variable;
    class condition_variable_any;
}
```

### 30.4.1 Class condition_variable

Draft
namespace std {
    class condition_variable {
        public:

            condition_variable();
            ~condition_variable();

            condition_variable(const condition_variable&) = delete;
            condition_variable& operator=(const condition_variable&) = delete;

            void notify_one();
            void notify_all();
            void wait(unique_lock<mutex>& lock);
            template <class Predicate>
                void wait(unique_lock<mutex>& lock, Predicate pred);
            template <class Duration>
                bool timed_wait(unique_lock<mutex>& lock, const Duration& rel_time);
            bool timed_wait(unique_lock<mutex>& lock, const system_time& abs_time);
            template <class Predicate>
                bool timed_wait(unique_lock<mutex>& lock, const system_time& abs_time, Predicate pred);
            template <class Duration, class Predicate>
                bool timed_wait(unique_lock<mutex>& lock, const Duration& rel_time, Predicate pred);

            typedef implementation-defined native_handle_type; // See 30.1.3
            native_handle_type native_handle(); // See 30.1.3
    }
}

The class `condition_variable` shall be a standard-layout class (9).

`condition_variable();`

**Effects:** Constructs an object of type `condition_variable`.

`~condition_variable();`

**Precondition:** There shall be no thread blocked on `*this`. [Note: That is, all threads shall have been notified; they may subsequently block on the lock specified in the wait. Beware that destroying a `condition_variable` object while the corresponding predicate is false is likely to lead to undefined behavior. — end note]

`void notify_one();`

**Effects:** Destroys the object.

**Throws:** Nothing.

`void notify_all();`

**Effects:** If any threads are blocked waiting for `*this`, unblocks one of those threads.

`void notify_all();`

**Effects:** Unblocks all threads that are blocked waiting for `*this`. Draft
void wait(unique_lock<mutex>& lock);

**Precondition:** lock is locked by the calling thread, and either

— no other thread is waiting on this condition_variable object or
— lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting threads (via wait or timed_wait).

**Effects:**

— Atomically calls lock.unlock() and blocks on *this.
— When unblocked, calls lock.lock() (possibly blocking on the lock) and returns.
— The function will unblock when signaled by a call to notify_one(), a call to notify_all(), or spuriously.
— If the function exits via an exception, lock.unlock() shall be called prior to exiting the function scope.

**Postcondition:** lock is locked by the calling thread.

**Throws:** std::system_error when the effects or postcondition cannot be achieved.

template <class Predicate>
void wait(unique_lock<mutex>& lock, Predicate pred);

**Effects:**

while (!pred())
    wait(lock);

bool timed_wait(unique_lock<mutex>& lock, const system_time& abs_time);

**Precondition:** lock is locked by the calling thread, and either

— no other thread is waiting on this condition_variable object or
— lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting threads (via wait or timed_wait).

**Effects:**

— Atomically calls lock.unlock() and blocks on *this.
— When unblocked, calls lock.lock() (possibly blocking on the lock) and returns.
— The function will unblock when signaled by a call to notify_one(), a call to notify_all(), by the current time exceeding abs_time, or spuriously.
— If the function exits via an exception, lock.unlock() shall be called prior to exiting the function scope.

**Postcondition:** lock is locked by the calling thread.

**Returns:** false if the call is returning because the time specified by abs_time was reached, otherwise true.

**Throws:** std::system_error when the returned value, effects, or postcondition cannot be achieved.
template <class Duration>
bool timed_wait(unique_lock<mutex>& lock, const Duration& rel_time);
    timed_wait(lock, std::get_system_time() + rel_time)

Returns: false if the call is returning because the time duration specified by rel_time has elapsed, otherwise true.

template <class Predicate>
bool timed_wait(unique_lock<mutex>& lock, const system_time& abs_time, Predicate pred);

Effects:
    while (!pred())
        if (!timed_wait(lock, abs_time))
            return pred();
    return true;

Returns: pred()

[Note: The returned value indicates whether the predicate evaluates to true regardless of whether the timeout was triggered. — end note]

template <class Duration, class Predicate>
bool timed_wait(unique_lock<mutex>& lock, const Duration& rel_time, Predicate pred);

Effects:
    timed_wait(lock, std::get_system_time() + rel_time, std::move(pred))

[Note: There is no blocking if pred() is initially true, even if the timeout has already expired. — end note]

Returns: pred()

[Note: The returned value indicates whether the predicate evaluates to true regardless of whether the timeout was triggered. — end note]

30.4.2 Class condition_variable_any

A Lock type shall meet the requirements for a Mutex type, except that try_lock is not required. [Note: All of the standard mutex types meet this requirement. — end note]

namespace std {
    class condition_variable_any {
public:
        condition_variable_any();
    ~condition_variable_any();

        condition_variable_any(const condition_variable_any&) = delete;
        condition_variable_any& operator=(const condition_variable_any&) = delete;
    }
30.4 Condition variables

void notify_one();
void notify_all();
template <class Lock>
  void wait(Lock& lock);
template <class Lock, class Predicate>
  void wait(Lock& lock, Predicate pred);
template <class Lock>
  bool timed_wait(Lock& lock, const system_time& abs_time);
template <class Lock, class Duration>
  bool timed_wait(Lock& lock, const Duration& rel_time);
template <class Lock, class Predicate>
  bool timed_wait(Lock& lock, const system_time& abs_time, Predicate pred);
template <class Lock, class Duration, class Predicate>
  bool timed_wait(Lock& lock, const Duration& rel_time, Predicate pred);

typedef implementation-defined native_handle_type; // See 30.1.3
native_handle_type native_handle(); // See 30.1.3

condition_variable_any();

Effects: Constructs an object of type condition_variable_any.
~condition_variable_any();

Precondition: There shall be no thread blocked on *this. [Note: That is, all threads shall have been notified; they
may subsequently block on the lock specified in the wait. Beware that destroying a condition_variable_any
object while the corresponding predicate is false is likely to lead to undefined behavior. — end note]

Effects: Destroys the object.

Throws: Nothing.

void notify_one();

Effects: If any threads are blocked waiting for *this, unblocks one of those theads.

void notify_all();

Effects: Unblocks all threads that are blocked waiting for *this.

template <class Lock>
  void wait(Lock& lock);

Effects:
  — Atomically calls lock.unlock() and blocks on *this.
  — When unblocked, calls lock.lock() (possibly blocking on the lock) and returns.
  — The function will unblock when signaled by a call to notify_one(), a call to notify_all(), or spuriously.
  — If the function exits via an exception, lock.unlock() shall be called prior to exiting the function scope.

Draft
Postcondition: lock is locked by the calling thread.

Throws: std::system_error when the effects or postcondition cannot be achieved.

template <class Lock, class Predicate>
void wait(Lock& lock, Predicate pred);

Effects:
while (!pred())
  wait(lock);

template <class Lock>
bool timed_wait(Lock& lock, const system_time& abs_time);

Effects:
— Atomically calls lock.unlock() and blocks on *this.
— When unblocked, calls lock.lock() (possibly blocking on the lock) and returns.
— The function will unblock when signaled by a call to notify_one(), a call to notify_all(), by the current time exceeding abs_time, or spuriously.
— If the function exits via an exception, lock.unlock() shall be called prior to exiting the function scope.

Postcondition: lock is locked by the calling thread.

Returns: false if the call is returning because the time specified by abs_time was reached, otherwise true.

Throws: std::system_error when the returned value, effects, or postcondition cannot be achieved.

template <class Lock, class Duration>
bool timed_wait(Lock& lock, const Duration& rel_time);

Effects:
timed_wait(lock, std::get_system_time() + rel_time)

Returns: false if the call is returning because the time duration specified by rel_time has elapsed, otherwise true.

template <class Lock, class Predicate>
bool timed_wait(Lock& lock, const system_time& abs_time, Predicate pred);

Effects:
while (!pred())
  if (!timed_wait(lock, abs_time))
    return pred();
  return true;

Returns: pred()

[Note: The returned value indicates whether the predicate evaluates to true regardless of whether the timeout was triggered. —end note]
template <class Lock, class Duration, class Predicate>
bool timed_wait(Lock& lock, const Duration& rel_time, Predicate pred);

Effects:

timed_wait(lock, std::get_system_time() + rel_time, std::move(pred))

[Note: There is no blocking if pred() is initially true, even if the timeout has already expired. — end note]

Returns: pred()

[Note: The returned value indicates whether the predicate evaluates to true regardless of whether the timeout was triggered. — end note]
Appendix A
(informative)
Grammar summary

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 typedef (7.1.3), namespace (7.3.1), class (clause 9), enumeration (7.2), and template (clause 14) declarations.

typedef-name:
   identifier
namespace-name:
   original-namespace-name
      namespace-alias
original-namespace-name:
   identifier
namespace-alias:
   identifier
class-name:
   identifier
template-id
enum-name:
   identifier
template-name:
   identifier
Note that a typedef-name naming a class is also a class-name (9.1).

A.2 Lexical conventions

hex-quad:
   hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit
universal-character-name:
\u hex-quad
\U hex-quad hex-quad

preprocessing-token:
header-name
identifier
pp-number
character-literal
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 e sign
pp-number E sign
pp-number .

identifier:
identifier-nondigit
identifier identifier-nondigit
identifier digit

identifier-nondigit:
nondigit
universal-character-name
other implementation-defined characters
nondigit: one of
  a b c d e f g h i j k l m
  n o p q r s t u v w x y z
  A B C D E F G H I J K L M
  N O P Q R S T U V W X Y Z _

digit: one of
  0 1 2 3 4 5 6 7 8 9

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

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

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

Draft
A.2 Lexical conventions

integer-suffix:
  unsigned-suffix long-suffix\text{opt}
  unsigned-suffix long-long-suffix\text{opt}
  long-suffix unsigned-suffix\text{opt}
  long-long-suffix unsigned-suffix\text{opt}

unsigned-suffix: one of
  \textit{u} \ U

long-suffix: one of
  \textit{l} \ L

long-long-suffix: one of
  \textit{ll} \ LL

character-literal: one of
  \' c-char-sequence \\
  u\textquotesingle c-char-sequence \\
  U\textquotesingle c-char-sequence \\
  L\textquotesingle 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 \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
  \textbackslash \textbackslash \textbackslash a \textbackslash b \textbackslash f \textbackslash n \textbackslash r \textbackslash t \textbackslash v

ioctal-escape-sequence:
  \textbackslash octal-digit
  \textbackslash octal-digit octal-digit
  \textbackslash octal-digit octal-digit octal-digit

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

floating-literal:
  fractional-constant exponent-part\text{opt} floating-suffix\text{opt}
  digit-sequence exponent-part floating-suffix\text{opt}

fractional-constant:
  digit-sequence\text{opt} . digit-sequence
  digit-sequence .
A.2 Lexical conventions

exponent-part:
  • sign\textsubscript{opt} digit-sequence
  • E sign\textsubscript{opt} digit-sequence

sign: one of
  • + -

digit-sequence:
  • digit
digit-sequence digit

floating-suffix: one of
  • \texttt{f F L}

string-literal:
  • " s-char-sequence\textsubscript{opt} "
  • u8" s-char-sequence\textsubscript{opt} "
  • u" s-char-sequence\textsubscript{opt} "
  • U" s-char-sequence\textsubscript{opt} "
  • L" s-char-sequence\textsubscript{opt} "
  • R raw-string
  • u8R raw-string
  • uR raw-string
  • UR raw-string
  • LR raw-string

s-char-sequence:
  • s-char
  • s-char-sequence s-char

s-char:
  • any member of the source character set except
    the double-quote "," , backslash \, or new-line character
    escape-sequence
    universal-character-name

raw-string:
  • " d-char-sequence\textsubscript{opt} [ r-char-sequence\textsubscript{opt} ] d-char-sequence\textsubscript{opt} "

r-char-sequence:
  • r-char
  • r-char-sequence r-char

r-char:
  • any member of the source character set, except
    (1), a backslash \ followed by a u or U, or
    (2), a right square bracket ] followed by the initial d-char-sequence
    (which may be empty) followed by a double quote ".
    universal-character-name

d-char-sequence:
  • d-char
d-char-sequence d-char
A.3 Basic concepts

**d-char:**
any member of the basic source character set except:
- space, the left square bracket `[`, the right square bracket `]`,
- and the control characters representing horizontal tab,
  vertical tab, form feed, and newline.

**boolean-literal:**
- `false`
- `true`

**pointer-literal:**
- `nullptr`

---

A.4 Expressions

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

**lambda-expression:**
- `lambda-introducer lambda-parameter-declaration_opt compound-statement`

**lambda-introducer:**
- `[ lambda-capture_opt ]`

---

Draft
lambda-capture:
  capture-default
  capture-list
  capture-default, capture-list

capture-default:
  &
  =
capture-list:
  capture
  capture-list, capture
capture:
  identifier
  & identifier
  this

lambda-parameter-declaration:
  (lambda-parameter-declaration-listopt) exception-specificationopt lambda-return-type-claustopt

lambda-parameter-declaration-list:
  lambda-parameter
  lambda-parameter, lambda-parameter-declaration-list

lambda-parameter:
  decl-specifier-seq declarator

lambda-return-type-clause:
  -> type-id

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 )
  const_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

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
    * & + - ! ~

new-expression:
    ::= opt new new-placement_opt new-type-id new-initializer_opt
    ::= opt new new-placement_opt (type-id) new-initializer_opt

new-placement:
    (expression-list)

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

new-declarator:
    ptr-operator new-declarator_opt
    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

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
A.5 Statements

```plaintext
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

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
```

Draft
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

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-seq, opt init-declarator-list, opt;

static_assert-declaration:
   static_assert ( constant-expression, string-literal );

decl-specifier:
   storage-class-specifier
   type-specifier
   function-specifier
   friend
   typedef
   constexpr
   alignment-specifier
A.6 Declarations

**Grammar summary**

### decl-specifier-seq:
- decl-specifier-seq\opt 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-qualifier

### type-specifier-seq:
- type-specifier type-specifier-seq\opt

### simple-type-specifier:
- ::\opt nested-name-specifier\opt 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
- decltype ( expression )

### type-name:
- class-name
- enum-name
- typedef-name

Draft
elaborated-type-specifier:
  class-key :: opt nested-name-specifier opt identifier
  class-key :: opt nested-name-specifier opt template opt simple-template-id
  enum-key :: opt nested-name-specifier opt identifier

alignment-specifier:
  alignas ( constant-expression )
  alignas ( type-id )

enum-name:
  identifier
enum-specifier:
  enum-key identifier opt enum-base opt { enumerator-list opt }
  enum-key identifier opt enum-base opt { 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
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:
  inline opt namespace identifier { namespace-body }

extension-namespace-definition:
  inline opt namespace original-namespace-name { namespace-body }

unnamed-namespace-definition:
  inline opt namespace { namespace-body }

namespace-body:
  declaration-seq opt

Draft
A.7 Declarators

namespace-alias:
  identifier

namespace-alias-definition:
  namespace identifier = qualified-namespace-specifier ;

qualified-namespace-specifier:
  ::opt  nested-name-specifier opt namespace-name

using-declaration:
  using typenameopt  ::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-seqopt }
  extern string-literal declaration

A.7 Declarators [gram.decl]

init-declarator-list:
  init-declarator
  init-declarator-list , init-declarator

init-declarator:
  declarator initializeropt

declarator:
  direct-declarator

ptr-operator declarator

direct-declarator:
  declarator-id
  direct-declarator ( parameter-declaration-clause )
    cv-qualifier-seqopt  ref-qualifieropt  exception-specificationopt
  direct-declarator ( parameter-declaration-clause )
    cv-qualifier-seqopt  ref-qualifieropt  exception-specificationopt -> type-id
  direct-declarator [ constant-expressionopt ]
  ( declarator )

ptr-operator:
  * cv-qualifier-seqopt
  &
  &&
    ::opt  nested-name-specifier  * cv-qualifier-seqopt

cv-qualifier-seq:
  cv-qualifier cv-qualifier-seqopt

cv-qualifier:
  const
  volatile

Draft
ref-qualifier:
   &
   &&

declarator-id:
   ...opt id-expression
   ::opt nested-name-specifier opt class-name

type-id:
   type-specifier-seq abstract-declarator

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 ref-qualifier opt exception-specification opt
   direct-abstract-declarator opt ( parameter-declaration-clause )
   cv-qualifier-seq opt ref-qualifier opt exception-specification opt -> type-id
   direct-abstract-declarator opt [ constant-expression opt ]
   ( abstract-declarator )

parameter-declaration-clause:
   parameter-declaration-list 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
   decl-specifier-seq abstract-declarator opt = assignment-expression

function-definition:
   decl-specifier-seq declarator function-body
   decl-specifier-seq declarator = default ;
   decl-specifier-seq declarator = delete ;

function-body:
   ctor-initializer opt compound-statement
   function-try-block

initializer:
   = initializer-clause
   ( expression-list )
A.8 Classes

**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 , base-clause opt`

`class-key nested-name-specifier identifier base-clause opt`

`class-key nested-name-specifier , 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 , 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`

**member-declarator:**

`declarator pure-specifier opt`

`declarator constant-initializer opt`

`identifier , constant-initializer opt`

`constant-initializer : constant-expression`

`= 0`

**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

operator: one of

new delete new[] delete[]

+ - * / % ~ & |

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

<= >= && || ++ -- , ->* ->

\( () \ [ ] \)

A.12 Templates

template-declaration:

exportopt 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 identifieropt
class identifieropt = type-id
typename . . . opt identifieropt
typename identifieropt = type-id
template < template-parameter-list > class . . . opt identifieropt
template < template-parameter-list > class identifieropt = id-expression

simple-template-id:

template-name < template-argument-listopt >

template-id:

simple-template-id

operator-function-id < template-argument-listopt >

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 templateopt simple-template-id

explicit-instantiation:

externopt template declaration

extPLICIT-specialization:

template < > declaration

Draft
A.13 Exception handling

try-block:
   try compound-statement handler-seq

function-try-block:
   try ctor-initializer_opt compound-statement handler-seq

handler-seq:
   handler handler-seq_opt

handler:
   catch ( exception-declaration ) compound-statement

exception-declaration:
   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

Draft
elif-group:
    # elif constant-expression new-line group_{opt}

else-group:
    # else new-line group_{opt}

endif-line:
    # endif new-line

ccontrol-line:
    # include pp-tokens new-line
    # define identifier replacement-list new-line
    # define identifier lparen identifier-list_{opt} ) replacement-list new-line
    # define identifier lparen . . . ) replacement-list new-line
    # define identifier lparen identifier-list, . . . ) replacement-list new-line
    # undef identifier new-line
    # line pp-tokens new-line
    # error pp-tokens_{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
Appendix B  
(informative)  
Implementation quantities  

Because computers are finite, C++ implementations are inevitably limited in the size of the programs they can successfully process. Every implementation shall document those limitations where known. This documentation may cite fixed limits where they exist, say how to compute variable limits as a function of available resources, or say that fixed limits do not exist or are unknown.

The limits may constrain quantities that include those described below or others. The bracketed number following each quantity is recommended as the minimum for that quantity. However, these quantities are only guidelines and do not determine compliance.

- 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 [1024].
- Number of characters in an external identifier [1024].
- External identifiers in one translation unit [65 536].
- Identifiers with block scope declared in one block [1024].
- 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 };
```

Draft
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.

