Programming Languages — C++
Langages de programmation — C++

Warning
This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.
Copyright notice

This ISO document is a working draft or committee draft and is copyright-protected by ISO. While the reproduction of working drafts or committee drafts in any form for use by participants in the ISO standards development process is permitted without prior permission from ISO, neither this document nor any extract from it may be reproduced, stored or transmitted in any form for any other purpose without prior written permission from ISO.

Requests for permission to reproduce this document for the purpose of selling it should be addressed as shown below or to ISO’s member body in the country of the requestor.

ISO copyright office  
Case postale 56, CH-1211 Geneva 20  
Tel. + 41 22 749 01 11  
Fax + 41 22 749 09 47  
E-mail copyright@iso.org  
Web www.iso.org

Reproduction for sales purposes may be subject to royalty payments or a licensing agreement.

Violators may be prosecuted.
Contents

List of Tables xi
List of Figures xv

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 5
1.8 The C++ object model 6
1.9 Program execution 7
1.10 Multi-threaded executions and data races 10
1.11 Acknowledgments 14

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

3 Basic concepts
3.1 Declarations and definitions 31
3.2 One definition rule 33
3.3 Scope 35
3.4 Name lookup 41
3.5 Program and linkage 55
3.6 Start and termination 58
3.7 Storage duration 61
3.8 Object lifetime 65
3.9 Types 69
3.10 Lvalues and rvalues 74

CONTENTS
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3</td>
<td>Namespaces</td>
<td>153</td>
</tr>
<tr>
<td>7.4</td>
<td>The <code>asm</code> declaration</td>
<td>166</td>
</tr>
<tr>
<td>7.5</td>
<td>Linkage specifications</td>
<td>167</td>
</tr>
<tr>
<td>7.6</td>
<td>Attributes</td>
<td>170</td>
</tr>
<tr>
<td><strong>8</strong></td>
<td>Declarators</td>
<td><strong>176</strong></td>
</tr>
<tr>
<td>8.1</td>
<td>Type names</td>
<td>177</td>
</tr>
<tr>
<td>8.2</td>
<td>Ambiguity resolution</td>
<td>178</td>
</tr>
<tr>
<td>8.3</td>
<td>Meaning of declarators</td>
<td>179</td>
</tr>
<tr>
<td>8.4</td>
<td>Function definitions</td>
<td>192</td>
</tr>
<tr>
<td>8.5</td>
<td>Initializers</td>
<td>196</td>
</tr>
<tr>
<td><strong>9</strong></td>
<td>Classes</td>
<td><strong>210</strong></td>
</tr>
<tr>
<td>9.1</td>
<td>Class names</td>
<td>212</td>
</tr>
<tr>
<td>9.2</td>
<td>Class members</td>
<td>214</td>
</tr>
<tr>
<td>9.3</td>
<td>Member functions</td>
<td>216</td>
</tr>
<tr>
<td>9.4</td>
<td>Static members</td>
<td>219</td>
</tr>
<tr>
<td>9.5</td>
<td>Unions</td>
<td>221</td>
</tr>
<tr>
<td>9.6</td>
<td>Bit-fields</td>
<td>222</td>
</tr>
<tr>
<td>9.7</td>
<td>Nested class declarations</td>
<td>223</td>
</tr>
<tr>
<td>9.8</td>
<td>Local class declarations</td>
<td>224</td>
</tr>
<tr>
<td>9.9</td>
<td>Nested type names</td>
<td>225</td>
</tr>
<tr>
<td><strong>10</strong></td>
<td>Derived classes</td>
<td><strong>226</strong></td>
</tr>
<tr>
<td>10.1</td>
<td>Multiple base classes</td>
<td>227</td>
</tr>
<tr>
<td>10.2</td>
<td>Member name lookup</td>
<td>229</td>
</tr>
<tr>
<td>10.3</td>
<td>Virtual functions</td>
<td>233</td>
</tr>
<tr>
<td>10.4</td>
<td>Abstract classes</td>
<td>236</td>
</tr>
<tr>
<td><strong>11</strong></td>
<td>Member access control</td>
<td><strong>239</strong></td>
</tr>
<tr>
<td>11.1</td>
<td>Access specifiers</td>
<td>241</td>
</tr>
<tr>
<td>11.2</td>
<td>Accessibility of base classes and base class members</td>
<td>242</td>
</tr>
<tr>
<td>11.3</td>
<td>Access declarations</td>
<td>244</td>
</tr>
<tr>
<td>11.4</td>
<td>Friends</td>
<td>245</td>
</tr>
<tr>
<td>11.5</td>
<td>Protected member access</td>
<td>248</td>
</tr>
<tr>
<td>11.6</td>
<td>Access to virtual functions</td>
<td>249</td>
</tr>
<tr>
<td>11.7</td>
<td>Multiple access</td>
<td>250</td>
</tr>
<tr>
<td>11.8</td>
<td>Nested classes</td>
<td>250</td>
</tr>
<tr>
<td><strong>12</strong></td>
<td>Special member functions</td>
<td><strong>251</strong></td>
</tr>
<tr>
<td>12.1</td>
<td>Constructors</td>
<td>251</td>
</tr>
<tr>
<td>12.2</td>
<td>Temporary objects</td>
<td>253</td>
</tr>
<tr>
<td>12.3</td>
<td>Conversions</td>
<td>255</td>
</tr>
<tr>
<td>12.4</td>
<td>Destructors</td>
<td>258</td>
</tr>
<tr>
<td>12.5</td>
<td>Free store</td>
<td>261</td>
</tr>
<tr>
<td>12.6</td>
<td>Initialization</td>
<td>263</td>
</tr>
<tr>
<td>12.7</td>
<td>Construction and destruction</td>
<td>269</td>
</tr>
<tr>
<td>12.8</td>
<td>Copying and moving class objects</td>
<td>271</td>
</tr>
<tr>
<td>12.9</td>
<td>Inheriting Constructors</td>
<td>279</td>
</tr>
<tr>
<td><strong>13</strong></td>
<td>Overloading</td>
<td><strong>283</strong></td>
</tr>
</tbody>
</table>