4.10
Change: Converting void* to a pointer-to-object type requires casting

```
char a[10];
void *b=a;
void foo() {
```
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 “obsolete” in ISO C.

Difficulty of converting: Syntactic transformation. Facilities for producing explicit function declarations are fairly widespread commercially.

How widely used: Common.

5.3.3, 5.4

Change: Types must be declared in declarations, not in expressions. In C, a `sizeof` expression or cast expression may create a new type. For example,

```c
p = (void*)(struct x {int i;}) *0;
```

declares a new type, `struct x`.

Rationale: This prohibition helps to clarify the location of declarations in the source code.

Effect on original feature: Deletion of a semantically well-defined feature.

Difficulty of converting: Syntactic transformation.

How widely used: Seldom.

5.16, 5.17, 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 lvalues.

Effect on original feature: Change to semantics of well-defined feature. Some C expressions that implicitly rely on lvalue-to-rvalue conversions will yield different results. For example,

```c
char arr[100];
sizeof(0, arr)
```
yields 100 in C++ and `sizeof(char*)` in C.

Difficulty of converting: Programs must add explicit casts to the appropriate rvalue.

How widely used: Rare.

C.1.4 Clause 6: statements

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

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;
};
```

Draft
Rationale: Storage class specifiers don’t have any meaning when associated with a type. In C++, class members can be declared with the static storage class specifier. Allowing storage class specifiers on type declarations could render the code confusing for users.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Syntactic transformation.

How widely used: Seldom.

7.1.3

Change: A C++ typedef name must be different from any class type name declared in the same scope (except if the typedef is a synonym of the class name with the same name). In C, a typedef name and a struct tag name declared in the same scope can have the same name (because they have different name spaces)

Example:

typedef struct name1 { /*...*/ } name1; // valid C and C++
struct name { /*...*/ }; // valid C, invalid C++
typedef int name; // valid C, invalid C++

Rationale: For ease of use, C++ doesn’t require that a type name be prefixed with the keywords class, struct or union when used in object declarations or type casts.

Example:

class name { /*...*/ }; // i has type class name

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++:

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.

7.1.6.4
Change: The keyword auto cannot be used as a storage class specifier.

```cpp
void f() {
    auto int x;       // valid C, invalid C++
}
```

Rationale: Allowing the use of auto to deduce the type of a variable from its initializer results in undesired interpretations of auto as a storage class specifier in certain contexts.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Syntactic transformation.

How widely used: Rare.

7.2
Change: C++ objects of enumeration type can only be assigned values of the same enumeration type. In C, objects of enumeration type can be assigned values of any integral type

Example:

```cpp
enum color { red, blue, green };
enum color c = 1;       // valid C, invalid C++
```

Rationale: The type-safe nature of C++.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Syntactic transformation. (The type error produced by the assignment can be automatically corrected by applying an explicit cast.)

How widely used: Common.

7.2
Change: In C++, the type of an enumerator is its enumeration. In C, the type of an enumerator is int.

Example:

```cpp
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 a parameter list would be in the scope of the function, the only legal calls in C++ would be from within the function itself.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. The type definitions must be moved to file scope, or in header files.
How widely used: Seldom. This style of type 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:
```
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

**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:
```
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:
```
struct X {
    struct Y { /* */ y;
```
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 Y and struct X are at the same scope
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;
```
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 [diff.cpp]

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 [diff.library]

1 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, 21.5).
2 The C++ Standard library provides 56 standard macros from the C library, as shown in Table 140.
### Table 140: Standard Macros

<table>
<thead>
<tr>
<th>Macro</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert</td>
<td>HUGE_VAL</td>
</tr>
<tr>
<td>BUFSIZ</td>
<td>L_tmpnam</td>
</tr>
<tr>
<td>CLOCKS_PER_SEC</td>
<td>LC_ALL</td>
</tr>
<tr>
<td>EDOM</td>
<td>LC_COLLATE</td>
</tr>
<tr>
<td>EILSEQ</td>
<td>LCCTYPE</td>
</tr>
<tr>
<td>EOF</td>
<td>LC_MONETARY</td>
</tr>
<tr>
<td>ERANGE</td>
<td>LC_NUMERIC</td>
</tr>
<tr>
<td>errno</td>
<td>LC_TIME</td>
</tr>
<tr>
<td>EXIT_FAILURE</td>
<td>MB_CUR_MAX</td>
</tr>
<tr>
<td>EXIT_SUCCESS</td>
<td>NULL</td>
</tr>
<tr>
<td>FILENAME_MAX</td>
<td>NULL</td>
</tr>
<tr>
<td>FOPEN_MAX</td>
<td>NULL</td>
</tr>
<tr>
<td>SIG_ERR</td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>SIG_IEEE</td>
<td>&lt;cstring&gt;</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>&lt;ctime&gt;</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>&lt;cwchar&gt;</td>
</tr>
<tr>
<td>SIG_IGN</td>
<td>&lt;cwchar&gt;</td>
</tr>
<tr>
<td>SIGINT</td>
<td>&lt;cwctype&gt;</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>&lt;cwctype&gt;</td>
</tr>
<tr>
<td>WCHARTX</td>
<td>&lt;cwctype&gt;</td>
</tr>
<tr>
<td>_IOLBF</td>
<td>set jmp</td>
</tr>
<tr>
<td>_IOPBF</td>
<td>set jmp</td>
</tr>
</tbody>
</table>

### Table 141: Standard Values

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR_BIT</td>
<td>FLT_DIG</td>
</tr>
<tr>
<td>CHAR_MAX</td>
<td>FLT_EPSILON</td>
</tr>
<tr>
<td>CHAR_MIN</td>
<td>FLT_MANT_DIG</td>
</tr>
<tr>
<td>DBL_DIG</td>
<td>FLT_MAX</td>
</tr>
<tr>
<td>DBL_EPSILON</td>
<td>FLT_MAX</td>
</tr>
<tr>
<td>DBL_MIN</td>
<td>FLT_MIN</td>
</tr>
<tr>
<td>DBL_MAX</td>
<td>FLT_MAX</td>
</tr>
<tr>
<td>DBL_MIN_10_EXP</td>
<td>FLT_MIN</td>
</tr>
<tr>
<td>DBL_MAX_10_EXP</td>
<td>FLT_MAX</td>
</tr>
<tr>
<td>DBL_MIN_EXP</td>
<td>FLT_MIN</td>
</tr>
<tr>
<td>DBL_MAX_EXP</td>
<td>FLT_MAX</td>
</tr>
<tr>
<td>DBL_MIN_10_EXP</td>
<td>FLT_MIN</td>
</tr>
<tr>
<td>DBL_MAX_10_EXP</td>
<td>FLT_MAX</td>
</tr>
<tr>
<td>USRT_MAX</td>
<td>SIZE_MAX</td>
</tr>
<tr>
<td>USRT_MIN</td>
<td>SIZE_MIN</td>
</tr>
<tr>
<td>USRT_MAX</td>
<td>SIZE_MAX</td>
</tr>
<tr>
<td>USRT_MIN</td>
<td>SIZE_MIN</td>
</tr>
</tbody>
</table>

### Table 142: Standard Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock_t</td>
<td>ldiv_t</td>
</tr>
<tr>
<td>div_t</td>
<td>mbstate_t</td>
</tr>
<tr>
<td>FILE</td>
<td>ptdiff_t</td>
</tr>
<tr>
<td>fpos_t</td>
<td>sig_atomic_t</td>
</tr>
<tr>
<td>jmp_buf</td>
<td>size_t</td>
</tr>
<tr>
<td>time_t</td>
<td>wint_t</td>
</tr>
<tr>
<td>va_list</td>
<td>va_list</td>
</tr>
<tr>
<td>wctrans_t</td>
<td>wctrans_t</td>
</tr>
<tr>
<td>wchar_t</td>
<td>wchar_t</td>
</tr>
</tbody>
</table>

### Table 143: Standard Structs

<table>
<thead>
<tr>
<th>Structure</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>_lconv</td>
<td>tm</td>
</tr>
<tr>
<td>abort</td>
<td>fmod</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>abs</td>
<td>fopen</td>
</tr>
<tr>
<td>acos</td>
<td>fprintf</td>
</tr>
<tr>
<td>asctime</td>
<td>fputc</td>
</tr>
<tr>
<td>asin</td>
<td>fputs</td>
</tr>
<tr>
<td>atan</td>
<td>fprintf</td>
</tr>
<tr>
<td>atan2</td>
<td>fprintf</td>
</tr>
<tr>
<td>atexit</td>
<td>fread</td>
</tr>
<tr>
<td>atof</td>
<td>free</td>
</tr>
<tr>
<td>atoi</td>
<td>freopen</td>
</tr>
<tr>
<td>atol</td>
<td>frexp</td>
</tr>
<tr>
<td>bsearch</td>
<td>fscanf</td>
</tr>
<tr>
<td>bтовс</td>
<td>fseek</td>
</tr>
<tr>
<td>calloc</td>
<td>fsetpos</td>
</tr>
<tr>
<td>ceil</td>
<td>ftell</td>
</tr>
<tr>
<td>clearerr</td>
<td>fwrite</td>
</tr>
<tr>
<td>clock</td>
<td>fprintf</td>
</tr>
<tr>
<td>cos</td>
<td>fwrite</td>
</tr>
<tr>
<td>csh</td>
<td>fscanf</td>
</tr>
<tr>
<td>ctime</td>
<td>getc</td>
</tr>
<tr>
<td>difftime</td>
<td>getchar</td>
</tr>
<tr>
<td>div</td>
<td>getenv</td>
</tr>
<tr>
<td>exit</td>
<td>gets</td>
</tr>
<tr>
<td>exp</td>
<td>getwc</td>
</tr>
<tr>
<td>fabs</td>
<td>getwchar</td>
</tr>
<tr>
<td>fclose</td>
<td>g姿态</td>
</tr>
<tr>
<td>feof</td>
<td>isalnum</td>
</tr>
<tr>
<td>ferror</td>
<td>isalpha</td>
</tr>
<tr>
<td>fflush</td>
<td>iscntrl</td>
</tr>
<tr>
<td>fgetc</td>
<td>isdigit</td>
</tr>
<tr>
<td>fgetpos</td>
<td>isgraph</td>
</tr>
<tr>
<td>fgets</td>
<td>islower</td>
</tr>
<tr>
<td>fgetwc</td>
<td>isprint</td>
</tr>
<tr>
<td>fgetws</td>
<td>ispunct</td>
</tr>
<tr>
<td>floor</td>
<td>isspace</td>
</tr>
</tbody>
</table>

Table 144: Standard Functions
3 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).
4 The C++ Standard library provides 57 standard values from the C library, as shown in Table 141.
5 The C++ Standard library provides 20 standard types from the C library, as shown in Table 142.
6 The C++ Standard library provides 2 standard structs from the C library, as shown in Table 143.
7 The C++ Standard library provides 209 standard functions from the C library, as shown in Table 144.

C.2.1 Modifications to headers [diff.mods.to.headers]
1 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 [diff.mods.to.definitions]
C.2.2.1 Types char16_t and char32_t [diff.char16]
1 The types char16_t and char32_t are distinct types rather than typedefs to existing integral types.

C.2.2.2 Type wchar_t [diff.wchar.t]
1 wchar_t is a keyword in this International Standard (2.11). It does not appear as a type name defined in any of <stddef>, <cstdlib>, or <cwchar> (21.5).

C.2.2.3 Header <iso646.h> [diff.header.iso646.h]
1 The tokens and, and_eq, bitand, bitor, compl, not_eq, not, or, or_eq, xor, and xor_eq are keywords in this International Standard (2.11). They do not appear as macro names defined in <ciso646>.

C.2.2.4 Macro NULL [diff.null]
1 The macro NULL, defined in any of <locale>, <stddef>, <stdio>, <stdlib>, <string>, <ctime>, or <cwchar>, is an implementation-defined C++ null pointer constant in this International Standard (18.1).

C.2.3 Modifications to declarations [diff.mods.to.declarations]
1 Header <cstring>: The following functions have different declarations:
   — strchr
   — strpbrk
   — strrchr
   — strstr
   — memchr
21.5 describes the changes.
C.2.4  Modifications to behavior  [diff.mods.to.behavior]

1 Header <cstdlib>: The following functions have different behavior:
   — atexit
   — exit
   — abort
   18.4 describes the changes.

2 Header <csetjmp>: The following functions have different behavior:
   — longjmp
   18.8 describes the changes.

C.2.4.1  Macro offsetof(type,member-designator)  [diff offsetof]

1 The macro offsetof, defined in <cstdlib>, accepts a restricted set of type arguments in this International Standard. 18.1 describes the change.

C.2.4.2  Memory allocation functions  [diff malloc]

1 The functions calloc, malloc, and realloc are restricted in this International Standard. 20.6.14 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 145.

<table>
<thead>
<tr>
<th>assert.h</th>
<th>float.h</th>
<th>math.h</th>
<th>stddef.h</th>
<th>tgmath.h</th>
</tr>
</thead>
<tbody>
<tr>
<td>complex.h</td>
<td>inttypes.h</td>
<td>setjmp.h</td>
<td>stdio.h</td>
<td>time.h</td>
</tr>
<tr>
<td>ctype.h</td>
<td>iso646.h</td>
<td>signal.h</td>
<td>stdint.h</td>
<td>uchar.h</td>
</tr>
<tr>
<td>errno.h</td>
<td>limits.h</td>
<td>stdarg.h</td>
<td>stdlib.h</td>
<td>wchar.h</td>
</tr>
<tr>
<td>fenv.h</td>
<td>locale.h</td>
<td>stdbool.h</td>
<td>string.h</td>
<td>wctype.h</td>
</tr>
</tbody>
</table>

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 name 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 std and are then injected into the global namespace scope by explicit using-declarations (7.3.3).

3 [Example: The header <cstdlib> assuredly provides its declarations and definitions within the namespace std. It may also provide these names within the global namespace. The header <stdlib.h> assuredly provides the same declarations and definitions within the global namespace, much as in the C Standard. It may also provide these names within the namespace std. — end example ]

D.6 Old iostreams members [depr.ios.members]

1 The following member names are in addition to names specified in clause 27:

```cpp
namespace std {
    class ios_base {
        public:
            typedef T1 io_state;
            typedef T2 open_mode;
            typedef T3 seek_dir;
            typedef OFF_T streamoff;
            typedef POS_T streampos;
            // remainder unchanged
    };
}
```

2 The type `io_state` is a synonym for an integer type (indicated here as `T1`) that permits certain member functions to overload others on parameters of type `iostate` and provide the same behavior.

3 The type `open_mode` is a synonym for an integer type (indicated here as `T2`) that permits certain member functions to overload others on parameters of type `openmode` and provide the same behavior.

4 The type `seek_dir` is a synonym for an integer type (indicated here as `T3`) that permits certain member functions to overload others on parameters of type `seekdir` and provide the same behavior.

5 The type `streamoff` is an implementation-defined type that satisfies the requirements of type `OFF_T` (27.4.1).

6 The type `streampos` is an implementation-defined type that satisfies the requirements of type `POS_T` (27.2).

7 An implementation may provide the following additional member function, which has the effect of calling `sbumpc()` (27.5.2.2.3):

```cpp
namespace std {
    template<class charT, class traits = char_traits<charT> >
    class basic_streambuf {
        public:
            void stossc();
            // remainder unchanged
    };
}
```

8 An implementation may provide the following member functions that overload signatures specified in clause 27:
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
    };
}
D.7 char* streams [depr.str.strstreams]

The effects of these functions is to call the corresponding member function specified in clause 27.

D.7 char* streams [depr.str.strstreams]

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 [depr.strstreambuf]

```cpp
namespace std {
    class strstreambuf : public basic_streambuf<char> {
    public:
        explicit strstreambuf(streamsize alsize_arg = 0);
        strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
        strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = 0);
        strstreambuf(const char* gnext_arg, streamsize n);
        strstreambuf(signed char* gnext_arg, streamsize n,
                      signed char* pbeg_arg = 0);
        strstreambuf(const signed char* gnext_arg, streamsize n);
        strstreambuf(unsigned char* gnext_arg, streamsize n,
                      unsigned char* pbeg_arg = 0);
        strstreambuf(const unsigned char* gnext_arg, streamsize n);
        virtual ~strstreambuf();

        void freeze(bool freezeflag = 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 alsize;
        // void* (*palloc)(size_t);
        // void (*pfree)(void*);

Draft
1 The class strstreambuf associates the input sequence, and possibly the output sequence, with an object of some character array type, whose elements store arbitrary values. The array object has several attributes.

2 [Note: For the sake of exposition, these are represented as elements of a bitmask type (indicated here as T1) called strstate. The elements are:
   — 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]

3 [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 alsize, the suggested minimum size for a dynamic array object;
   — void* palloc)(size_t), points to the function to call to allocate a dynamic array object;
   — void (*pfree)(void*), points to the function to call to free a dynamic array object.
   — end note]

4 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 [depr.strstreambuf.cons]

explicit strstreambuf(streamsize alsize_arg = 0);

Effects: Constructs an object of class strstreambuf, initializing the base class with streambuf(). The postconditions of this function are indicated in Table 146.

Table 146: strstreambuf(streamsize) 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>alsize_arg</td>
</tr>
<tr>
<td>palloc</td>
<td>a null pointer</td>
</tr>
<tr>
<td>pfree</td>
<td>a null pointer</td>
</tr>
</tbody>
</table>
D.7 `char* streams` Compatibility features 1176

`strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*))`;

Effects: Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 147.

Table 147: `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 postconditions of this function are indicated in Table 148.

Table 148: `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`.  

If `pbeg_arg` is a null pointer, the function executes:

```c
setg(gnext_arg, gnext_arg, gnext_arg + N);
```

Otherwise, the function executes:

```c
setg(gnext_arg, gnext_arg, pbeg_arg);
setp(pbeg_arg, pbeg_arg + N);
```

`strstreambuf(const char* gnext_arg, streamsize n);`
`strstreambuf(const signed char* gnext_arg, streamsize n);`
`strstreambuf(const unsigned char* gnext_arg, streamsize n);`

337) The function signature `strlen(const char*)` is declared in `<cstring>`, (21.5). The macro `INT_MAX` is defined in `<climits>` (18.2).
Effects: Behaves the same as `strstreambuf((char*)gnext_arg,n)`, except that the constructor also sets constant in strmode.

```cpp
virtual ~strstreambuf();
```

Effects: Destroys an object of class `strstreambuf`. The function frees the dynamically allocated array object only if strmode & allocated != 0 and strmode & frozen == 0. (D.7.1.3 describes how a dynamically allocated array object is freed.)

### D.7.1.2 Member functions

```cpp
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.

```cpp
char* str();
```

**Effects:** Calls `freeze()`, then returns the beginning pointer for the input sequence, gbeg.

**Remarks:** The return value can be a null pointer.

```cpp
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

```cpp
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.\textsuperscript{338) If} palloc is not a null pointer, the function calls (*palloc)(n) to allocate the new dynamic array object. Otherwise, it evaluates the expression new charT[n]. In either case, if the allocation fails, the function returns EOF. Otherwise, it sets allocated in strmode.

To free a previously existing dynamic array object whose first element address is p: If pfree is not a null pointer, the function calls (*pfree)(p). Otherwise, it evaluates the expression delete[] p.

If strmode & dynamic == 0, or if strmode & frozen != 0, the function cannot extend the array (reallocate it with greater length) to make a write position available.

\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:

- If c != EOF, if the input sequence has a putback position available, and if (char)c == gnext[-1], assigns gnext - 1 to gnext.
  Returns c.
- If c != EOF, if the input sequence has a putback position available, and if strmode & constant is zero, assigns c to **--gnext.
  Returns c.
- If c == EOF and if the input sequence has a putback position available, assigns gnext - 1 to gnext.
  Returns a value other than EOF.

Returns EOF to indicate failure.

\begin{verbatim}
int_type underflow();
\end{verbatim}

Effects: Reads a character from the input sequence, if possible, without moving the stream position past it, as follows:

- If the input sequence has a read position available, the function signals success by returning \texttt{(unsigned char*)gnext}.
- Otherwise, if the current write next pointer pnext is not a null pointer and is greater than the current read end pointer gend, makes a read position available by assigning to gend a value greater than gnext and no greater than pnext.
  Returns \texttt{(unsigned char*)gnext}.

Returns EOF to indicate failure.

Remarks: The function can alter the number of read positions available as a result of any call.

\begin{verbatim}
pos_type seekoff(off_type off, seekdir way, openmode which = in | out);
\end{verbatim}

\textsuperscript{338) An implementation should consider alsize in making this decision.}
Effects: Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 149.

Table 149: seekoff positioning

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(which &amp; ios::in) != 0</td>
<td>positions the input sequence</td>
</tr>
<tr>
<td>(which &amp; ios::out) != 0</td>
<td>positions the output sequence</td>
</tr>
<tr>
<td>(which &amp; (ios::in</td>
<td>ios::out)) == (ios::in</td>
</tr>
<tr>
<td>Otherwise</td>
<td>the positioning operation fails.</td>
</tr>
</tbody>
</table>

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 150.

Table 150: 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 istreamstream [depr.istrstream]

namespace std {
    class istreamstream : public basic_istream<char> {
        public:
            explicit istreamstream(const char* s);
            explicit istreamstream(char* s);
            istreamstream(const char* s, streamsize n);
            istreamstream(char* s, streamsize n);
            virtual ~istreamstream();

            strstreambuf* rdbuf() const;
            char *str();
        private:
            // strstreambuf sb;
            exposition only
    };
}

The class istreamstream 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 istreamstream constructors [depr.istreamstream.cons]

explicit istreamstream(const char* s);
explicit istreamstream(char* s);

Effects: Constructs an object of class istreamstream, initializing the base class with istream(&sb) and initializing sb with strstreambuf(s,0). s shall designate the first element of an NTBS.

istreamstream(const char* s, streamsize n);
Effects: Constructs an object of class istringstream, initializing the base class with istringstream(&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);

Effects: Constructs an object of class ostrstream, initializing the base class with ostream(&sb), and initializing sb with one of two constructors:
D.7 char* streams Compatibility features 1182

— 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.7.3.2 Member functions [depr.ostrstream.members]

strstreambuf* rdbuf() const;

Returns: (strstreambuf*)&sb.

void freeze(bool freeze1 = true);

Effects: Calls rdbuf()->freeze(freeze1).

char* str();

Returns: rdbuf()->str().

int pcount() const;

Returns: rdbuf()->pcount().

D.7.4 Class strstream [depr.strstream]

namespace std {
    class strstream :
        public basic_iostream<char> {
        public:
            // Types
            typedef char char_type;
            typedef typename char_traits<char>::int_type int_type;
            typedef typename char_traits<char>::pos_type pos_type;
            typedef typename char_traits<char>::off_type off_type;

            // constructors/destructor
            strstream();
            strstream(char* s, int n,
                      ios_base::openmode mode = ios_base::in|ios_base::out);
            virtual ~strstream();

            // Members:
            strstreambuf* rdbuf() const;
            void freeze(bool freeze1 = true);
            int pcount() const;
            char* str();

339) The function signature strlen(const char*) is declared in <cstring> (21.5).
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

```cpp
strstream();
```

**Effects:** Constructs an object of class `strstream`, initializing the base class with `iostream(&sb)`.

```cpp
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

```cpp
virtual ~strstream()
```

**Effects:** Destroys an object of class `strstream`.

```cpp
strstreambuf* rdbuf() const;
```

**Returns:** `&sb`.

### D.7.4.3 `strstream` operations

```cpp
void freeze(bool freezefl = true);
```

**Effects:** Calls `rdbuf()->freeze(freezefl)`.

```cpp
char* str();
```

**Returns:** `rdbuf()->str()`.

```cpp
int pcount() const;
```

Draft
Returns: rdbuf()->pcount().

D.8 Binders

The binders binder1st, bindist, binder2nd, and bind2nd are deprecated. [Note: The template function bind (20.5.11) provides a better solution. — end note ]

D.8.1 Class template binder1st

```cpp
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;
};
```

1 The constructor initializes fn with x and value with y.
2 operator() returns fn(value, x).

D.8.2 bindist

```cpp
template <class Fn, class T>
binder1st<Fn> bindist(const Fn& fn, const T& x);
```

1 Returns: binder1st<Fn>(fn, typename Fn::first_argument_type(x)).

D.8.3 Class template binder2nd

```cpp
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;
public:
  binder2nd(const Fn& x,
    const typename Fn::second_argument_type& y);
  typename Fn::result_type
    operator() const;
};
```

Draft
1. The constructor initializes \( \text{fn} \) with \( x \) and value with \( y \).
2. \( \text{operator()} \) returns \( \text{fn}(x, \text{value}) \).

**D.8.4 bind2nd**

```cpp
template <class Fn, class T>
binder2nd<Fn> bind2nd(const Fn& op, const T& x);
```

- Returns: \( \text{binder2nd<Fn>(op, typename \text{Fn}::second_argument_type(x))} \).

- **Example:**
  ```cpp
  find_if(v.begin(), v.end(), bind2nd(greater<int>(), 5));
  ```
  finds the first integer in vector \( v \) greater than 5;
  ```cpp
  find_if(v.begin(), v.end(), bind2nd(greater<int>(), 5));
  ```
  finds the first integer in \( v \) less than 5. —end example

**D.9 auto_ptr**

The class template auto_ptr is deprecated. [Note: The class template unique_ptr (20.6.11) provides a better solution. —end note]

**D.9.1 Class template 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.

```cpp
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();
    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();
}
```
D.9 **auto_ptr**  

```
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();
```

```

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();

1 Postconditions: *this holds the pointer p.

auto_ptr(auto_ptr& a) throw();

2 Effects: Calls a.release().

3 Postconditions: *this holds the pointer returned from a.release().

template<class Y> auto_ptr(auto_ptr<Y>& a) throw();

4 Requires: Y* can be implicitly converted to X*.

5 Effects: Calls a.release().

6 Postconditions: *this holds the pointer returned from a.release().

auto_ptr& operator=(auto_ptr& a) throw();
```
D.9.1.2 auto_ptr members

X& operator*() const throw();

Requires: get() != 0

Returns: *get()

X* operator->() const throw();

Returns: get()

D.9.1.3 auto_ptr conversions

auto_ptr(auto_ptr_ref<X> r) throw();

Effects: Calls p.release() for the auto_ptr p that r holds.

Postconditions: *this holds the pointer returned from release().
template<class Y> operator auto_ptr_ref<Y>() throw();

Returns: An auto_ptr_ref<Y> that holds *this.

template<class Y> operator auto_ptr<Y>() throw();

Effects: Calls release().

Returns: An auto_ptr<Y> that holds the pointer returned from release().

auto_ptr& operator=(auto_ptr_ref<X> r) throw()

Effects: Calls reset(p.release()) for the auto_ptr p that r holds a reference to.

Returns: *this
Appendix E
(normative)
Universal-character-names

1. This clause lists the complete set of hexadecimal code values that are valid in universal-character-names in C++ identifiers (2.10).

2. 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, 1fca–1fdd, 1fe0–1fec, 1ff0–1ff1, 1ff4–1ff5 |
| Cyrillic: 0401–040d, 040f–044f, 0451–045c, 045e–0481, 0490–04c4, 047–04c8, 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: 0985–099c, 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, 0b9c, 0b9e–0b9f, 0ba3–0ba4, 0ba8–0baa, 0baf–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–0eb0, 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–fb81, fb83–fd3f, fd50–fd8f, fd92–fdca, fd0–f9b, fe70–fe72, fe74, fe76–fe7c, ff21–ff3a, ff41–ff5a, ff66–ffbe, ffc2–ff7, ffc2–ff7, ffc2–ff7, ffd2–ffd7, ffda–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>Section Label</th>
<th>Section Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>accumulate</td>
<td>26.6.1</td>
</tr>
<tr>
<td>adjacent.difference</td>
<td>26.6.4</td>
</tr>
<tr>
<td>adjustfield.manip</td>
<td>27.4.5.2</td>
</tr>
<tr>
<td>alg.adjacent.find</td>
<td>25.1.5</td>
</tr>
<tr>
<td>alg.binary.search</td>
<td>25.3.3</td>
</tr>
<tr>
<td>alg.c.library</td>
<td>25.4</td>
</tr>
<tr>
<td>alg.copy</td>
<td>25.2.1</td>
</tr>
<tr>
<td>alg.count</td>
<td>25.1.6</td>
</tr>
<tr>
<td>alg.equal</td>
<td>25.1.8</td>
</tr>
<tr>
<td>alg.fill</td>
<td>25.2.6</td>
</tr>
<tr>
<td>alg.find</td>
<td>25.1.2</td>
</tr>
<tr>
<td>alg.find.end</td>
<td>25.1.3</td>
</tr>
<tr>
<td>alg.find.first.of</td>
<td>25.1.4</td>
</tr>
<tr>
<td>alg.foreach</td>
<td>25.1.1</td>
</tr>
<tr>
<td>alg.generate</td>
<td>25.2.7</td>
</tr>
<tr>
<td>alg.heap.operations</td>
<td>25.3.6</td>
</tr>
<tr>
<td>alg.lex.comparison</td>
<td>25.3.8</td>
</tr>
<tr>
<td>alg.merge</td>
<td>25.3.4</td>
</tr>
<tr>
<td>alg.min.max</td>
<td>25.3.7</td>
</tr>
<tr>
<td>alg.modifying.operations</td>
<td>25.2</td>
</tr>
<tr>
<td>alg.move</td>
<td>25.2.2</td>
</tr>
<tr>
<td>alg.nonmodifying</td>
<td>25.1</td>
</tr>
<tr>
<td>alg.nth.element</td>
<td>25.3.2</td>
</tr>
<tr>
<td>alg.partitions</td>
<td>25.2.13</td>
</tr>
<tr>
<td>alg.permutation.generators</td>
<td>25.3.9</td>
</tr>
<tr>
<td>alg.random.shuffle</td>
<td>25.2.12</td>
</tr>
<tr>
<td>alg.remove</td>
<td>25.2.8</td>
</tr>
<tr>
<td>alg.replace</td>
<td>25.2.5</td>
</tr>
<tr>
<td>alg.reverse</td>
<td>25.2.10</td>
</tr>
<tr>
<td>alg.rotate</td>
<td>25.2.11</td>
</tr>
<tr>
<td>alg.search</td>
<td>25.1.9</td>
</tr>
<tr>
<td>alg.set.operations</td>
<td>25.3.5</td>
</tr>
<tr>
<td>alg.sort</td>
<td>25.3.1</td>
</tr>
</tbody>
</table>
back.insert.iter.op* 24.4.2.2.3
back.insert.iter.op++ 24.4.2.2.4
back.insert.iter.op= 24.4.2.2.2
back.insert.iter.ops 24.4.2.2
back.insert.iterator 24.4.2.1
back inserter 24.4.2.2.5
bad.alloc 18.5.2.1
bad.cast 18.6.2
bad.exception 18.7.2.1
bad.typeid 18.6.3
base 20.5.3
basefield.manip 27.4.5.3
basic 3
basic.align 3.11
basic.compound 3.9.2
basic.def 3.1
basic.def.odr 3.2
basic.fundamental 3.9.1
basic.funscope 3.3.4
basic.iocons 27.4.4.1
basic.io.members 27.4.4.2
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.hiding 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
cross references

complxh 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
constraints 17.4.3
construct.element 20.6.9
container.adaptors 23.2.5
container.requirements 23.1
containers 23
contents 17.4.1.1
conv 4
conv.array 4.2
conv.bool 4.12
conv.double 4.8
conv.fpint 4.9
conv.fprom 4.6
conv.func 4.3
conv.integral 4.7
conv.lval 4.1
conv.mem 4.11
conv.prom 4.5
conv.ptr 4.10
conv.qual 4.4
conv.rank 4.13
conventions 17.3.2
conversions 22.1.3.2
conversions.buffer 22.1.3.2.3
conversions.character 22.1.3.2.1
conversions.string 22.1.3.2.2
cpp 16
cpp.concat 16.3.3
cpp.cond 16.1
cpp.error 16.5
cpp.include 16.2
cpp.line 16.4
cpp.null 16.7
cpppragma 16.6
cpppragma.op 16.9
cpp.predefined 16.8
cpp.replace 16.3
cpp.rescan 16.3.4
cpp.scope 16.3.5
cpp.stringize 16.3.2
cpp.subst 16.3.1
ctstdint 18.3
cst dint.syn 18.3.1

d

date.time 20.7
dcl.align 7.1.7
dcl.ambig.res 8.2
dcl.array 8.3.4
dcl.asm 7.4
dcl.constexpr 7.1.5
dcl.dcl 7
dcl.decl 8
dcl.enum 7.2
dcl.fct 8.3.5
dcl.fct.def 8.4
dcl.fct.default 8.3.6
dcl.fct.spec 7.1.2
dcl.friend 7.1.4
dcl.init 8.5
dcl.init.aggr 8.5.1
dcl.init.ref 8.5.3
dcl.init.string 8.5.2
dcl.link 7.5
dcl.meaning 8.3
dcl.mptr 8.3.3
dcl.name 8.1
dcl.ptr 8.3.1
dcl.ref 8.3.2
dcl.spec 7.1
dcl.spec.auto 7.1.6.4
dcl.ste 7.1.1
dcl.type 7.1.6
dcl.type.cv 7.1.6.1
dcl.type.elab 7.1.6.3
dcl.type.simple 7.1.6.2
dcl.typedef 7.1.3
default.allocator 20.6.5

draft
diff.header.iso646.h  C.2.2.3
diff.iso   C.1
diff.lex   C.1.1
diff.library C.2
diff.malloc C.2.4.2
diff.mods.to.behavior C.2.4
diff.mods.to.declarations C.2.3
diff.mods.to.declarations C.2.2
diff.mods.to.headers C.2.1
diff.null C.2.2.4
diff offsetof C.2.4.1
diff.special C.1.8
diff.stat C.1.4
diff.wchar.t C.2.2.2
domain.error 19.1.2

e
enumerated.types 17.3.2.1.1
equal.range 25.3.3.3
errno 19.3
except 15
except.access 15.6
except_ctor 15.2
except.handle 15.3
except.nested 18.7.6
except.spec 15.4
except.special 15.5
except.terminate 15.5.1
except.throw 15.1
except.uncaught 15.5.3
except.unexpected 15.5.2
exception 18.7.1
exception.terminate 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.prim.lambda 5.1.1
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.unicast 5.3
expr.unicast.op 5.3.1
expr.xor 5.12
ext.manip 27.6.4
extendid E
extern.names 17.4.3.2.3
extern.types 17.4.3.2.4

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
locales 22.1
localization 22
logic.error 19.1.1
logical.operations 20.5.8
lower.bound 25.3.3.1

M
macro.names 17.4.3.2.1
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 26.5.8.1
mask.array.comp.assign 26.5.8.2
mask.array.fill 26.5.8.3
member.functions 17.4.4.4
member.pointer.adaptors 20.5.13
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
meta.unary 20.4.4
meta.unary.cat 20.4.4.1
meta.unary.comp 20.4.4.2
meta.unary.prop 20.4.4.3
mismatch 25.1.7
move.iter.nonmember 24.4.3.3.14
move.iter.op.+ 24.4.3.3.8
move.iter.op.++ 24.4.3.3.9
move.iter.op.- 24.4.3.3.10
move.iter.op.-- 24.4.3.3.11
move.iter.op.comp 24.4.3.3.13
move.iter.op.const 24.4.3.3.1
move.iter.op.conv 24.4.3.3.3
move.iter.op.decr 24.4.3.3.7
move.iter.op.incr 24.4.3.3.6
move.iter.op.index 24.4.3.3.12
move.iter.op.ref 24.4.3.3.5
move.iter.op.star 24.4.3.3.4
move.iter.op= 24.4.3.3.2
move.iter.ops 24.4.3.3
move.iter.requirements 24.4.3.2
move.iterator 24.4.3.1
move.iterators 24.4.3
multibyte.strings 17.3.2.1.3.2
multimap 23.3.2
multimap.cons 23.3.2.1
multimap.modifiers 23.3.2.2
multimap.ops 23.3.2.3
multimap.special 23.3.2.4
multiset 23.3.4
multiset.cons 23.3.4.1
multiset.special 23.3.4.2