CONTENTS
## Index

13.1 Overloadable declarations .......................................................... 283
13.2 Declaration matching ................................................................. 285
13.3 Overload resolution ..................................................................... 286
13.4 Address of overloaded function .................................................... 305
13.5 Overloaded operators .................................................................. 306
13.6 Built-in operators ........................................................................ 311

14 Templates ....................................................................................... 315
14.1 Template parameters ................................................................. 316
14.2 Names of template specializations ................................................. 319
14.3 Template arguments .................................................................... 320
14.4 Type equivalence ......................................................................... 326
14.5 Template declarations ............................................................... 327
14.6 Name resolution .......................................................................... 343
14.7 Template instantiation and specialization ...................................... 356
14.8 Function template specializations .................................................. 367

15 Exception handling ........................................................................ 387
15.1 Throwing an exception .................................................................. 388
15.2 Constructors and destructors ....................................................... 390
15.3 Handling an exception ................................................................... 390
15.4 Exception specifications ............................................................... 392
15.5 Special functions ......................................................................... 395

16 Preprocessing directives ................................................................. 397
16.1 Conditional inclusion .................................................................... 399
16.2 Source file inclusion ..................................................................... 400
16.3 Macro replacement ....................................................................... 401
16.4 Line control .................................................................................. 406
16.5 Error directive .............................................................................. 407
16.6 Pragma directive .......................................................................... 407
16.7 Null directive ............................................................................... 407
16.8 Predefined macro names ............................................................... 407
16.9 Pragma operator .......................................................................... 408

17 Library introduction ........................................................................ 410
17.1 General ......................................................................................... 410
17.2 The C standard library .................................................................... 411
17.3 Definitions ..................................................................................... 411
17.4 Additional definitions .................................................................... 414
17.5 Method of description (Informative) ................................................ 414
17.6 Library-wide requirements ............................................................. 421

18 Language support library ................................................................. 432
18.1 General ......................................................................................... 432
18.2 Types ............................................................................................. 432
18.3 Implementation properties ............................................................ 433
18.4 Integer types .................................................................................. 442
18.5 Start and termination ................................................................. 443
18.6 Dynamic memory management .................................................... 445
18.7 Type identification ........................................................................ 450

CONTENTS vi
29.2 Header <atomic> synopsis ....................................................... 1107
29.3 Order and Consistency ............................................................ 1110
29.4 Lock-free Property .................................................................. 1112
29.5 Atomic Types ........................................................................... 1113
29.6 Operations on Atomic Types ...................................................... 1122
29.7 Flag Type and Operations ........................................................... 1126
29.8 Fences ..................................................................................... 1127