N
namespace.alias 7.3.2
namespace.def 7.3.1
namespace.memdef 7.3.1.2
namespace.qual 3.4.3.2
namespace.std 17.4.3.1
namespace.udecl 7.3.3
namespace.udir 7.3.4
namespace.unnamed 7.3.1.1
narrow.stream.objects 27.3.1
negators 20.5.10
new.delete 18.5.1
new.delete.array 18.5.1.2
new.delete.placement 18.5.1.3
new.delete.single 18.5.1.1
new.handler 18.5.2.2
numarray 26.5
numeric.limits 18.2.1.1
numeric.limits.members 18.2.1.2
numeric.ops 26.6
numeric.requirements 26.1
numeric.special 18.2.1.5
numerics 26
O

objects.within.classes 17.3.2.3
ofstream 27.8.1.10
ofstream.assign 27.8.1.12
ofstream.cons 27.8.1.11
ofstream.members 27.8.1.13
operators 20.2.1
organization 17.4.1
ostream 27.6.2.1
ostream.assign 27.6.2.3
ostream.cons 27.6.2.2
ostream.formatted 27.6.2.6
ostream.formatted.reqmts 27.6.2.6.1
ostream.inserters 27.6.2.6.3
ostream.inserters.arithmetic 27.6.2.6.2
ostream.inserters.character 27.6.2.6.4
ostream iterator 24.5.2
ostream.iterator.cons.des 24.5.2.1
ostream.iterator.ops 24.5.2.2
ostream.manip 27.6.2.8
ostream.seeks 27.6.2.5
ostream.unformatted 27.6.2.7
ostream::sentry 27.6.2.4
ostreambuf.iter.cons 24.5.4.1
ostreambuf.iter.ops 24.5.4.2
ostreambuf.iterator 24.5.4
ostreamstream 27.7.3
ostreamstream.assign 27.7.3.2
ostreamstream.cons 27.7.3.1
ostreamstream.members 27.7.3.3
out.of.range 19.1.5
output.iterators 24.1.2
output.streams 27.6.2
over 13
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.match.ctor 13.3.1.3
over.match.funcs 13.3.1
over.match.oper 13.3.1.2
over.match.ref 13.3.1.6
over.match.viable 13.3.2
over.oper 13.5
over.over 13.4
over.ref 13.5.6
over.sub 13.5.5
over.unary 13.5.1
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 23.2.5.2
priorqueue.cons 23.2.5.2.1
priorqueue.members 23.2.5.2.2
priorqueue.special 23.2.5.2.3
propagation 18.7.5
protection.within.classes 17.4.4.6
ptr.align 20.6.13
push.heap 25.3.6.1

Q

queue 23.2.5.1
queue.defn 23.2.5.1.1
queue.ops 23.2.5.1.2
queue.special 23.2.5.1.3
Cross references 1202

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 26.4.8.3.4
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.gengf 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
rand.eng.lcong 26.4.3.1
rand.eng.mers 26.4.3.2
rand.eng.sub 26.4.3.3
rand.predef 26.4.5
rand.req 26.4.1
rand.req.adapt 26.4.1.4
rand.req.dist 26.4.1.5
rand.req.eng 26.4.1.3
rand.req.genl 26.4.1.1
rand.req.urng 26.4.1.2
rand.synopsis 26.4.2
rand.util 26.4.7
rand.util.canonical 26.4.7.2
rand.util.seedseq 26.4.7.1
random.access.iterators 24.1.5
range.error 19.1.7
re 28
re.alg 28.11
re.alg.match 28.11.2
re.alg.replace 28.11.4
re.alg.search 28.11.3
re.badexp 28.6
re.const 28.5
re.def 28.1
re.err 28.5.3
re.except 28.11.1
re.grammar 28.13
re.iter 28.12
re.matchflag 28.5.2
re.regex 28.8
re.regex.assign 28.8.3
re.regex.const 28.8.1
re.regex.construct 28.8.2
re.regex.locale 28.8.5
re.regex.nmswap 28.8.7.1
re.regex.nonmemb 28.8.7
re.regex.operations 28.8.4
re.regex.swap 28.8.6
re.register 28.12.1
re.register.cnstr 28.12.1.1
re.register.comp 28.12.1.2
re.register.deref 28.12.1.3
re.register.incr 28.12.1.4
re.req 28.2
re.results 28.10
re.results.acc 28.10.3
re.results.all 28.10.5
re.results.const 28.10.1
re.results.form 28.10.4
re.results.nonmember 28.10.7
re.results.size 28.10.2
re.results.swap 28.10.6
re.submatch 28.9
re.submatch.members 28.9.1
re.submatch.op 28.9.2
re.sum 28.3
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 20.6.7
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
streambuf.pub.pback 27.5.2.2.4
streambuf.pub.put 27.5.2.2.5
streambuf.put.area 27.5.2.3.3
streambuf.reqs 27.5.1
streambuf.virt.buffer 27.5.2.4.2
streambuf.virt.get 27.5.2.4.3
streambuf.virt.locales 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.accessors 21.3.7.1
string.capacity 21.3.4
string.classes 21.2
string.cons 21.3.2
string.conversions 21.4
string.io 21.3.8.9
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 21.3.8.8
string.streams 27.7
string::append 21.3.6.2
string::assign 21.3.6.3
string::compare 21.3.7.9
string::copy 21.3.6.7
string::erase 21.3.6.5
string::find 21.3.7.2
string::find.first.not.of 21.3.7.6
string::find.first.of 21.3.7.4
string::find.last.not.of 21.3.7.7
string::find.last.of 21.3.7.5
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 21.3.7.3
string::substr 21.3.7.8
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.runtime 18.8
support.start.term 18.4
support.types 18.1
syntax 1.6
syserr 19.4
syserr.compare 19.4.4
syserr.errcat 19.4.1
syserr.errcat.derived 19.4.1.4
syserr.errcat.nonvirtuals 19.4.1.3
syserr.errcat.objects 19.4.1.5
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 19.4.2.5
syserr.errcode.observers 19.4.2.4
syserr.errcode.overview 19.4.2.1
syserr.errcondition 19.4.3
syserr.errcondition.constructors 19.4.3.2
syserr.errcondition.modifiers 19.4.3.3
syserr.errcondition.nonmembers 19.4.3.5
syserr.errcondition.observers 19.4.3.4
syserr.errcondition.overview 19.4.3.1
syserr.syserr 19.4.5
syserr.syserr.members 19.4.5.2
syserr.syserr.overview 19.4.5.1

temp 14
temp.alias 14.5.7
temp.arg 14.3
temp.arg.explicit 14.8.1
temp.arg.notype 14.3.2
temp.arg.template 14.3.3
temp.arg.type 14.3.1
temp.class 14.5.1
temp.class.order 14.5.5.2
temp.class.spec 14.5.5
temp.class.spec.match 14.5.5.1
temp.class.spec.mfunc 14.5.5.3
temp.decls 14.5
temp.deduct 14.8.2
temp.deduct.call 14.8.2.1
temp.deduct.conv 14.8.2.3
temp.deduct.funcaddr 14.8.2.2
temp.deduct.partial 14.8.2.4
temp.deduct.type 14.8.2.5
temp.dep 14.6.2

temp.dep.candidate 14.6.4.2
temp.dep.constexpr 14.6.2.3
temp.dep.expr 14.6.2.2
temp.dep.res 14.6.4
temp.dep.temp 14.6.2.4
temp.dep.type 14.6.2.1
temp.exp.spec 14.7.3
temp.explicit 14.7.2
temp.fct 14.5.6
temp.fct.spec 14.8
temp.friend 14.5.4
temp.func.order 14.5.6.2
temp.inject 14.6.5
temp.inst 14.7.1
temp.local 14.6.1
temp.mem 14.5.2
temp.mem.class 14.5.1.2
temp.mem.mfunc 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 20.6.8
terminate 18.7.3.3
terminate.handler 18.7.3.1
thread 30
thread.condition 30.4
thread.condition.condvar 30.4.1
thread.condition.condvarany 30.4.2
thread.lock 30.3.3
thread.lock.algorithm 30.3.4
thread.lock.guard 30.3.3.1
thread.lock.unique 30.3.3.2

Draft
util.smartptr.getdeleter 20.6.12.2.11
util.smartptr.shared 20.6.12.2
util.smartptr.shared.assign 20.6.12.2.3
util.smartptr.shared.cast 20.6.12.2.10
util.smartptr.shared.cmp 20.6.12.2.7
util.smartptr.shared.const 20.6.12.2.1
util.smartptr.shared.create 20.6.12.2.6
util.smartptr.shared.dest 20.6.12.2.2
util.smartptr.shared.io 20.6.12.2.8
util.smartptr.shared.mod 20.6.12.2.4
util.smartptr.shared.obs 20.6.12.2.5
util.smartptr.shared.spec 20.6.12.2.9
util.smartptr.weak 20.6.12.3
util.smartptr.weak.assign 20.6.12.3.3
util.smartptr.weak.cmp 20.6.12.3.6
util.smartptr.weak.const 20.6.12.3.1
util.smartptr.weak.dest 20.6.12.3.2
util.smartptr.weak.mod 20.6.12.3.4
util.smartptr.weak.obs 20.6.12.3.5
util.smartptr.weak.spec 20.6.12.3.7
util.smartptr.weakptr 20.6.12.1
utilities 20
utility 20
utility.arg.requirements 20.1.1
utility.requirements 20.1

valarray.access 26.5.2.3
valarray.assign 26.5.2.2
valarray.binary 26.5.3.1
valarray.cassign 26.5.2.6
valarray.comparison 26.5.3.2
valarray.cons 26.5.2.1
valarray.members 26.5.2.7
valarray.nonmembers 26.5.3
valarray.special 26.5.3.4
valarray.sub 26.5.2.4
valarray.synopsis 26.5.1
valarray.transcend 26.5.3.3
valarray.unity 26.5.2.5
value.error.codes 17.4.4.9
vector 23.2.6
vector.bool 23.2.7
vector.capacity 23.2.6.2
vector.cons 23.2.6.1
vector.data 23.2.6.3
vector.modifiers 23.2.6.4
vector.special 23.2.6.5

W
wide.characters 17.3.2.1.3.5
wide.stream.objects 27.3.2

X
xref F

Y

Z
Index

!, see logical negation operator
! =, see inequality operator
(), see function call operator
    function declarator, 176
*, see indirection operator, see multiplication operator
    pointer declarator, 172
+, 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 pointer to member operator
,..., see ellipsis
/, see division operator
:
    field declaration, 210
    label specifier, 119
::, see scope resolution operator
::*

* , see pointer to member declarator, 174
<, see less than operator
    template and, 307, 308
<=, see left shift operator
<=, see less than or equal to operator
<condition_variable>, 1126
<mutex>, 1112
<thread>, 1106
=, 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, 175
#define, 394
#else, 392
#endif, 392
#error, 399
#endif, 392
#include, 392, 416
#line, 399
#pragma, 399
#undef, 396, 417
% , see modulus operator
& , see address-of operator, see bitwise AND operator
    reference declarator, 173
&&, see logical AND operator
`. , see bitwise exclusive OR operator
## operator, 395
# operator, 395
basic_ios::failure argument
    implementation-defined, 968
const object
    undefined change to, 139
exception::what message
    implementation-defined, 446
friend function
    nested class, 212
delete, 103, 106
new, 103
operator bool
    basic_ios, 967
\ , see backslash
    _cplusplus, 400
    _DATE__, 400
    _FILE__, 400
    _LINE__, 400
{}
alias, 152
alignment
    extended, 74
    fundamental, 74
alignment requirement
    implementation-defined, 74
all
    bitset, 767
allocation
    alignment storage, 103
    implementation defined bit-field, 210
    unspecified, 202
allocator, 769, 773, 776, 779, 1074
allocator, 537
alpha()
    gamma_distribution<>, 904
always_noconv
codecvt, 643
ambiguity
    base class member, 218
    class conversion, 221
    declaration type, 131
    declaration versus cast, 169
    declaration versus expression, 126
    function declaration, 188
    member access, 218
    parentheses and, 102
ambiguity detection
    overloaded function, 277
Amendment 1, 418
any
    bitset, 767
append
    basic_string, 596
apply
    valarray, 926
arg, 870
    complex, 868
argc, 58
argument, 2, 420, 421, 456
    access checking and default, 230
    binding of default, 181
    evaluation of default, 181, 182
    example of default, 180, 181
    overloaded operator and default, 297
    reference, 89
    scope of default, 182
    template, 309
    type checking of default, 181
arguments
    implementation-defined order of evaluation of function, 182
argument and name hiding
    default, 182
argument and virtual function
    default, 183
argument list
    empty, 177
    variable, 177
argument passing, 89
    reference and, 194
argument substitution, 394
argument type
    unknown, 177
argv, 58
arithmetic
    pointer, 109
    unsigned, 70
array, 178
    bound, 175
    const, 72
    delete, 105
    multidimensional, 176
    new, 102
    overloading and pointer versus, 275
    sizeof, 100
    storage of, 176
<array>, 701
array, 705, 707
    as aggregate, 705
    begin, 705
    contiguous storage, 705
    data, 706
    end, 705
    get, 707
    initialization, 705, 706
    max_size, 705
    size, 705, 706
    swap, 706
    tuple interface to, 707
    zero sized, 707
array size
Draft
default, 175
arrow operator, see class member access operator
asin, 929, 943
   complex, 869
asinh, 943
   complex, 869
asm
   implementation-defined, 162
assembler, 162
<assert.h>, 416
assign
   basic_string, 597
deque, 710
list, 722
vector, 737
basic_regex, 1063
swap
   function, 527
assignment
   and lvalue, 115
   conversion by, 115
   reference, 194
assignment-expression, 115
assignment-operator, 115
assignment operator
   copy, 266
   overloaded, 297
associative containers
   unordered, see unordered associative containers
at
   basic_string, 596
map, 748
atan, 929, 943
   complex, 869
atan2, 929, 943
atanh, 943
complex, 869
atexit, 60, 415, 436
auto_ptr, 561, 1185
   auto_ptr, 1186
awk, 1054

b()
   cauchy_distribution>, 910
   extreme_value_distribution>, 906
   uniform_int_distribution>, 897
   uniform_real_distribution>, 898
   weibull_distribution>, 905
   back
      basic_string, 596
      back_insert_iterator, 801
      back_insert_iterator, 802
      back_inserter, 803
      backslash character, 25
      bad
         basic_ios, 968
         bad_alloc, 103, 438, 442
         bad_alloc, 442
         bad_alloc::what
            implementation-defined, 442
         bad_cast, 92, 444
         bad_cast, 444
         bad_cast::what
            implementation-defined, 444
         bad_exception, 388, 446
         bad_exception, 446, 447
         bad_exception::what
            implementation-defined, 447
         bad_function_call, 523
         bad_function_call, 523
         bad_typeid, 93, 444
         bad_typeid, 444, 445
         bad_typeid::what
            implementation-defined, 445
         bad_weak_ptr, 558
         bad_weak_ptr, 558
         what, 558
   baseSpecifier, 215
   baseSpecifier-list, 215
base class, 215, 216
   direct, 215
   indirect, 215
   private, 232
   protected, 232
   public, 232
base class virtual, see virtual base class
basic_filebuf, 948, 1027
   basic_filebuf, 1028
   basic_filebuf<char>, 1026
   basic_filebuf<wchar_t>, 1026
   basic_fstream, 948, 1039
   basic_fstream, 1040
basic_ifstream, 948, 1034
   basic_ifstream, 1035
basic_ifstream<char>, 1026
basic_ifstream<wchar_t>, 1026
basic_ios, 948, 963
   basic_ios, 965
basic_ios<char>, 953
basic_ios<wchar_t>, 953
basic_iostream, 996
   basic_iostream, 997
basic_istream, 948, 983
   basic_istream, 986
basic_istream<char>, 982
basic_istream<wchar_t>, 982
basic_istreambuf_iterator, 948
basic_iostreamstream, 948, 1019
   basic_iostreamstream, 1020
basic_iostreamstream<char>, 1013
basic_iostreamstream<wchar_t>, 1013
basic_ofstream, 948, 106
   basic_ofstream, 1037
basic_ofstream<char>, 1026
basic_ofstream<wchar_t>, 1026
basic_ostream, 948, 1070
   basic_ostream, 1001
basic_ostream<char>, 983
basic_ostream<wchar_t>, 983
basic_ostreambuf_iterator, 948
basic_ostringstream, 948, 1023
   basic_ostringstream, 1024
   basic_ostringstream<char>, 1023
   basic_ostringstream<wchar_t>, 1023
basic_regex, 1046, 1059, 1085
   assign, 1063
   basic_regex, 1061, 1062
   constants, 1061
   operator=, 1062, 1063
   swap, 1064
basic_stringbuf, 948, 972
   basic_stringbuf, 974
basic_stringbuf<char>, 971
basic_stringbuf<wchar_t>, 971
basic_string, 585, 608, 1012
basic_stringbuf, 948, 1013
   basic_stringbuf, 1014
basic_stringbuf<char>, 1013
basic_stringbuf<wchar_t>, 1013
   basic_stringstream, 948, 1023
   basic_stringstream, 1024
   basic_stringstream<char>, 1023
   basic_stringstream<wchar_t>, 1023
   before
   type_info, 443
begin
   basic_string, 594
   array, 705
   match_results, 1073
   unordered associative containers, 700
   behavior
   conditionally-supported, 2, 5
   default, 404, 409
   implementation-defined, 3, 666
   locale-specific, 3
   required, 406, 409
   undefined, 4
   unspecified, 4
   Ben, 276
Bernoulli distributions, 898–901
bernoulli_distribution, 898
   constructor, 899
   discrete probability function, 898
   p(), 899
beta()
   gamma_distribution<> , 904
bidirectional_iterator_tag, 793, 794
binary function, 512, 513, 523
binary_function, 511
binary_negate, 517
binary_search, 847
binary operator
   interpretation of, 297
   overloaded, 297
bind, 518–520
bindist, 1184
bind2nd, 1185
binder1st, 1184
binder2nd, 1184
binding
   reference, 194
binomial_distribution<>, 899
   constructor, 899
   discrete probability function, 899
   p(), 900
   t(), 900

Draft
bit-field, 210
  address of, 210
  alignment of, 210
  implementation-defined sign of, 210
  implementation defined alignment of, 210
  type of, 210
  unnamed, 210
  zero width of, 210
bit_and, 516
bit_or, 516
bit_xor, 517
<biset>, 761
bitset, 761
  bitset, 763
block
  initialization in, 125
block scope; see local scope, 37
block structure, 125
body
  function, 184
bool()
  basic_istream operator, 987
  basic_ostream operator, 1002
boolalpha, 968
Boolean, 210
Boolean literal, 29
boolean-literal, 29
Boolean type, 70
bound arguments, 519
bound, of array, 175
bucket
  unordered associative containers, 699
bucket_count
  unordered associative containers, 699
bucket_size
  unordered associative containers, 700
buckets, 696
built-in type; see fundamental type, 69
byte, 100
C
  linkage to, 163
c-char, 24
c-char-sequence, 24
c_str
  basic_string, 602
cabs
  complex, 868
cacos
  complex, 869
cacosh
  complex, 869
call, see also function call, member function call, overloaded function call, virtual function call
  operator function, 296
  pseudo destructor, 90
call signature, 510
call wrapper, 510, 511
  forwarding, 511
  simple, 511
call wrapper type, 510
callable object, 510, 525
callable type, 510
calloc, 572, 1169
capacity
  basic_string, 595
  vector, 737
carry
  subtract_with_carry_engine<>, 884
casin
  complex, 869
casinh
  complex, 869
<cassert>, 416
cast
  base class, 95
  const, 97
  derived class, 95
  dynamic, 91, 444
  integer to pointer, 96
  lvalue, 94, 96
  pointer to function, 96
  pointer to integer, 96
  pointer to member, 95, 96
  reference, 94, 97
  reinterpret, 96
  reinterpret_cast
    lvalue, 96
    reference, 97
    static, 94
static_cast
  lvalue, 94
Draft
reference, 94
undefined pointer to function, 96
cast-expression, 106
casting, 89, 107
catan
complex, 869
catanh
complex, 869
catch, 379
category
locale, 622
cauchy_distribution<>, 910
a(), 910
b(), 910
constructor, 910
probability density function, 910
cbrt, 943
<ccomplex>, 871
cerr, 951
<cerrno>, 417
<cfenv>, 860
char
implementation-defined sign of, 69
char-like object, 575
char-like type, 575
char16_t character, 24
char32_t character, 24
char_class_type
regex_traits, 1057
Regular Expression Traits, 1044
character, 404
decimal-point, 411
multibyte, 3
set
basic execution, 6
basic source, 16
signed, 69
underscore, 417, 418
in identifier, 21
character-literal, 24
character string, 27
checking
point of error, 335
syntax, 335
chi_squared_distribution<>, 909
constructor, 909
n(), 909
probability density function, 909
cin, 951
<ciso646>, 1168
class, 71, 197
abstract, 226
base, 418, 421, 422
cast to incomplete, 107
constructor and abstract, 227
definition, 33
derived, 421
linkage of, 56
linkage specification, 164
pointer to abstract, 226
polymorphic, 222
scope of enumerator, 147
standard-layout, 198
template, 763
trivial, 197
unnamed, 135
class-key, 197
class-name, 197
class-specifier, 197
classes
narrow-oriented iostream, 405
wide-oriented iostream, 406
classic
locale, 628
classic_table
cctype<char>, 641
class base, see base class
class derived, see derived class
class local, see local class
class member, see also member
class name, 168
elaborated, 143, 200
point of declaration, 201
scope of, 199
typedef, 135, 201
class nested, see nested class
class object
assignment to, 116
const, 72
member, 202
concatenation
  string, 28
  undefined string literal, 28

condition, 120

conditions
  rules for, 120

conditional-expression
  throw-expression in, 114

cnj, 870
  complex, 868

consistency
  linkage, 132
  linkage specification, 164
  type declaration, 58

const, 72
  constructor and, 206, 244
  destructor and, 206, 250
  linkage of, 55, 132
  overloading and, 275

const_local_iterator, 696
  unordered associative containers, 696

const_mem_fun1_ref_t, 522
const_mem_fun1_t, 522
const_mem_fun_ref_t, 522
const_mem_fun_t, 521
const_pointer_cast
  shared_ptr, 566

constant, 22, 83
  enumeration, 146
  null pointer, 79

constant-expression, 116

constant-initializer, 201

constructor, 243
  address of, 244
  array of class objects and, 255
  conversion by, 248
  copy, 244, 246, 263, 413
  exception handling, 382
  inheritance of, 244
  non-trivial, 244
  random number engine requirement, 873
  type of, 244
  union, 208
  unspecified argument to, 105

constructor call
  explicit, 245
constructor conversion by, see also user-defined conversion
constructor default, see default constructor context
    non-deduced, 371
control line, see preprocessing directive convention, 409
conversion
    argument, 177
    array pointer, 76
    array-to-pointer, 76
    Boolean, 80
class, 247
derived-to-base, 288
floating point, 78
floating-integral, 79
function-to-pointer, 76
implementation-defined floating point, 78
implementation defined pointer integer, 96
implicit, 75, 247
implicit user-defined, 247
inheritance of user-defined, 250
integer, 78
integer rank, 80
lvalue-to-rvalue, 76, 1158
overload resolution and, 286
overload resolution and pointer, 295
pointer, 79
pointer to function, 76
pointer to member, 79
    void*, 80
return type, 125
reverse_iterator, 798
signed unsigned integer, 78
standard, 75
static user-defined, 250
type of, 249
user-defined, 247–249
virtual user-defined, 250
conversion operator, see conversion function
conversion rank, 289
conversion-function-id, 249
conversions
    qualification, 76
    usual arithmetic, 82
conversion explicit type, see casting
conversion function, see also user-defined conversion
copy
    class object, 263
copy, 834
    basic_string, 602
copy_backward, 834
copyfmt
    basic_ios, 966
copysign, 943
copy assignment operator
    implicitly-declared, 266
copy constructor
    implicitly-declared, 264
cos, 929, 943
    complex, 869
cosh, 929, 943
    complex, 869
count, 832
    bitset, 766
    unordered associative containers, 699
count_if, 832
cout, 951
cref
    reference_wrapper, 514
<csjmp>, 418
cshift
    valarray, 926
<cstdarg>, 418
<cstdio>, 950–952, 1026, 1030, 1168
<cstdio>, 1031
<cstdlib>, 58, 60, 415, 1168, 1172
<cstring>, 412, 1168
<cstring>, 412, 1176, 1182
<ctgmath>, 941
<ctime>, 1168
<ctime>, 621
tor-initializer, 256
cfunc, 634
cfunc<char>
    cfunc<char>, 639
cfunc_byname, 638
<cuchar>, 412, 418
cv-qualifier, 72
INDEX

cv-qualified, 168
<wchar>, 413, 418, 1168
<wctype>, 418

DAG
  multiple inheritance, 217, 218
  non-virtual base class, 218
  virtual base class, 217, 218

data
  basic_string, 603
  vector, 738
  array, 706

data member, see member
  static, 206

date_order
  time_get, 661
deadlock, 404
deallocation, see delete
dec, 970
dec, 1005
decimal-literal, 22
decimal_point
  numpunct, 657
decl-specifier, 131
declaration, 31, 129
  extern reference, 194
  typedef as type, 134
  access, 234
  array, 175
  asm, 162
  bit-field, 210
  class member, 201
  class name, 32
  constant pointer, 172
  default argument, 180
  definition versus, 31
  ellipsis in function, 89, 177
  enumerator point of, 37
  extern, 32
  forward, 133
  forward class, 200
  function, 31, 177
  member, 201
  multiple, 58
  name, 31
  overloaded, 273

overloaded name and friend, 237
parameter, 177
parentheses in, 169, 172
pointer, 172
reference, 173
register, 132
static member, 32
storage class, 131
type, 171
typedef, 32
declaration, 129
declaration-statement, 125
declaration hiding, see name hiding
declaration matching
  overloaded function, 276
declarator, 130, 167
  meaning of, 171
  multidimensional array, 175
declarator, 167
declarator-id, 168
decrement operator
  overloaded, 298
default
  access control, 229
default-initialization, 187
default_random_engine, 891
defaultfloat, 971
default argument
  overload resolution and, 285
default constructor, 244
default initializers
  overloading and, 275
definition, 31, 403
  static member, 208
  alternate, 418
  class, 197, 201
  class name as type, 199
  constructor, 184
declaration as, 130
  empty class, 197
  function, 184
  local class, 212
  member function, 203
  namespace, 148
  nested class, 211
  pure virtual function, 226

Draft
scope of class, 199
virtual function, 224
delete, 61, 105, 106, 253
operator, 572
destructor and, 106, 251
operator, 418, 419, 439, 440
overloading and, 63
type of, 253
undefined, 106
delete-expression, 105
densities()
    piecewise_constant_distribution<>, 915
deprecated features, 91, 100
<deque>, 701
deque, 707
dereferencing, see also indirection
derivation, see inheritance
derived class, 215
    most, 7
    overloading and, 276
derived object
    most, 7
destructor, 250, 413
    default, 250
    exception handling, 382
    non-trivial, 250
    program termination and, 251
    pure virtual, 251
    union, 208
    virtual, 251
destructor call
    explicit, 251
    implicit, 251
digit, 20
digit-sequence, 26
digraph, 18
direct-abstract-declarator, 169
direct-declarator, 168
direct-new-declarator, 101
directed acyclic graph, see DAG
directive
    error, 399
    null, 400
    pragma, 399
    preprocessing, 389
discard()
INDEX

moneypunct, 672

do_get
  messages, 674
  money_get, 667
  num_get, 648

do_get_date
  time_get, 663

do_get_monthname
  time_get, 663

do_get_time
  time_get, 662

do_get_weekday
  time_get, 663

do_get_year
  time_get, 663

do_grouping
  moneypunct, 672
  numpunct, 657

do_hash
  collate, 659

do_in
  codecvt, 643

do_is
  ctype, 636

do_length
  codecvt, 645

do_max_length
  codecvt, 645

do_narrow, 640
  ctype, 637

do_neg_format
  moneypunct, 672

do_negative_sign
  moneypunct, 672

do_open
  messages, 674

do_out
  codecvt, 643

do_pos_format
  moneypunct, 672

do_positive_sign
  moneypunct, 672

do_put
  money_put, 669
  num_put, 652
  time_put, 665

do_scan_is
  ctype, 636

do_scan_not
  ctype, 636

do_thousands_sep
  moneypunct, 672
  numpunct, 657

do_tolower
  ctype, 637

do_toupper
  ctype, 637

do_transform
  collate, 659

do_truename
  numpunct, 657

do_unshift
  codecvt, 645

do_widen
  codecvt, 645
  ctype, 637

domain_error, 454

domain_error, 454
dominance
  virtual base class, 220
dot operator, see class member access operator
dynamic binding, see virtual function
dynamic_pointer_cast
  shared_ptr, 566
eback
  basic_streambuf, 977
ECMAScript, 1054, 1085
egptr
  basic_streambuf, 977
egrep, 1054
elaborated-type-specifier, 143
elaborated type specifier, see elaborated class name elision
  copy constructor, 268
ellipsis
  conversion sequence, 89, 290
  overload resolution and, 285
else, 120
empty, 794
  basic_string, 595
  match_results, 1073
  enable_shared_from_this, 570

Draft
∼enable_shared_from_this, 570
enable_shared_from_this, 570
operator=, 570
shared_from_this, 570
encoding
    multibyte, 28
encoding
    codecvt, 643
end
    basic_string, 594
    array, 705
    match_results, 1073
    unordered associative containers, 700
end-of-file, 768
endl, 1008
endl, 1004
ends, 1008
game, see random number engine
game adaptor, see random number engine adaptor
games with predefined parameters
    knuth_b, 891
    minstd_rand, 891
    minstd_rand0, 890
    mt19937, 891
    ranlux24, 891
    ranlux24_base, 891
    ranlux48, 891
    ranlux48_base, 891
entity, 31
entropy()
random_device, 892
enum, 71
    overloading and, 274
type of, 145, 146
    underlying type, 146
enumeration, 145, 146
    linkage of, 56
    scoped, 146
    unscoped, 146
enumeration scope, 40
enumeration type
    conversion to, 95
    static_cast
    conversion to, 95
enumerator
    definition, 33
    value of, 146
enumerator, 146
enum name
    typedef, 135
environment
    program, 58
eof
    basic_ios, 968
epptr
    basic_streambuf, 978
eq
    char_traits, 603–606
equal, 832
    istreambuf_iterator, 817
equal_range, 846
    unordered associative containers, 699
equal_to, 515
equality-expression, 112
equivalence
    template type, 316
type, 134, 199
equivalent parameter declarations, 274
    overloading and, 274
erase
    basic_string, 599
deque, 711
list, 723
vector, 738
    unordered associative containers, 699
erf, 943
erfc, 943
error_type, 1055–1057
escape-sequence, 24
escape character, see backslash
escape sequence
    undefined, 25
evaluation
    order of argument, 89
    unspecified order of, 10, 59
    unspecified order of argument, 89
    unspecified order of function call, 89
example
    *const, 172
    static member, 208
array, 175
class definition, 202
const, 172
constant pointer, 172
constructor, 245
constructor and initialization, 255
declaration, 32, 179
declarator, 169
definition, 32
delete, 253
derived class, 215
derived class and delete, 254
dead, 177
enumeration, 147
explicit destructor call, 252
explicit qualification, 219
friend, 200
friend function, 236
function declaration, 178
function definition, 184
linkage consistency, 132
local class, 212
member function, 205, 236
member name access, 235
nested type name, 213
nested class, 211
nested class definition, 211, 241
nested class forward declaration, 212
pointer to member, 174
pure virtual function, 226
scope of delete, 254
scope resolution operator, 219
subscripting, 175
typedef, 134
type name, 169
unnamed parameter, 184
variable parameter list, 177
virtual function, 223, 224
exception
allowing an, 386
arithmetic, 81
bad_function_call, 523
bad_weak_ptr, 558
handling, 379
object, 381
undefined arithmetic, 81
<exception>, 445
exception
b(), 906
constructor, 906
probability density function, 906
fabs
complex, 868
facet
locale, 625
fail
basic_ios, 968
failed
ostreambuf_iterator, 818
failure
ios_base::failure, 956
false
numpunct, 657
fclose, 1031
fclose, 1031
fdim, 943
FE_ALL_EXCEPT, 861
FE_DFL_ENV, 861
FE_DIVBYZERO, 861
FE_DOWNWARD, 861
FE_INEXACT, 861
FE_INVALID, 861
FE_OVERFLOW, 861
FE_TONEAREST, 861
FE_TOWARDZERO, 861
<fenv.h>, 861
fenv_t, 861
fegetexceptflag, 861
fegetround, 861
feholdexcept, 861
<fenv.h>, 861
fenv_t, 861
feraiseexcept, 861
fesetenv, 861
fesetexceptflag, 861
fesetround, 861
fetestexcept, 861
feupdateenv, 861
fexcept_t, 861
file, 15
source, 15, 416, 418
filebuf, 948, 1026
implementation-defined, 1033
fill, 837
basic_ios, 966
gslice_array, 935
indirect_array, 938
mask_array, 937
slice_array, 932
fill_n, 837
find, 830
basic_string, 603
unordered associative containers, 699
find_end, 830
find_first_not_of
basic_string, 605
find_first_of, 831
basic_string, 604
find_if, 830
find_last_not_of
basic_string, 606
find_last_of
basic_string, 605
finite state machine, 1043
fisher_f_distribution<>, 911
constructor, 911
m(), 911
n(), 911
probability density function, 911
fixed, 971
flags
ios_base, 634, 959
flip
bitset, 765
float_round_style, 431
floating-literal, 26
floating-suffix, 26
floating point type, 70
implementation-defined, 70
floor, 943
flush, 959, 987, 1002, 1008
basic_ostream, 1008
fma, 943
fmax, 943
fmin, 943
fmtflags
ios_base, 956
Draft
ios, 1009
fopen, 1030
for
    scope of declaration in, 124
for_each, 830
formal argument, see parameter
format
    match_results, 1074
format specifier, 1043
format_default, 1053, 1055
format_first_only, 1053, 1055, 1079
format_no_copy, 1053, 1055, 1079
format_sed, 1053, 1055
forward, 480
forward_iterator_tag, 793, 794
<forward_list>, 702
forwading call wrapper, 511
fpclassify, 945
fpos, 953, 962, 963
fractional-constant, 26
free, 572
freeze
    ostrstream, 1182
    strstreambuf, 1177
    strstream, 1183
free store, see also new, delete
frexp, 943
friend
    virtual and, 224
    access specifier and, 238
    class access and, 236
    inheritance and, 238
    local class and, 238
    template and, 323
friend function
    access and, 235
    inline, 237
    linkage of, 237
    member function and, 236
front
    basic_string, 596
front_insert_iterator, 803
    front_insert_iterator, 803
front_inserter, 804
fseek, 1030
fstream, 1026
function, see also friend function, member function, in-
line function, virtual function, 178
allocation, 62, 102
comparison, 404
conversion, 249
deallocation, 63, 106, 253
definition, 33
global, 418, 420, 421
handler, 405
linkage specification overloaded, 165
modifier, 405
observer, 405
operator, 296
plain old, 452
pointer to member, 108
replacement, 405
reserved, 406
viable, 278
virtual member, 418, 421
function, 523
~function, 527
assign, 527
bool conversion, 527
function, 525
invocation, 528
operator(), 528
operator=, 526, 527
swap, 527, 529
target, 528
type, 528
function objects
    binders, 518–520
    mem_fn, 523
    reference_wrapper, 512
    return type, 511–512
    wrapper, 523–529
function-body, 184
function-definition, 184
function-specifier, 133
function-try-block, 379
<functional>, 506
functions
    candidate, 346
function argument, see argument
function call, 89
    recursive, 89
    undefined, 96
function call operator
    overloaded, 298
function overloaded, see overloading
function parameter, see parameter
function prototype, 38
function return, see return
function return type, see return type
function virtual, see virtual function
fundamental alignment, 74
fundamental type
destructor and and, 252
fundamental type conversion, see conversion, user-defined conversion
gamma_distribution<>, 903
    alpha(), 904
    beta(), 904
    constructor, 904
    probability density function, 903
gbump
    basic_streambuf, 978
gcount
    basic_istream, 991
general_pdf_distribution<>, 916
    constructor, 916
    probability density function, 916
    xmax(), 917
    xmin(), 917
generate, 837
generate()
    seed_seq, 894
generate_canonical<>(), 895
generate_n, 837
generated destructor, see default destructor
generation algorithm
discard_block_engine<>, 885
    independent_bits_engine<>, 887
    linear_congruential_engine<>, 881
    mersenne_twister_engine<>, 882
    random number engine, 873
    shuffle_order_engine<>, 888
    subtract_with_carry_engine<>, 884
    xor_combine_engine<>, 889
geometric_distribution<>, 900
    constructor, 900
    discrete probability function, 900
    p(), 900
gget
    auto_ptr, 1187
    basic_istream, 991
    money_get, 667
    num_get, 647
    array, 707
    pair, 484
    reference_wrapper, 513
    shared_ptr, 563
tuple, 492
gget_date
time_get, 661
gget_deleter
    shared_ptr, 566
gget_money, 1010
gget_monthname
time_get, 661
gget_temporary_buffer, 544, 545
gget_time, 1011
time_get, 661
gget_weekday
time_get, 661
gget_year
time_get, 661
ggetline
    basic_istream, 993, 994
    basic_string, 612
ggetloc
    basic_istream, 993
    ios_base, 960
gglobal
    locale, 628
ggood
    basic_ios, 968
ggoto
    initialization and, 125
gpstr
    basic_streambuf, 977
ggrammar, 1133
    regular expression, 1085
ggreater, 515
greater_equal, 516
grep, 1054
grouping
   num_punct, 657
gslice
gslice, 934
class, 932
gslice_array, 934
handler
   exception, 382, 422
   incomplete_type_in_exception, 382
handler, 379
handler_seq, 379
has_facet
   locale, 628
hash, 529
   collate, 659
   instantiation_restrictions, 529
   operator(), 529
hash_code, 696
hash_function, 695
hash_tables, see unordered associative containers
hash_function
   unordered_associative_containers, 697
hasher
   unordered_associative_containers, 696
header
   C, 416, 418, 420, 1171
header_name, 19
headers
   C++, 414
hex, 970
hex_quad, 17
hexadecimal-digit, 22
hexadecimal_escape_sequence, 24
hexadecimal_literal, 22
hexfloat, 971
hiding; see name hiding, 40
hypot, 943
id
   qualified, 84
id
   locale, 625
id_expression, 83