30 Thread support library ................................................................ 1129
30.1 General .................................................................................... 1129
30.2 Requirements .......................................................................... 1129
30.3 Threads .................................................................................... 1130
30.4 Mutual exclusion ...................................................................... 1136
30.5 Condition variables ................................................................... 1150
30.6 Futures ..................................................................................... 1158

A Grammar summary ..................................................................... 1178
A.1 Keywords ................................................................................ 1178
A.2 Lexical conventions ................................................................... 1178
A.3 Basic concepts .......................................................................... 1183
A.4 Expressions .............................................................................. 1183
A.5 Statements ................................................................................. 1186
A.6 Declarations ............................................................................. 1187
A.7 Declarators ................................................................................. 1191
A.8 Classes ...................................................................................... 1193
A.9 Derived classes .......................................................................... 1193
A.10 Special member functions .......................................................... 1194
A.11 Overloading .............................................................................. 1194
A.12 Templates ................................................................................. 1194
A.13 Exception handling ................................................................. 1195
A.14 Preprocessing directives ............................................................ 1196

B Implementation quantities ............................................................. 1198

C Compatibility ............................................................................. 1200
C.1 C++ and ISO C ........................................................................ 1200
C.2 Standard C library ..................................................................... 1209

D Compatibility features ................................................................. 1214
D.1 Increment operator with bool operand ........................................... 1214
D.2 static keyword .......................................................................... 1214
D.3 Access declarations ................................................................... 1214
D.4 register keyword ....................................................................... 1214
D.5 Dynamic exception specifications ............................................... 1214
D.6 C standard library headers .......................................................... 1214
D.7 Old iostreams members ................................................................. 1215
D.8 char* streams ........................................................................... 1216
D.9 Binders ..................................................................................... 1225
D.10 auto_ptr ................................................................................ 1227

E Cross references ......................................................................... 1230

CONTENTS ix
Index 1247
Index of Grammar Productions 1272
Index of Library Names 1275
Index of Implementation Defined Behavior 1309
# List of Tables