id_expression, 83
identifier, 20, 83, 130
identifier, 20
identifier_nondigit, 20
identity, 480
ifstream, 948, 1026
ignore
   basic_istream, 994
ilogb, 943
imag, 870
   complex, 868
imbue
   basic_filebuf, 1033
   basic_ios, 966
   basic_streambuf, 978
   ios_base, 960
immolation
   self, 356
implementation
   freestanding, 415
   hosted, 415
implementation_defined, 78, 195, 415, 418, 421, 424, 437,
   442, 444, 446, 447, 960, 1019, 1032, 1168
implementation_dependent, 987, 1002
implementation_generated, 32
implicitly_declared_default_constructor, 244, see also default_constructor
implicit_object_argument, 278
implied_object_parameter, 278
   implicit_conversion_sequences, 278
in
   codecvt, 643
in_avail
   basic_streambuf, 976
includes, 848
inclusion
   conditional, 391
   source_file, 392
incomplete, 109
increment
   bool, 91, 100
increment_operator
   overloaded, 298
independent_bits_engine, 886
   generation_algorithm, 887
   state, 886

Draft
template parameters, 887
textual representation, 887
transition algorithm, 886
indirect_array, 937
indirection, 99
inheritance, 215, see also multiple inheritance
Init
  ios_base::Init, 959
init
  basic_ios, 986, 1001
init-declarator, 167
init-declarator-list, 167
initialization, 186
  array, 190
  array of class objects, 193, 255
  automatic, 125, 126
  automatic object, 187
  base class, 256, 257
  character array, 193
  class member, 188
  class object, 190, 255
  const, 139, 189
  const member, 258
  constant, 59
  constructor and, 255
  copy, 188
  default, 187
  default constructor and, 255
  definition and, 130
  direct, 188
  dynamic, 58
  jump past, 125
  local static, 126
  member, 256
  member object, 257
  order of, 58, 216
  order of base class, 258
  order of member, 258
  order of virtual base class, 258
  overloaded assignment and, 255
  parameter, 88
  reference, 174, 193
  reference member, 258
  run-time, 58
  static member, 208
  static object, 58
  static object, 187
  struct, 190
  union, 193, 209
  virtual base class, 258, 266
initialization class object, see also constructor
initializer
  base class, 184
  constant, 201
  member, 184
  scope of member, 259
  temporary and declarator, 246
initializer, 186
initializer-clause, 186
initializer-list, 186
injected-class-name, 197
inline, 420
inline
  linkage of, 55
  inline function, 133
inner_product, 940
inplace_merge, 848
input_iterator_tag, 793, 794
emplace
  deque, 711
insert
  basic_string, 598
  deque, 711
  list, 723
  vector, 738
  unordered associative containers, 698, 699
push_back
  deque, 711
push_front
  deque, 711
insert_iterator, 804
  insert_iterator, 805
 inserter, 806
instantiation
  explicit, 351
  point of, 345
  template implicit, 348
int
  bool promotion to, 78
int16_t, 434
int32_t, 434
int64_t, 434
Draft
int8_t, 434
int_fast16_t, 434
int_fast32_t, 434
int_fast64_t, 434
int_fast8_t, 434
int_least16_t, 434
int_least32_t, 434
int_least64_t, 434
int_least8_t, 434
integer-literal, 22
integer-suffix, 23
internal, 970
interval boundaries
    piecewise_constant_distribution<>, 914
intervals()
    piecewise_constant_distribution<>, 915
intptr_t, 434
invalid_argument, 455, 763, 764
    invalid_argument, 455
invocation
    macro, 394
INVOKED, 510, 511<br>iomanip>, 983<br>iostate, 952<br>ios_base, 953
    ios_base, 962
    ios_base::failure, 956
    ios_base::Init, 958<br>iosfw>, 948<br>iosfwd>, 948
istream
    ios_base, 956<br>istream>, 950<br>is<br>    ctype<char>, 640
ctype, 635
    is_bind_expression, 518
value, 518
    is_heap, 853
is_open
    basic_filebuf, 1029, 1041
    basic_ifstream, 1036
basic_ofstream, 1038
    is_placeholder, 518
value, 518
    is_sorted, 844
is_sorted_until, 845
isalnum, 629
isalpha, 629
iscntrl, 629
isctype
    regex_traits, 1058
    Regular Expression Traits, 1045, 1087
isdigit, 629
isfinite, 945
isgraph, 629
isgreater, 945
isgreaterequal, 945
isinf, 945
isless, 945
islessequal, 945
islessgreater, 945
islower, 629
ismn, 945
isnormal, 945
    <iso646.h>, 1168
isprint, 629
ispunct, 629
isspace, 629
    <istream>, 982
istream, 948, 982
    istream_iterator, 811
    operator!=, 813
    operator==, 813
    istreambuf_iterator, 814
    istreambuf_iterator, 816
    istreamstream, 948, 1013
    istringstream, 1180
    iostream_iterator, 1180
    isunordered, 945
isupper, 629
isxdigit, 629
iter_swap, 835
iteration-statement, 122, 124, 125
    <iterator>, 788
isword
    ios_base, 961
Draft
Jessie, 248
jump-statement, 124

key_eq
unordered associative containers, 697
key_equal
unordered associative containers, 696
key_type
unordered associative containers, 696
keyword, 1133
knuth_b, 891

label, 125
case, 119, 121
default, 119, 121
scope of, 38, 119

lambda()
exponential_distribution<>, 903

lattice; see DAG
subobject, 216

layout
bit-field, 210
class object, 202, 216
layout-compatible type, 69
left, 970
left shift
undefined, 110
left shift operator, 110
length
char_traits, 592, 594, 596–599, 601, 603–606, 608–610
codecvt, 643
sub_match, 1065
valarray, 926
length_error, 455, 585
length_error, 455
less, 515
less_equal, 516
lexical conventions, 15
lexicographical_compare, 856
lgamma, 943

library
C++ Standard, 403, 418, 420–422
Standard C, 403, 411, 414, 416, 1165, 1168, 1171
limits
implementation, 3

<limits>, 424
linear_congruential_engine<>, 881
constructor, 882
generation algorithm, 881
state, 881
template parameters, 881
textual representation, 881
transition algorithm, 881

linkage, 31, 55
external, 55, 416–418
implementation-defined object, 165
internal, 55

linkage-specification, 163
linkage specification, 163
extern, 163
implementation-defined, 163

list
operator, 21, 295

list, 702
list, 719

literal, 22, 83
base of integer, 23
char16_t, 24
char32_t, 24
character, 24
decimal, 23
decimal, 23
double, 26
double, 26
float, 26
floating point, 26
hexadecimal, 23
char, 25
implementation-defined value of multicharacter, 24
integer, 23
long, 23
long double, 26
multicharacter, 24
narrow-character, 24
octal, 23
type of character, 24
type of floating point, 26
type of integer, 23
unsigned, 23

literal, 22
llrint, 943
llround, 943
load_factor
unordered associative containers, 700
local_iterator, 696
unordered associative containers, 696
locale, 1043–1045, 1054, 1059, 1064, 1085
<locale>, 619
locale
locale, 626
local class
friend, 238
member function in, 204
scope of, 212
local variable
destruction of, 124, 125
lock
weak_ptr, 569
log, 929, 943
complex, 869
log10, 929, 943
complex, 869
log1p, 943
log2, 943
logb, 943
logic_error, 454
logic_error, 454
logical_and, 516
logical_not, 516
logical_or, 516
lognormal_distribution<>, 908
constructor, 908
m(), 908
probability density function, 908
s(), 909
long
typedef and, 131
long-long-suffix, 23
long-suffix, 23
longjmp, 452
lookup
argument-dependent, 45
member name, 218
name, 31, 41
template name, 333
lookup_classname
regex_traits, 1058
Regular Expression Traits, 1045, 1087
lower_bound, 845
lowercase, 411
lrint, 943
lround, 943
lvalue, 72, 1158
modifiable, 72
lvalue reference, 71, 173
m()
fisher_f_distribution<>, 911
lognormal_distribution<>, 908
macro
function-like, 393
masking, 420
object-like, 393
main(), 58
implementation-defined linkage of, 58
implementation-defined parameters to, 58
parameters to, 58
return from, 58, 60
make_heap, 852
make_pair, 484
make_tuple, 490
malloc, 572, 1169
<map>, 741
map, 744
operator<, 747
operator==, 747
mask_array, 935
match_any, 1053, 1055
match_continuous, 1053, 1055, 1081
match_default, 1053
match_flag_type, 1053, 1087
match_not_bol, 1053, 1055
match_not_bow, 1053, 1055
match_not_eol, 1053, 1055
match_not_eow, 1053, 1055
match_not_null, 1053, 1055, 1081
match_prev_avail, 1053, 1055, 1081
match_results, 1070, 1079, 1082
as sequence, 1070
begin, 1073
empty, 1073
end, 1073
Draft
format, 1074
match_results, 1072
matched, 1070
max_size, 1073
operator=, 1072
operator[], 1073
prefix, 1073
size, 1072
str, 1073
suffix, 1073
swap, 1074
matched, 1043
max, 854
    random_device, 892
    uniform random number generator requirement, 872
valarray, 926
xor_combine_engine<>, 890
max_bucket_count
    unordered associative containers, 699
max_element, 855
max_length
codecvt, 643
max_load_factor
    unordered associative containers, 700
max_size
    basic_string, 594
    array, 705
    match_results, 1073
mean
    normal_distribution<>, 907
    poisson_distribution<>, 902
mean()
    normal_distribution<>, 907
    poisson_distribution<>, 902
    student_t_distribution<>, 912
mem-initializer, 256
mem-initializer-id, 256
mem_fn, 523
mem_fun, 521, 522
mem_fun1_ref_t, 521
mem_fun1_t, 521
mem_fun_ref, 521, 522
mem_fun_ref_t, 521
mem_fun_t, 520
member, see also base class member
class static, 61
enumerator, 148
static, 206
template and static, 319
static, 99
member-declaration, 201
member-declarator, 201
member-specification, 201
member access operator
    overloaded, 298
member function
class, 203
    const, 206
    constructor and, 245
    destructor and, 251
friend, 237
    inline, 203
    local class, 213
    nested class, 241
    overload resolution and, 278
    static, 206, 207
    union, 208
    volatile, 206
member function call
    undefined, 204
member pointer to; see pointer to member, 71
member use
    static, 206, 207
memchr, 616
<memory>, 531
memory model, 6
memory management, see also new, delete
merge, 847
    list, 725
mersenne_twister_engine<>, 882
    constructor, 883
generation algorithm, 882
    state, 882
template parameters, 883
textual representation, 883
    transition algorithm, 882
message
diagnostic, 2
messages, 673
messages_byname, 674
min, 853
    random_device, 892
Draft
uniform random number generator requirement, 872
valarray, 926
min_element, 855
minmax, 854
minmax_element, 855
minstd_rand, 891
minstd_rand0, 890
minus, 514
mismatch, 832
mod, 943
modf, 943
modulo
  implementation defined, 109
modulus, 515
money_get, 666
money_put, 668
moneypunct, 670
moneypunct_byname, 672
move, 480
movemove, 834
move_backward, 834
move_iterator, 806
  move_iterator, 808
mt19937, 891
multimap, 749
  operator<, 752
  operator==, 752
multiple inheritance, 215, 216
  virtual and, 224
multiplicative-expression, 109
multiplies, 515
multiset, 757
  operator<, 760
  operator==, 760
mutable, 131

n()
  chi_squared_distribution<>, 909
  fisher_f_distribution<>, 911
name, 20, 31, 83
  address of cv-qualified, 99
  dependent, 339, 345
  elaborated enum, 143
  global, 39
  length of, 20
  macro, 394

overloaded function, 273
overloaded member, 202
point of declaration, 202
predefined macro, 36
qualified, 47
reserved, 417
scope of, 35
unqualified, 41

name
  locale, 627
  type_info, 443
namespace, 414, 1172
  global, 417
  unnamed, 150
namespaces, 148
  placeholders, 519
  regex_constants, 1052
name class, see class name
name hiding, 36, 40, 83, 84, 125
  class definition, 199
  function, 276
  overloading versus, 276
  user-defined conversion and, 247
name space
  label, 119
nan, 943
narrow
  basic_ios, 966
  ctype<char>, 640
  ctype, 636
NDEBUG, 416
nearbyint, 943
negate, 515
negative_binomial_distribution<>, 901
  constructor, 901
  discrete probability function, 901
  p(), 901
  t(), 901
nested-name-specifier, 84
nested class
  local class, 213
  scope of, 211
new, 419, 437
  operator, 438, 441, 572
  array of class objects and, 104
constructor and, 104
default constructor and, 104
default constructor and, 104
exception and, 104
initialization and, 104
operator, 418, 439–441
scoping and, 101
storage allocation, 101
type of, 253
unspecified constructor and, 105
unspecified order of evaluation, 105

new-declarator, 101
new-expression, 101
new-initializer, 101
new-placement, 101
new-type-id, 101
new_handler, 62, 442
next, 795
next_permutation, 856
nextafter, 943
nexttoward, 943
noboolalpha, 968
nonzero-digit, 20
none
bitset, 767
nonzero-digit, 22
norm, 870
complex, 868
normal distributions, 907–912
normal_distribution<>, 907
constructor, 907
mean, 907
mean(), 907
probability density function, 907
standard deviation, 907
stddev(), 908
noshowbase, 969
noshowpoint, 969
noshowpos, 969
noskipws, 969
not1, 517
not2, 517
not_equal_to, 515
notation
syntax, 5
nounitbuf, 970
nouppercase, 969

NTBS, 412, 1030, 1180, 1182
static, 412
NTCI6S, 412
static, 412
NTC32S, 412, 413
static, 413
NTCTS, 405
nth_element, 845
NTMBS, 412
static, 412
NTWCS, 413
static, 413
NULL, 424
num_get, 646
num_put, 651
number
hex, 25
octal, 25
<numeric>, 938
numeric_limits, 425
numeric_limits, 70
numpunct, 656
numpunct_byname, 658
object, 6, 31, 72
complete, 7
definition, 33
delete, 105
static, 60
destructor and placement of, 252
linkage specification, 165
local static, 61
undefined deleted, 63
unnamed, 245
object-expression, 82
object class, see also class object
object lifetime, 63
object temporary, see temporary
object type, 68
oct, 970
octal-digit, 22
octal-escape-sequence, 24
octal-literal, 22
offsetof, 1169
ofstream, 948, 1026
open

Draft
basic_filebuf, 1030, 1041
basic_ifstream, 1036
basic_ofstream, 1038, 1039
messages, 673
openmode
   ios_base, 958
operator, 296, 927, 928
   **=, 115
   +=, 100, 115
   -=, 115
   /=, 115
   operator==()
      random number engine requirement, 874
   %=, 115
   &=, 115
   ^=, 115
   <<=, 115
   >>=, 115
   |=, 115
   additive, 109
   address-of, 99
   assignment, 115, 413
   bitwise, 113
   bitwise AND, 113
   bitwise exclusive OR, 113
   bitwise inclusive OR, 113
   cast, 99, 107, 168
   class member access, 90
   comma, 116
   conditional expression, 114
   copy assignment, 263
   decrement, 91, 99, 100
   division, 109
   equality, 112
   function call, 88, 296
   greater than, 111
   greater than or equal to, 111
   increment, 91, 99, 100
   indirection, 99
   inequality, 112
   less than, 111
   less than or equal to, 111
   logical AND, 113
   logical negation, 99, 100
   logical OR, 113
   modulus, 109
   multiplication, 109
   multiplicative, 108
   one’s complement, 99, 100
   overloaded, 81
   pointer to member, 108
   pragma, 401
   precedence of, 8
   relational, 111
   scope resolution, 83, 84, 103, 204, 215, 225
   side effects and comma, 116
   side effects and logical AND, 113
   side effects and logical OR, 113
   sizeof, 99, 100
   subscripting, 88, 296
   unary, 99
   unary minus, 99, 100
   unary plus, 99, 100
operator
   overloaded, 296
operator delete, 103, see also delete, 106, 253
operator new, see also new, 103
operator!
   basic_ios, 967
   valarray, 924
operator!=, 479
   basic_string, 610
   complex, 867
   istreambuf_iterator, 817
   locale, 627
   reverse_iterator, 800
   type_info, 443
   bitset, 767
   queue, 728
   regex_iterator, 1081
   regex_token_iterator, 1085
   shared_ptr, 565
   stack, 732
   sub_match, 1065, 1066, 1068–1070
   tuple, 493
   valarray, 928
operator()
   locale, 627
   function, 528
   hash, 529
   reference_wrapper, 514

Draft
operator()()
    random number engine requirement, 873
    random_device, 893
    uniform random number generator requirement, 872
operator*
    auto_ptr, 1187
    back_insert_iterator, 802
    complex, 867
    front_insert_iterator, 804
    insert_iterator, 805
    istreambuf_iterator, 816
    ostreambuf_iterator, 818
    reverse_iterator, 798
    regex_iterator, 1081
    regex_token_iterator, 1085
    shared_ptr, 563
    valarray, 927
operator**=
    complex, 866
    gsl_slice_array, 935
    indirect_array, 938
    mask_array, 936
    slice_array, 932
    valarray, 925
operator+
    basic_string, 608
    complex, 866
    reverse_iterator, 799, 801
    valarray, 924
    valarray, 927
operator++
    back_insert_iterator, 803
    front_insert_iterator, 804
    insert_iterator, 806
    istreambuf_iterator, 816
    ostreambuf_iterator, 818
    reverse_iterator, 798
    regex_iterator, 1081, 1082
    regex_token_iterator, 1085
operator+=
    basic_string, 596
    complex, 865, 866
    gsl_slice_array, 935
    indirect_array, 938
    mask_array, 936
    reverse_iterator, 799
    slice_array, 932
    valarray, 925
operator--
    complex, 867
    reverse_iterator, 799, 801
    valarray, 924
    valarray, 927
operator->
    auto_ptr, 1187
    reverse_iterator, 798
    shared_ptr, 564
operator-function-id, 296
operator/>
    valarray, 927
operator/==
    complex, 866
    gsl_slice_array, 935
    indirect_array, 938
    mask_array, 936
    slice_array, 932
    valarray, 925
operator<
    basic_string, 610
    pair, 484
    reverse_iterator, 800
    queue, 728
    shared_ptr, 565
    stack, 733
    sub_match, 1066, 1068–1070
    tuple, 493
    valarray, 928
    weak_ptr, 569
operator<<
    basic_ostream, 1003, 1004
    basic_string, 612
    complex, 868
    bitset, 767, 768
    valarray, 927
Draft
INDEX 1236

operator<<
  sub_match, 1070
operator<<
  shared_ptr, 565
operator<<
  gslice_array, 935
  indirect_array, 938
  mask_array, 936
  slice_array, 932
  bitset, 764
  valarray, 925
operator<<, 479
  basic_string, 611
  reverse_iterator, 801
  queue, 728
  stack, 733
  sub_match, 1066–1070
  tuple, 493
  valarray, 928
operator=, 1186
auto_ptr, 1186
back_insert_iterator, 802
bad_alloc, 442
bad_cast, 444
bad_exception, 447
bad_typeid, 445
basic_string, 593
exception, 446
front_insert_iterator, 804
gslice_array, 935
indirect_array, 937
insert_iterator, 805
mask_array, 936
ostreambuf_iterator, 818
reverse_iterator, 797
slice_array, 931
valarray, 923
basic_regex, 1062, 1063
enable_shared_from_this, 570
function, 526, 527
match_results, 1072
shared_ptr, 562
tuple, 489, 490
weak_ptr, 568
operator==
  basic_string, 609
complex, 867
istreambuf_iterator, 817
locale, 627
pair, 483
reverse_iterator, 800
type_info, 443
bitset, 766
queue, 728
regex_iterator, 1081
regex_token_iterator, 1082, 1084
shared_ptr, 565
stack, 732
sub_match, 1065, 1066, 1068, 1069
tuple, 493
valarray, 928
operator==()
  random number engine requirement, 874
operator>, 479
  basic_string, 611
  reverse_iterator, 800
  queue, 728
  stack, 733
  sub_match, 1066, 1068–1070
  tuple, 493
  valarray, 928
operator>=, 479
  basic_string, 611
  reverse_iterator, 800
  queue, 728
  stack, 733
  sub_match, 1066, 1068–1070
  tuple, 493
  valarray, 928
operator>>
  basic_istream, 989
  basic_string, 612
  complex, 867
  istream, 988
  bitset, 767, 768
  valarray, 927
operator>>=
  gslice_array, 935
  indirect_array, 938
  mask_array, 936
  slice_array, 932
  bitset, 765

Draft
valarray, 925
operator[]
    basic_string, 595
    map, 748
    match_results, 1073
    reverse_iterator, 799
    unordered_map, 772
    valarray, 923, 924
operator%
    valarray, 927
operator%=
    gslice_array, 935
    indirect_array, 938
    mask_array, 936
    slice_array, 932
    valarray, 925
operator&
    bitset, 767
    valarray, 927
operator&=
    gslice_array, 935
    indirect_array, 938
    mask_array, 936
    slice_array, 932
    bitset, 764
    valarray, 925
operator&&
    valarray, 927, 928
operator~
    bitset, 768
    valarray, 927
operator~=
    gslice_array, 935
    indirect_array, 938
    mask_array, 936
    slice_array, 932
    bitset, 764
    valarray, 925
operator--
    reverse_iterator, 798
operator<<()
    random number engine requirement, 874
operator>>()
    random number engine requirement, 874
operator~
    valarray, 924
bitset, 765
operator|
    bitset, 768
    valarray, 927
operator|=
    gslice_array, 935
    indirect_array, 938
    mask_array, 936
    slice_array, 932
    bitset, 764
    valarray, 925
operator left shift, see left shift operator
operator overloading, see also overloaded operator
operator right shift; right shift operator, 110
operator shift, see left shift operator, right shift operator
operator use
    scope resolution, 208
optimization of temporary, see elimination of temporary
ordering
    function template partial, 331
order of execution
    base class constructor, 244
    base class destructor, 251
    constructor and static objects, 256
    constructor and array, 255
    destructor, 251
    destructor and array, 251
    member constructor, 244
    member destructor, 251
{oostream}, 982
ostream, 948, 983
ostream_iterator, 813
ostreambuf_iterator, 817
    ostreambuf_iterator, 818
ostreamstream, 948, 1013
ostrstream, 1181
    ostrstream, 1181
out
    codecvt, 642
out_of_range, 455, 585, 763–767
    out_of_range, 456
output_iterator_tag, 793, 794
over-aligned type, 74
overflow, 81
    undefined, 81
overflow
basic_filebuf, 1032
basic_streambuf, 981
basic_stringbuf, 1017
strstreambuf, 1177
overflow_error, 457, 763, 766
overflow_error, 457
overloaded function
address of, 99, 294
overloaded operator, 295
inheritance of, 296
overloading, 177, 199, 273, 330
example of, 273
overloads
floating point, 870
overload resolution contexts, 277
overrider
final, 222

p()
bernoulli_distribution, 899
binomial_distribution<>, 900
geometric_distribution<>, 900
negative_binomial_distribution<>, 901
pair, 481, 488–490
get, 484
tuple interface to, 481

param()
seed_seq, 895
parameter, 3
reference, 173
scope of, 38
void, 177
parameter-declaration, 177
parameterized type, see template
parameters
macro, 394
random number distribution, 876
parameter list
variable, 89, 177
parameter type list, 178
partial_sort, 843
partial_sort_copy, 844
partial_sum, 940
partition, 841
pbackfail
basic_filebuf, 1031

basic_streambuf, 981
basic_stringbuf, 1017
strstreambuf, 1178

pbase
basic_streambuf, 978
pbump
basic_streambuf, 978
pcount
ostrstream, 1182
strstreambuf, 1177
strstream, 1183
peek
basic_istream, 994
period, 411

phases
translation, 15
piecewise_constant_distribution<>, 914
constructor, 915
densities(), 915
interval boundaries, 914
intervals(), 915
probability density function, 914
weights, 915
placeholders, 519
placement syntax
new, 103

plus, 514
pm-expression, 108
POD class, 198
POD struct, 198
POD union, 198
POF, 452

pointer
zero, 79
pointer-literal, 29

void*, 71
pointer_to_binary_function, 520
pointer_to_unary_function, 520
pointer to member, 71, 108
Poisson distributions, 902–906
poisson_distribution<>, 902
constructor, 902
discrete probability function, 902
mean, 902
mean(), 902
polar, 870
complex, 868
pop
    priority_queue, 731
pop_back
    basic_string, 600
pop_heap, 852
POSIX
    extended regular expressions, 1054
    regular expressions, 1054
postfix ++ and --
    overloading, 298
postfix ++ and --, 91
pov, 870, 929, 943
    complex, 869
pp-number, 20
pptr
    basic_streambuf, 978
precision
    ios_base, 634, 959
prefix
    L, 25, 28
prefix
    match_results, 1073
prefix ++ and --
    overloading, 298
prefix ++ and dcr, 100
preprocessing, 389
preprocessing-op-or-punc, 22
preprocessing-token, 18
preprocessor
    macro, 389
prev, 795
prev_permutation, 857
primary equivalence class, 1043
priority_queue, 729
    priority_queue, 730
private, 229
probabilities()
    discrete_distribution<>, 914
probability density function, 876
    cauchy_distribution<>, 910
    chi_squared_distribution<>, 909
    exponential_distribution<>, 903
    extreme_value_distribution<>, 906
    fisher_f_distribution<>, 911
    gamma_distribution<>, 903
    general_pdf_distribution<>, 916
    lognormal_distribution<>, 908
    normal_distribution<>, 907
    piecewise_constant_distribution<>, 914
    student_t_distribution<>, 912
    uniform_real_distribution<>, 897
    weibull_distribution<>, 904
program, 55
    ill-formed, 2
    well-formed, 4
promotion
    floating point, 78
    integral, 78
protected, 229
protection, see access control, 421
proxy
    istreambuf_iterator, 815
pseudo-destructor-name, 90
pseudo-destructor-name, 87
ptr-operator, 168
ptr_fun, 520
ptrdiff_t
    implementation defined type of, 110
pubimbue
    basic_streambuf, 975
public, 229
pubseekoff
    basic_streambuf, 975
pubseekpos
    basic_streambuf, 976
pubsetbuf
    basic_streambuf, 976
pubsync
    basic_streambuf, 976
punctuators, 21
pure-specifier, 201
pure specifier, 201
push
    priority_queue, 730, 731
push_heap, 851
put
    basic_ostream, 1007
    money_put, 669
    num_put, 652
    time_put, 665
    put_money, 1011
INDEX

put_time, 1012
putback
  basic_istream, 995
pword
  ios_base, 961
qualification
  explicit, 47
qualified-id, 84
<queue>, 703
queue, 726

<random>, 878–881
random number distribution
  bernoulli_distribution, 898
  binomial_distribution<>, 899
  chi_squared_distribution<>, 909
  discrete_probability_function, 876
  discrete_distribution<>, 913
  exponential_distribution<>, 902
  extreme_value_distribution<>, 905
  fisher_f_distribution<>, 911
  gamma_distribution<>, 903
  general_pdf_distribution<>, 916
  geometric_distribution<>, 900
  lognormal_distribution<>, 908
  negative_binomial_distribution<>, 901
  normal_distribution<>, 907
  parameters, 876
  piecewise_constant_distribution<>, 914
  poisson_distribution<>, 902
  probability_density_function, 876
  requirements, 876–878
  student_t_distribution<>, 912
  uniform_int_distribution<>, 896
  uniform_real_distribution<>, 897
random number distributions
  Bernoulli, 898–901
  normal, 907–912
  Poisson, 902–906
  sampling, 913–917
  uniform, 896–898
random number engine
  generation algorithm, 873
  linear_congruential_engine<>, 881
  mersenne_twister_engine<>, 882
  requirements, 872–875
  state, 873
  subtract_with_carry_engine<>, 883
  successor state, 873
  transition algorithm, 873
  with predefined parameters, 890–891
random number engine adaptor
  discard_block_engine<>, 885
  independent_bits_engine<>, 886
  requirements, 875
  shuffle_order_engine<>, 888
  with predefined parameters, 890–891
  xor_combine_engine<>, 889
random number generation, 871–917
random number generator, see uniform random number generator
random_access_iterator_tag, 793, 794
random_device, 892
  constructor, 892
  entropy(), 892
  implementation leeway, 892
  max, 892
  min, 892
  operator()(), 893
random_shuffle, 840
range_error, 456
  range_error, 457
ranlux24, 891
ranlux24_base, 891
ranlux48, 891
ranlux48_base, 891
rbegin
  basic_string, 594
rdbuf
  basic_filebuf, 1041
  basic_ifstream, 1036
  basic_ios, 966
  basic_ostream, 1038
  basic_ofstream, 1038
  basic_ostream, 1023
  basic_stringstream, 1025
  istrstream, 1181
  ostringstream, 1182
  istringstream, 1183
rdstate
  basic_ios, 967
Draft
read
  basic_istream, 994
readsome
  basic_istream, 994
real, 870
  complex, 868
realloc, 572
redefinition
  typedef, 134
reentrancy, 421
ref
  reference_wrapper, 514
reference, 71
    assignment to, 116
    call by, 89
    const, 195
    direct binding of, 194
    lvalue, 71
    null, 174
    rvalue, 71
    sizeof, 100
reference-compatible, 194
reference-related, 194
reference_wrapper, 512
  cref, 514
  get, 513
  operator(), 514
  ref, 514
  reference_wrapper, 513
<regex>, 1046
regex, 1046
regex_constants, 1052
  error_type, 1055–1057
  match_flag_type, 1053
  syntax_option_type, 1053
regex_error, 1056, 1059, 1087
regex_iterator, 1079
  end-of-sequence, 1080
  increment, 1081
  operator!=, 1081
  operator*, 1081
  operator++, 1081, 1082
  operator==, 1081
  regex_iterator, 1080
regex_match, 1075–1077
regex_replace, 1079
regex_search, 1077, 1078
regex_token_iterator, 1082
  end-of-sequence, 1082
  operator!=, 1085
  operator*, 1085
  operator++, 1085
  operator==, 1082, 1084
  regex_token_iterator, 1084
regex_traits, 1057
  char_class_type, 1057
  isctype, 1058
  lookup_classname, 1058
  lookup_collatename, 1058
  specializations, 1057
  transform, 1058
  transform_primary, 1058
  translate, 1057
  translate_nocase, 1058
region
  declarative, 31, 35
register, 131
register_callback
  ios_base, 961
regular expression, 1043–1088
  grammar, 1085
  matched, 1043
  requirements, 1044
Regular Expression Traits, 1085
  char_class_type, 1044
  isctype, 1045, 1087
  lookup_classname, 1045, 1087
  lookup_collatename, 1045, 1087
  requirements, 1044, 1057
  transform, 1045, 1087
  transform_primary, 1045, 1087
  translate, 1044, 1087
  translate_nocase, 1044, 1087
rehash
  unordered associative containers, 700
rel_ops, 478
relational-expression, 111
release
  auto_ptr, 1187
remainder, 943
remainder operator, see modulus operator
remove, 838
list, 725
remove_copy, 838
remove_copy_if, 838
remove_if, 838
remquo, 943
rend
  basic_string, 594
replace, 836
  basic_string, 600
replace_copy, 836
replace_copy_if, 836
replace_if, 836
replacement
  macro, 393
representation
  object, 67
  value, 67
requirements, 408
  Allocator, 475
  ConstructibleAsElement, 682
Container, 696, 705, 706
  not required for unordered associated containers, 695
  not supported by unordered associated containers, 700
container, 681, 1070
CopyAssignable, 475
CopyConstructible, 474
Destructible, 475
EqualityComparable, 474
iterator, 783
LessThanComparable, 474
MoveAssignable, 474
MoveConstructible, 474
numeric type, 859
random number distribution, 876–878
random number engine, 872–875
random number engine adaptor, 875
Regular Expression Traits, 1044, 1057
sequence, 1070
Swappable, 475
uniform random number generator, 872
Unordered Associative Container, 696
reraise, 382
rescanning and replacement, 396
reserve
  basic_string, 595
  vector, 737
reserved identifier, 21
reset
  auto_ptr, 1187
  bitset, 765
  shared_ptr, 563
  weak_ptr, 568
resetiosflags, 1008
resize
  basic_string, 595
deqe, 711
list, 723
vector, 737, 738
valarray, 926
resolution
  argument matching, see overload
function template overload, 377
overload, 277
overloaded function call resolution, see also argument
  matching, overload
resolution overloading, see overload
  scoping ambiguity, 219
template name, 333
template overload, 331
restriction, 420, 422
  static member local class, 208
address of bit-field, 210
anonymous union, 209
bit-field, 210
constructor, 244
copy assignment operator, 267
copy constructor, 265
destructor, 250
extern, 132
local class, 213
overloading, 296
pointer to bit-field, 210
reference, 174
register, 131
static, 132
union, 208, 209
restrictions
  operator overloading, 296
result_of, 511
type, 512
Draft
result_type
    entity characterization based on, 871
    uniform random number generator requirement, 872
    xor_combine_engine<>, 890
rethrow, 382
return, 124, 125
    constructor and, 125
    reference and, 194
return statement, see also return
return type, 178
    overloading and, 273
reverse, 839
    list, 725
reverse_copy, 839
reverse_iterator, 795
    reverse_iterator, 797
rfind
    basic_string, 604
right, 970
right shift
    implementation defined, 111
right shift operator, 110
rint, 943
rotate, 840
rotate_copy, 840
round, 943
rounding, 79
rule
    as-if, 7
    one-definition, 33
runtime_error, 456
    runtime_error, 456
rvalue, 72
    lvalue conversion to, 76
    lvalue conversion to, 1158
rvalue reference, 71, 173
s()
    lognormal_distribution<>, 909
s-char, 27
s-char-sequence, 26
sampling distributions, 913–917
sbumpc
    basic_streambuf, 976
scalar type, 68
scalbn, 943
scalbn, 943
scan_is
    ctype<char>, 640
cstype, 635
scan_not
    ctype<char>, 640
cstype, 636
scientific, 971
scope, 31, 35
    anonymous union at namespace, 209
class, 39
destructor and exit from, 124
enumeration, 40
exception declaration, 38
function, 38
global, 39
global namespace, 39
iteration-statement, 122
local, 37
macro definition, 396
namespace, 38
overloading and, 276
potential, 35
selection-statement, 120
scope resolution operator, 47
search, 833
seed()
    random number engine requirement, 873
seed_seq
    constructor, 893, 894
generate(), 894
overview, 893
param(), 895
size(), 895
seekdir
    ios_base, 958
seekg
    basic_istream, 995
seekoff
    basic_filebuf, 1032
    basic_streambuf, 979
    basic_stringbuf, 1017
    strstreambuf, 1178
seekp
    basic_ostream, 1002
seekpos
Draft
side effect, 10
side effects, 10
sign, 26
signature, 3
signbit, 945
signed
  typedef and, 131
signed integer type, 69
simple call wrapper, 511
simple-escape-sequence, 24
template-id, 308
simple-type-specifier, 141
sin, 929, 943
  complex, 870
sinh, 929, 943
  complex, 870
size
  basic_string, 594
gslice, 934
slice, 930
array, 705, 706
bitset, 766
match_results, 1072
size()
  seed_seq, 895
size_t, 101
sizeof
  empty class, 197
skipws, 969
slice, 930
  slice, 930
slice_array, 931
smart pointers, 558–571
snputc
  basic_streambuf, 976
sort, 843
  list, 726
sort_heap, 852
space
  white, 18
specialization
  class template, 309
class template partial, 324
template, 346
template explicit, 353
special member function, see constructor, destructor, inline
  function, user-defined conversion, virtual function
specification
  template argument, 360
specifications
  C++ Standard Library exception, 422
  implementation-defined exception, 422
  Standard C library exception, 422
specifier
  declaration, 130
  explicit, 134
  friend, 421
  friend, 136
  function, 133
  inline, 133
  missing storage class, 132
  static, 131
  storage class, 131
typedef, 134
  virtual, 134
specifier access, see access specifier
specifier type, see type specifier
splice
  list, 724
sputbackc
  basic_streambuf, 976
sputc
  basic_streambuf, 977
sputn
  basic_streambuf, 977
sqrt, 929, 943
  complex, 870
<sstream>, 1013
stable algorithm, 406
stable_partition, 841
stable_sort, 843
<stack>, 704
stack, 731
standard
  structure of, 5
  standard deviation
    normal_distribution<>, 907
standard-layout class, 198
standard-layout struct, 198
standard-layout union, 198