1. Trigraph sequences .................................................. 17
2. Alternative tokens .................................................... 19
3. Keywords .................................................................. 21
4. Alternative representations ......................................... 21
5. Types of integer constants ........................................... 23
6. Escape sequences ........................................................ 25
7. String literal concatenations ......................................... 28
8. Relations on `const` and `volatile` ................................. 74
9. `simple-type-specifiers` and the types they specify .......... 147
10. Relationship between operator and function call notation .................................................. 291
11. Conversions ............................................................. 299
12. Library categories ..................................................... 410
13. C++ library headers ................................................... 422
14. C++ headers for C library facilities .............................. 422
15. C++ headers for freestanding implementations ............ 423
16. Language support library summary ............................... 432
17. Header `<cassert>` synopsis ........................................ 465
18. Header `<cerrno>` synopsis .......................................... 466
19. General utilities library summary ................................ 478
20. EqualityComparable requirements ............................... 479
21. LessThanComparable requirements ......................... 479
22. DefaultConstructible requirements ............................. 479
23. MoveConstructible requirements ................................. 479
24. CopyConstructible requirements (in addition to MoveConstructible) ........................................ 479
25. MoveAssignable requirements ..................................... 479
26. CopyAssignable requirements (in addition to MoveAssignble) .................................................. 480
27. Destructible requirements .......................................... 480
28. NullablePointer requirements .................................... 483
<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>Hash requirements</td>
</tr>
<tr>
<td>41</td>
<td>Descriptive variable definitions</td>
</tr>
<tr>
<td>42</td>
<td>Allocator requirements</td>
</tr>
<tr>
<td>43</td>
<td>Primary type category predicates</td>
</tr>
<tr>
<td>44</td>
<td>Composite type category predicates</td>
</tr>
<tr>
<td>45</td>
<td>Type property predicates</td>
</tr>
<tr>
<td>46</td>
<td>Type property queries</td>
</tr>
<tr>
<td>47</td>
<td>Type relationship predicates</td>
</tr>
<tr>
<td>48</td>
<td>Const-volatile modifications</td>
</tr>
<tr>
<td>49</td>
<td>Reference modifications</td>
</tr>
<tr>
<td>50</td>
<td>Sign modifications</td>
</tr>
<tr>
<td>51</td>
<td>Array modifications</td>
</tr>
<tr>
<td>52</td>
<td>Pointer modifications</td>
</tr>
<tr>
<td>53</td>
<td>Other transformations</td>
</tr>
<tr>
<td>54</td>
<td>Header <code>&lt;cstdlib&gt;</code> synopsis</td>
</tr>
<tr>
<td>55</td>
<td>Header <code>&lt;cstring&gt;</code> synopsis</td>
</tr>
<tr>
<td>56</td>
<td>Clock requirements</td>
</tr>
<tr>
<td>57</td>
<td>Header <code>&lt;ctime&gt;</code> synopsis</td>
</tr>
<tr>
<td>58</td>
<td>Strings library summary</td>
</tr>
<tr>
<td>59</td>
<td>Character traits requirements</td>
</tr>
<tr>
<td>60</td>
<td><code>basic_string(const Allocator&amp;)</code> effects</td>
</tr>
<tr>
<td>61</td>
<td><code>basic_string(const basic_string&amp;)</code> effects</td>
</tr>
<tr>
<td>62</td>
<td><code>basic_string(const basic_string&amp;, size_type, size_type, const Allocator&amp;)</code> effects</td>
</tr>
<tr>
<td>63</td>
<td><code>basic_string(const charT*, size_type, const Allocator&amp;)</code> effects</td>
</tr>
<tr>
<td>64</td>
<td><code>basic_string(const charT*, const Allocator&amp;)</code> effects</td>
</tr>
<tr>
<td>65</td>
<td><code>basic_string(size_t, charT, const Allocator&amp;)</code> effects</td>
</tr>
<tr>
<td>66</td>
<td><code>basic_string(const basic_string&amp;, const Allocator&amp;)</code> and <code>basic_string(basic_string&amp;, const Allocator&amp;)</code> effects</td>
</tr>
<tr>
<td>67</td>
<td><code>operator=(const basic_string&lt;charT, traits, Allocator&gt;&amp;)</code> effects</td>
</tr>
<tr>
<td>68</td>
<td><code>operator=(const basic_string&lt;charT, traits, Allocator&gt;&amp;&amp;)</code> effects</td>
</tr>
<tr>
<td>69</td>
<td><code>compare()</code> results</td>
</tr>
<tr>
<td>70</td>
<td>Potential <code>mbstate_t</code> data races</td>
</tr>
<tr>
<td>71</td>
<td>Header <code>&lt;cctype&gt;</code> synopsis</td>
</tr>
<tr>
<td>72</td>
<td>Header <code>&lt;cwctype&gt;</code> synopsis</td>
</tr>
<tr>
<td>73</td>
<td>Header <code>&lt;cstring&gt;</code> synopsis</td>
</tr>
<tr>
<td>74</td>
<td>Header <code>&lt;cwchar&gt;</code> synopsis</td>
</tr>
<tr>
<td>75</td>
<td>Header <code>&lt;cstdlib&gt;</code> synopsis</td>
</tr>
<tr>
<td>76</td>
<td>Header <code>&lt;uchar&gt;</code> synopsis</td>
</tr>
<tr>
<td>77</td>
<td>Localization library summary</td>
</tr>
<tr>
<td>78</td>
<td>Locale category facets</td>
</tr>
<tr>
<td>79</td>
<td>Required specializations</td>
</tr>
<tr>
<td>80</td>
<td><code>do_in/do_out</code> result values</td>
</tr>
<tr>
<td>81</td>
<td><code>do_unshift</code> result values</td>
</tr>
<tr>
<td>82</td>
<td>Integer conversions</td>
</tr>
<tr>
<td>83</td>
<td>Length modifier</td>
</tr>
<tr>
<td>84</td>
<td>Integer conversions</td>
</tr>
<tr>
<td>85</td>
<td>Floating-point conversions</td>
</tr>
<tr>
<td>86</td>
<td>Length modifier</td>
</tr>
<tr>
<td>87</td>
<td>Numeric conversions</td>
</tr>
</tbody>
</table>

List of Tables

List of Tables xii
<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>88</td>
<td>Fill padding</td>
</tr>
<tr>
<td>89</td>
<td><code>do_get_date</code> effects</td>
</tr>
<tr>
<td>90</td>
<td>Header <code>&lt;locale&gt;</code> synopsis</td>
</tr>
<tr>
<td>91</td>
<td>Potential <code>setlocale</code> data races</td>
</tr>
<tr>
<td>92</td>
<td>Containers library summary</td>
</tr>
<tr>
<td>93</td>
<td>Container requirements</td>
</tr>
<tr>
<td>94</td>
<td>Reversible container requirements</td>
</tr>
<tr>
<td>95</td>
<td>Optional container operations</td>
</tr>
<tr>
<td>96</td>
<td>Allocator-aware container requirements</td>
</tr>
<tr>
<td>97</td>
<td>Sequence container requirements (in addition to container)</td>
</tr>
<tr>
<td>98</td>
<td>Optional sequence container operations</td>
</tr>
<tr>
<td>99</td>
<td>Associative container requirements (in addition to container)</td>
</tr>
<tr>
<td>100</td>
<td>Unordered associative container requirements (in addition to container)</td>
</tr>
<tr>
<td>101</td>
<td>Iterators library summary</td>
</tr>
<tr>
<td>102</td>
<td>Relations among iterator categories</td>
</tr>
<tr>
<td>103</td>
<td>Iterator requirements</td>
</tr>
<tr>
<td>104</td>
<td>Input iterator requirements (in addition to <code>Iterator</code>)</td>
</tr>
<tr>
<td>105</td>
<td>Output iterator requirements (in addition to <code>Iterator</code>)</td>
</tr>
<tr>
<td>106</td>
<td>Forward iterator requirements (in addition to input iterator)</td>
</tr>
<tr>
<td>107</td>
<td>Bidirectional iterator requirements (in addition to forward iterator)</td>
</tr>
<tr>
<td>108</td>
<td>Random access iterator requirements (in addition to bidirectional iterator)</td>
</tr>
<tr>
<td>109</td>
<td>Algorithms library summary</td>
</tr>
<tr>
<td>110</td>
<td>Header <code>&lt;stdlib&gt;</code> synopsis</td>
</tr>
<tr>
<td>111</td>
<td>Numerics library summary</td>
</tr>
<tr>
<td>112</td>
<td>Seed sequence requirements</td>
</tr>
<tr>
<td>113</td>
<td>Uniform random number generator requirements</td>
</tr>
<tr>
<td>114</td>
<td>Random number engine requirements</td>
</tr>
<tr>
<td>115</td>
<td>Random number distribution requirements</td>
</tr>
<tr>
<td>116</td>
<td>Header <code>&lt;math&gt;</code> synopsis</td>
</tr>
<tr>
<td>117</td>
<td>Header <code>&lt;stdlib&gt;</code> synopsis</td>
</tr>
<tr>
<td>118</td>
<td>Input/output library summary</td>
</tr>
<tr>
<td>119</td>
<td><code>fmtflags</code> effects</td>
</tr>
<tr>
<td>120</td>
<td><code>fmtflags</code> constants</td>
</tr>
<tr>
<td>121</td>
<td><code>iostate</code> effects</td>
</tr>
<tr>
<td>122</td>
<td><code>openmode</code> effects</td>
</tr>
<tr>
<td>123</td>
<td><code>seekdir</code> effects</td>
</tr>
<tr>
<td>124</td>
<td>Position type requirements</td>
</tr>
<tr>
<td>125</td>
<td><code>basic_ios::init()</code> effects</td>
</tr>
<tr>
<td>126</td>
<td><code>basic_ios::copyfmt()</code> effects</td>
</tr>
<tr>
<td>127</td>
<td><code>seekoff</code> positioning</td>
</tr>
<tr>
<td>128</td>
<td><code>newoff</code> values</td>
</tr>
<tr>
<td>129</td>
<td>File open modes</td>
</tr>
<tr>
<td>130</td>
<td><code>seekoff</code> effects</td>
</tr>
<tr>
<td>131</td>
<td>Header <code>&lt;stdio&gt;</code> synopsis</td>
</tr>
<tr>
<td>132</td>
<td>Header <code>&lt;cstdint&gt;</code> synopsis</td>
</tr>
<tr>
<td>133</td>
<td>Regular expressions library summary</td>
</tr>
</tbody>
</table>