Draft
standard integer type, 70
standard signed integer type, 69
standard unsigned integer type, 69
start
  program, 58
start
  gslice, 934
  slice, 930
startup
  program, 416, 419
state
  discard_block_engine<>, 885
  independent_bits_engine<>, 886
  linear_congruential_engine<>, 881
  mersenne_twister_engine<>, 882
  object, 405
  random number engine, 873
  shuffle_order_engine<>, 888
  subtract_with_carry_engine<>, 884
  xor_combine_engine<>, 889
state
  fpos, 962
statement, 119
  continue in for, 124
  break, 124
  compound, 119
  continue, 124, 125
  declaration, 125
  declaration in for, 124
  declaration in switch, 122
  do, 122, 123
  empty, 119
  expression, 119
  for, 122, 123
  goto, 119, 124, 125
  if, 120, 121
  iteration, 122
  jump, 124
  labeled, 119
  null, 119
  selection, 120
  switch, 120, 121, 124
  while, 122
statement, 119
static, 131
  destruction of local, 126
  linkage of, 55, 132
  overloading and, 273
static_assert, 129, 130
static_pointer_cast
  shared_ptr, 566
<stdio.h>, 24, 28
stddev()
  normal_distribution<>, 908
<stdexcept>, 453
<stdint.h>, 435
<stdlib.h>, 1172
storage class, 31
storage duration, 60
  automatic, 61
  class member, 63
  dynamic, 61, 101
  local object, 61
  register, 61
storage management, see new, delete
str
  basic_istringstream, 1021
  basic_ostringstream, 1023
  basic_stringbuf, 1016
  basic_stringstream, 1025
  istrstream, 1181
  ostrstream, 1182
  strstreambuf, 1177
  strstream, 1183
  match_results, 1073
  sub_match, 1065
strchr, 615
stream
  arbitrary-positional, 404
  repositioned, 406
<stdio.h>, 971
sstreambuf, 948, 971
  implementation-defined, 948
streamoff, 953, 963, 1172
  implementation-defined, 953, 1172
streampos
  implementation-defined, 1172
streamsize, 953
strftime, 665
stride
  gslice, 934
  slice, 930
string
  distinct, 28
  null-terminated byte, 412
  null-terminated char16-character, 412
  null-terminated char32-character, 412
  null-terminated character type, 405
  null-terminated multibyte, 412
  null-terminated wide-character, 413
  sizeof, 28
  type of, 27
<string>, 581
string-literal, 26
stringbuf, 948, 1013
stringstream, 948
string literal, 27
  char16_t, 27, 28
  char32_t, 27, 28
  implementation-defined, 28
  narrow, 27
  type of, 28
  undefined change to, 28
  wide, 27, 28
strlen, 1176
strlen, 1176, 1182
strpbrk, 615
strrchr, 616
strstr, 616
strstream, 1182
  strstream, 1183
strstreambuf, 1174
  strstreambuf, 1175
struct
  standard-layout, 198
struct
  class versus, 197
structure, 197
structure tag, see class name
student_t_distribution<>, 912
  constructor, 912
  mean(), 912
  probability density function, 912
sub-expression, 1044
sub_match, 1064
  compare, 1065
  length, 1065
  operator!=, 1065, 1066, 1068–1070
  operator<, 1066, 1068–1070
  operator<<, 1070
  operator<, 1066–1070
  operator==, 1065, 1066, 1068, 1069
  operator>, 1066, 1068–1070
  operator>=, 1066, 1068–1070
  str, 1065
subobject, 7
subscripting operator
  overloaded, 298
subsequence rule
  overloading, 291
substr
  basic_string, 606
subtract_with_carry_engine<>
  carry, 884
  constructor, 884, 885
  generation algorithm, 884
  state, 884
  template parameters, 884
  textual representation, 884
  transition algorithm, 884
subtraction
  implementation defined pointer, 110
subtraction operator, 109
successor state
  random number engine, 873
suffix
  E, 26
  F, 26
  f, 26
  L, 23, 26
  l, 23, 26
  U, 23
  u, 23
suffix
  match_results, 1073
sum
  valarray, 926
summary
  compatibility with ISO C, 1155
  scope rules, 41
summary, syntax, 1133
sungetc
  basic_streambuf, 976
swap, 835

Draft
basic_string, 602, 612
pair, 484
vector, 737
array, 706
basic_regex, 1064
function, 527, 529
match_results, 1074
shared_ptr, 563, 565
unordered_map, 773
unordered_multimap, 776
unordered_multiset, 782
unordered_set, 779
weak_ptr, 568, 569
swap_ranges, 835
sync
  basic_filebuf, 1033
  basic_istream, 995
  basic_streambuf, 979
sync_with_stdio
  ios_base, 960
synonym, 152
  type name as, 134
syntax
  class member, 90
syntax_option_type, 1053
  awk, 1053, 1054
  basic, 1053, 1054
  collate, 1053, 1054, 1087
  ECMAScript, 1053, 1054
  egrep, 1053, 1054
  extended, 1053, 1054
  grep, 1053, 1054
  icase, 1053, 1054
  nosubs, 1053, 1054
  optimize, 1053, 1054
t()
  binomial_distribution<>, 900
  negative_binomial_distribution<>, 901
table
  ctype<char>, 640
tan, 929, 943
  complex, 870
tanh, 929, 943
  complex, 870
target
  function, 528
  target object, 510, 511
  target_type
    function, 528
tellg
  basic_istream, 995
tellp
  basic_ostream, 1002
template, 303
  definition of, 303
  function, 359
  member function, 318
  primary, 324
template, 303
  template-argument, 308
  template-argument-list, 308
  template-declaration, 303
  template-id, 308
  template-name, 308
  template-parameter, 304
  template-parameter-list, 303
template name
  linkage of, 303
temporary, 245
  constructor for, 246
  destruction of, 246
  destructor for, 246
  elimination of, 245, 268
  implementation-defined generation of, 245
  order of destruction of, 246
terminate, 60, 437, 447, 448
terminate(), 387, 388
terminate_handler, 419, 448
termination
  program, 58, 60
terminology
  pointer, 71
test
  bitset, 767
textual representation
  discard_block_engine<>, 886
  independent_bits_engine<>, 887
  shuffle_order_engine<>, 889
  subtract_with_carry_engine<>, 884
  xor_combine_engine<>, 890
tgamma, 943
Draft
<tgmath.h>, 942
this, 83

type of, 206
this pointer, see this
thousands_sep
numpunct, 657
thread, blocked, 404
twow, 379
throw-expression, 379
throwing

to_string
bitset, 766
to_ullong
bitset, 766
to_ulong
bitset, 766
token, 22
token, 19
tolower, 629
cctype<char>, 640
cctype, 636
toupper, 629
cctype<char>, 640
cctype, 636
traits, 406
transform, 835
collate, 659
regex_traits, 1058
Regular Expression Traits, 1045, 1087
transform_primary
Regular Expression Traits, 1045, 1087, 1088
transform_primary
regex_traits, 1058
transition algorithm

discard_block_engine<>, 885
independent_bits_engine<>, 886
linear_congruential_engine<>, 881
mersenne_twister_engine<>, 882
random number engine, 873
shuffle_order_engine<>, 888
subtract_with_carry_engine<>, 884
xor_combine_engine<>, 889
translate
regex_traits, 1057
Regular Expression Traits, 1044, 1087
translate_nocase
regex_traits, 1058
Regular Expression Traits, 1044, 1087
translation
separate, 15
translation unit, 15, 55
name and, 31
trigraph, 15
trivial class, 197
trivial class type, 104
trivial type, 104
truename
numpunct, 657
trunc, 943
truncation, 79
try, 379
try-block, 379
</tuple>, 485
tuple, 485, 486, 707
and array, 707
and pair, 481
get, 492
make_tuple, 490
operator! =, 493
operator<, 493
operator<=, 493
operator==, 493
operator>, 493
operator>=, 493
tie, 491
tuple, 487, 488
tuple_element, 484, 492, 707
tuple_size, 484, 492, 707
type, 31

arithmetic, 70
array, 71, 178
bitmask, 410, 411
Boolean, 69
char, 69

Draft
char16_t, 70
char32_t, 70
character, 69
character container, 404
class and, 197
compound, 71
const, 138
destination, 189
double, 70
dynamic, 2
enumerated, 71, 410
equation underlying, 147
example of incomplete, 68
extended integer, 70
extended signed integer, 69
extended unsigned integer, 69
float, 70
floating point, 69
function, 71, 176–178
fundamental, 69
sizeof, 69
int, 69
integral, 69
long, 69
long double, 70
long long, 69
multi-level mixed pointer and pointer to member, 77
multi-level pointer to member, 77
object, 7
over-aligned, 74
POD, 68
pointer, 71
polymorphic, 222
short, 69
signed char, 69
signed integer, 69
standard integer, 70
standard signed integer, 69
standard unsigned integer, 69
static, 3
underlying wchar_t, 70
unsigned, 69
unsigned char, 69
unsigned int, 69
unsigned long, 69
unsigned long long, 69
unsigned short, 69
unsigned integer, 69
void, 70
volatile, 138
wchar_t, 70
type
result_of, 512
type-id, 168
type-id-list, 385
type-name, 141
type-parameter, 304
type-specifier
bool, 141
wchar_t, 141
type-specifier, 138
type-specifier-seq, 138
type_info, 93, 443
type_info::name
implementation-defined, 443
typedef
function, 178
typedef
overloading and, 274
typedef-name, 134
typeid, 93
<typename>, 442
typename, 143
types
implementation-defined, 410
implementation-defined exception, 422
type checking
argument, 89
type conversion, explicit, see casting
type generator, see template
type name, 168
nested, 213
scope of nested, 213
type pun, 97
type specifier
char, 141
class, 197
double, 141
enum, 143
float, 141
int, 141
long, 141
short, 141
struct, 197
union, 197
unsigned, 141
void, 141
volatile, 140

uflow
  basic_filebuf, 1031
  basic_streambuf, 980
uint16_t, 434
uint32_t, 434
uint64_t, 434
uint8_t, 434
uint_fast16_t, 434
uint_fast32_t, 434
uint_fast64_t, 434
uint_fast8_t, 434
uint_least16_t, 434
uint_least32_t, 434
uint_least64_t, 434
uint_least8_t, 434
uintmax_t, 434
uintptr_t, 434

unary function, 512, 513, 523
unary_expression, 99
unary_operator, 99
unary_function, 511, 529
unary_negate, 517

unary operator
  interpretation of, 297
  overloaded, 297

uncaught_exception, 448
undefined, 78, 406, 417, 418, 420, 596, 922, 924–928, 933, 938, 963

undefined behavior, 815

underflow
  basic_filebuf, 1031
  basic_streambuf, 980
  basic_stringbuf, 1017
  strstreambuf, 1178
underflow_error
  underflow_error, 458
unexpected, 447

unexpected(), 388
unexpected_handler, 419, 447
unget
  basic_istream, 995
uniform distributions, 896–898
uniform random number generator
  requirements, 872
uniform_int_distribution<>, 896
  a(), 897
  b(), 897
  constructor, 897
discrete probability function, 896
uniform_real_distribution<>, 897
  a(), 898
  b(), 898
  constructor, 898
  probability density function, 897
uninitialized_copy, 545
uninitialized_fill, 546
uninitialized_fill_n, 546
union
  standard-layout, 198
union, 71, 208
  class versus, 197
  anonymous, 209
  global anonymous, 209
unique, 838
  list, 725
  shared_ptr, 564
unique_copy, 839
unit
  instantiation, 16
  translation, 416, 417
unitbuf, 970
ununiversal-character-name, 17
unordered associative containers, 695–782
  begin, 700
  bucket, 699
  bucket_count, 699
  bucket_size, 700
  clear, 699
  complexity, 695
  const_local_iterator, 696
  count, 699
  end, 700
  equal_range, 699
Draft
equality function, 695
equivalent keys, 695, 696, 773, 779
erase, 699
exception safety, 701
find, 699
hash function, 695
hash_function, 697
hasher, 696
insert, 698, 699
iterator invalidation, 700
iterators, 700
key_eq, 697
key_equal, 696
key_type, 696
lack of comparison operators, 695, 700
load_factor, 700
local_iterator, 696
max_bucket_count, 699
max_load_factor, 700
rehash, 700
requirements, 695, 696, 700, 701
unique keys, 695, 696, 769, 776
unordered_map, 768
unordered_map, 768–770
element access, 772
operator[], 772
swap, 773
unique keys, 769
unordered_map, 772
unordered_multimap, 768, 773
equivalent keys, 773
swap, 776
unordered_multimap, 775
unordered_multiset, 769, 779
equivalent keys, 779
swap, 782
unordered_multiset, 782
unordered_set, 769, 776
swap, 779
unique keys, 776
unordered_set, 778, 779
unqualified-id, 83
unsetf
ios_base, 959
unshift
codecv, 642
unsigned
typedef and, 131
unsigned-suffix, 23
unsigned integer type, 69
unspecified, 438, 439, 443, 844, 1017, 1176, 1178
unspecified behavior, 926
unwinding
stack, 382
upper_bound, 846
uppercase, 411, 417
uppercase, 969
use_count
shared_ptr, 564
weak_ptr, 569
use_facet
locale, 628
using-declaration, 153
using-directive, 159
<utility>, 478
va_end, 418
va_list, 418
<valarray>, 917
valarray, 920, 934
valarray, 922
value
call by, 89
null member pointer, 79
null pointer, 79
undefined unrepresentable integral, 79
value
is_bind_expression, 518
is_placeholder, 518
value-initialization, 187
variable
indeterminate uninitialized, 187
<vector>, 704
vector, 733
operator<, 736
operator==, 736
vector, 736
vector<bool>, 739
virtual base class, 217
virtual function, 222
pure, 226, 227
virtual function call, 225
  constructor and, 262
destructor and, 262
  undefined pure, 227
visibility, 41
void*
  type, 71
void&
  173
volatile
  constructor and, 206, 244
destructor and, 206, 250
  implementation-defined, 140
  overloading and, 275
wcserr, 952
wchar_t, 24, 28, 413, 615
  implementation-defined, 70
wcin, 951
wclog, 952
wcout, 951
wcschr, 616
wcspbrk, 616
wcsrchr, 616
wcsstr, 616
weak result type, 511
weak_ptr, 561, 567
  ~weak_ptr, 568
  expired, 569
  lock, 569
  operator<, 569
  operator=, 568
  reset, 568
  swap, 568, 569
  use_count, 569
weak_ptr, 567, 568
weibull_distribution<>, 904
  a(), 905
  b(), 905
  constructor, 905
  probability density function, 904
weibull_distribution<>, 905
weights
  discrete_distribution<>, 914
  piecewise_constant_distribution<>, 915
wfilebuf, 948, 1026
wifstream, 948
what
  bad_alloc, 442
  bad_cast, 444
  bad_exception, 447
  bad_typeid, 445
  exception, 446
  ios_base::failure, 956
  bad_weak_ptr, 558
white space, 19
wide-character, 24
widened
  basic_ios, 966
cctype<char>, 640
cctype, 636
width
  ios_base, 634, 959
wifstream, 948, 1026
wios, 953
wiostream, 948, 982
wstringstream, 948, 1013
wstringstream, 948
wregex, 1046
write
  basic_ostream, 1007
ws, 996
ws, 989
wostreambuf, 948, 971
wstreambuf, 948, 1013
wstringstream, 948
X(X&), see copy constructor
xalloc
  ios_base, 961
xmax()
  general_pdf_distribution<>, 917
xmin()
  general_pdf_distribution<>, 917
xor_combine_engine<>, 889
generation algorithm, 889
  max, 890
  result_type, 890
  state, 889
template parameters, 890
textual representation, 890
transition algorithm, 889
xsgetn
  basic_streambuf, 979
xsputn
  basic_streambuf, 981
zero
  undefined division by, 81, 109
  undefined modulus, 81
zero-initialization, 187