List of Tables
List of Tables

List of Tables

134 Regular expression traits class requirements ................................................. 1061
135 syntax_option_type effects ................................................................. 1071
136 regex_constants::match_flag_type effects when obtaining a match against a character container sequence [first,last) ......................................................... 1072
137 error_type values in the C locale .............................................................. 1073
138 match_results assignment operator effects ................................................. 1090
139 Effects of regex_match algorithm ............................................................. 1093
140 Effects of regex_search algorithm ............................................................. 1095
141 Atomics library summary ............................................................................. 1107
142 Atomics for built-in types ............................................................................ 1117
143 Atomics for standard typedef types ............................................................ 1118
144 Atomic arithmetic computations .................................................................. 1125
145 Thread support library summary .................................................................. 1129
146 Standard macros .......................................................................................... 1210
147 Standard values ........................................................................................... 1210
148 Standard types ............................................................................................. 1210
149 Standard structs .......................................................................................... 1210
150 Standard functions ....................................................................................... 1211
151 C headers ..................................................................................................... 1214
152 strstreambuf(streamsize) effects ................................................................. 1218
153 strstreambuf(void* (*)(size_t), void (*)(void*)) effects ................................. 1218
154 strstreambuf(charT*, streamsize, charT*) effects .......................................... 1219
155 seekoff positioning ....................................................................................... 1221
156 newoff values ............................................................................................... 1221

© ISO/IEC N3092
List of Figures

1  Expression category taxonomy ....................................................... 74
2  Directed acyclic graph ............................................................... 227
3  Non-virtual base ...................................................................... 228
4  Virtual base ............................................................................. 229
5  Virtual and non-virtual base ...................................................... 229
6  Name lookup ........................................................................... 231
7  Stream position, offset, and size types [non-normative] ............. 972
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 9899:1999 Programming languages — C (hereinafter referred to as the C standard). In addition to the facilities provided by C, C++ provides additional data types, classes, templates, exceptions, namespaces, 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 9899:1999, Programming languages — C
— ISO/IEC 9899:1999/Cor.1:2001(E), Programming languages — C, Technical Corrigendum 1
— ISO/IEC TR 19769:2004, Information technology — Programming languages, their environments and system software interfaces — Extensions for the programming language C to support new character data types


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

¹) With the qualifications noted in Clauses 18 through 30 and in C.2, the C standard library is a subset of the C++ standard library.
The operating system interface described in ISO/IEC 9945:2003 is hereinafter called POSIX.
The ECMAScript Language Specification described in Standard Ecma-262 is hereinafter called ECMA-262.

1.3 Definitions

For the purposes of this International Standard, the definitions given in ISO/IEC 2382 and the following definitions apply. 17.3 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 glvalue denoted by a glvalue expression refers. [Example: if a pointer (8.3.1) p whose static type is “pointer to class B” is pointing to an object of class D, derived from B (Clause 10), the dynamic type of the expression *p is “D.” References (8.3.2) are treated similarly.—end example] The dynamic type of a prvalue expression is its static type.

1.3.5 ill-formed program
a program that is not well formed.

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
implementation limits
restrictions imposed upon programs by the implementation.

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

1.3.9 [defns.multibyte]
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 (2.3).

1.3.10 [defns.parameter]
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 [defns.signature]
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 [defns.static.type]
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 [defns.undefined]
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

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

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

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

— 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.

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

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

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

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

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

7 Two kinds of implementations are defined: 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 (17.6.1.3).

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

2) “Correct execution” can include undefined behavior, depending on the data being processed; see 1.3 and 1.9.
Each implementation shall include documentation that identifies all conditionally-supported constructs that it does not support and defines all locale-specific characteristics.  

1.5 Structure of this International Standard

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

Annex B recommends lower bounds on the capacity of conforming implementations.

Annex C summarizes the evolution of C++ since its first published description, and explains in detail the differences between C++ and C. Certain features of C++ exist solely for compatibility purposes; Annex D describes those features.

Throughout this International Standard, each example is introduced by “[Example:]” and terminated by “— end example]”. Each note is introduced by “[Note:]” and terminated by “— end note]”. Examples and notes may be nested.

1.6 Syntax notation

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

```
{ expression opt }
```

indicates an optional expression enclosed in braces.

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

— 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.

---

3) This documentation also defines implementation-defined behavior; see 1.9.
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 (1.10) can update and access separate memory locations without interfering with each other.

[A structure declared as

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

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

1.8 The C++ object model

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

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

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

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

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

If a complete object, a data member (9.2), or an array element is of class type, its type is considered the most derived class, to distinguish it from the class type of any base class subobject; an object of a most derived class type or of a non-class type is called a most derived object.
Unless it is a bit-field (9.6), a most derived object shall have a non-zero size and shall occupy one or more bytes of storage. Base class subobjects may have zero size. An object of trivially copyable or standard-layout type (3.9) shall occupy contiguous bytes of storage.

Unless an object is a bit-field or a base class subobject of zero size, the address of that object is the address of the first byte it occupies. Two distinct objects that are neither bit-fields nor base class subobjects of zero size shall have distinct addresses.4

Example:

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

— end example

[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.5

2 Certain aspects and operations of the abstract machine are described in this International Standard as implementation-defined (for example, sizeof(int)). These constitute the parameters of the abstract machine. Each implementation shall include documentation describing its characteristics and behavior in these respects.6 Such documentation shall define the instance of the abstract machine that corresponds to that implementation (referred to as the “corresponding instance” below).

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 of allowable behaviors. These define the nondeterministic aspects of the abstract machine. An instance of the abstract machine can thus have more than one possible execution for a given program and a given input.

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 executions of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution contains an undefined operation, this International Standard places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).

4) Under the “as-if” rule an implementation is allowed to store two objects at the same machine address or not store an object at all if the program cannot observe the difference (1.9).

5) This provision is sometimes called the “as-if” rule, because an implementation is free to disregard any requirement of this International Standard as long as the result is as if the requirement had been obeyed, as far as can be determined from the observable behavior of the program. For instance, an actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no side effects affecting the observable behavior of the program are produced.

6) This documentation also includes conditionally-supported constructs and locale-specific behavior. See 1.4.
When the processing of the abstract machine is interrupted by receipt of a signal, the values of objects which are neither

— of type `volatile std::sig_atomic_t` nor

— lock-free atomic objects (29.4)

are unspecified, and the value of any object not in either of these two categories that is modified by the handler becomes undefined.

An instance of each object with automatic storage duration (3.7.3) is associated with each entry into its block. Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of a signal).

The least requirements on a conforming implementation are:

— 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 output is actually delivered before a program waits for input. What constitutes an interactive device is implementation-defined.

These collectively are referred to as the observable behavior of the program. [Note: more stringent correspondences between abstract and actual semantics may be defined by each implementation. — end note]

Note: operators can be regrouped according to the usual mathematical rules only where the operators really are associative or commutative. For example, in the following fragment

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

the expression statement behaves exactly the same as

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

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

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

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

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

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

7) Overloaded operators are never assumed to be associative or commutative.
since the values for a and b might have been, respectively, 4 and -8 or -17 and 12. However on a machine in which overflows do not produce an exception and in which the results of overflows are reversible, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur.  — end note]  

10 A full-expression is an expression that is not a subexpression of another expression. 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:  
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]  

11 [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]  

12 Accessing an object designated by a volatile glvalue (3.10), modifying an object, calling a library I/O function, or calling a function that does any of those operations are all side effects, which are changes in the state of the execution environment. Evaluation of an expression (or a sub-expression) in general includes both value computations (including determining the identity of an object for glvalue evaluation and fetching a value previously assigned to an object for prvalue evaluation) and initiation of side effects. When a call to a library I/O function returns or an access to a volatile object is evaluated the side effect is considered complete, even though some external actions implied by the call (such as the I/O itself) or by the volatile access may not have completed yet.  

13 Sequenced before is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread (1.10), which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B. If A is not sequenced before B and B is not sequenced before A, then A and B are unsequenced. [Note: The execution of unsequenced evaluations can overlap. — end note] Evaluations A and B are indeterminately sequenced when either A is sequenced before B or B is sequenced before A, but it is unspecified which. [Note: Indeterminately sequenced evaluations cannot overlap, but either could be executed first. — end note]
Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated.  

Except where noted, evaluations of operands of individual operators and of subexpressions of individual expressions are unsequenced.  [Note: In an expression that is evaluated more than once during the execution of a program, unsequenced and indeterminately sequenced evaluations of its subexpressions need not be performed consistently in different evaluations. — end note] The value computations of the operands of an operator are sequenced before the value computation of the result of the operator. If a side effect on a scalar object is unsequenced relative to either another side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined.  

[Example:

```c
void f(int, int);
void g(int i, int* v) {
    i = v[i++]; // the behavior is undefined
    i = 7, i++, i++; // i becomes 9
    i = i++ + 1; // the behavior is undefined
    i = i + 1; // the value of i is incremented
    f(i = -1, i = -1); // the behavior is undefined
}
```
— 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 post-fix 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.  [Example: Evaluation of a new expression invokes one or more allocation and constructor functions; see 5.3.4. For another example, invocation of a conversion function (12.3.2) can arise in contexts in which no function call syntax appears. — end example] The sequencing constraints on the execution of the called function (as described above) are features of the function calls as evaluated, whatever the syntax of the expression that calls the function might be.

1.10 Multi-threaded executions and data races  

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

Under a hosted implementation, a C++ program can have more than one thread running concurrently. The execution of each thread proceeds as defined by the remainder of this standard. The execution of the entire program consists of an execution of

---

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.

10) An object with automatic or thread storage duration (3.7) is associated with one specific thread, and can be accessed by a different thread only indirectly through a pointer or reference (3.9.2).
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 is the initial value of the object, a value assigned to the object by $T$, or a value assigned to the object by another thread, according to the rules below. [Note: In some cases, there may instead be undefined behavior. Much of this section is motivated by the desire to support atomic operations with explicit and detailed visibility constraints. However, it also implicitly supports a simpler view for more restricted programs. — end note]

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 on one or more memory locations is either a consume operation, an acquire operation, a release operation, or both an acquire and release operation. A synchronization operation without an associated memory location is a fence and can be either an acquire fence, a release fence, or both an acquire and release fence. In addition, there are relaxed atomic operations, which are not synchronization operations, and atomic read-modify-write operations, which have special characteristics. [Note: For example, a call that acquires a 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 a consume or an acquire operation on $A$. “Relaxed” atomic operations are not synchronization operations even though, like synchronization operations, they cannot contribute to data races. — end note]

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 (as defined below) $B$, then $A$ shall precede $B$ in the modification order of $M$, which is defined below. [Note: This states that the modification orders must respect the “happens before” relationship. — end note] [Note: There is a separate order for each atomic object. There is no requirement that these can be combined into a single total order for all objects. In general this will be impossible since different threads may observe modifications to different objects in inconsistent orders. — end note]

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 an atomic read-modify-write operation.

7 Certain library calls synchronize with other library calls performed by another thread. In particular, an atomic operation $A$ that performs a release operation on an atomic object $M$ synchronizes with an atomic operation $B$ that performs an acquire operation on $M$ and 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 objects, 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 $A$ carries a dependency to an evaluation $B$ if

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

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

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

— \( A \) is the left operand of the built-in comma (`,`) operator (5.18);

or

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

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

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

9 An evaluation \( A \) is \textit{dependency-ordered before} an evaluation \( B \) if

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

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

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

10 An evaluation \( A \) \textit{inter-thread happens before} an evaluation \( B \) if

— \( A \) synchronizes with \( B \), or

— \( A \) is dependency-ordered before \( B \), or

— for some evaluation \( X \)

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

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

— \( A \) inter-thread happens before \( X \) and \( X \) inter-thread happens before \( B \).

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

11 An evaluation \( A \) \textit{happens before} an evaluation \( B \) if:

— \( A \) is sequenced before \( B \), or

— \( A \) inter-thread happens before \( B \).
A visible side effect \( A \) on a scalar object or bit-field \( M \) with respect to a value computation \( B \) of \( M \) satisfies the conditions:

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

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. This effectively makes 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]

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]

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


All rights in these originals are reserved.
2 Lexical conventions

2.1 Separate translation

The text of the program is kept in units called source files in this International Standard. A source file together with all the headers (17.6.1.2) and source files included (16.2) via the preprocessing directive \#include, less any source lines skipped by any of the conditional inclusion (16.1) preprocessing directives, is called a translation unit. [Note: a C++ program need not all be translated at the same time. — end note]

[Note: previously translated translation units and instantiation units can be preserved individually or in libraries. The separate translation units of a program communicate (3.5) by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. Translation units can be separately translated and then later linked to produce an executable program (3.5). — end note]

2.2 Phases of translation

The precedence among the syntax rules of translation is specified by the following phases.11

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 (2.4) are replaced by corresponding single-character internal representations. Any source file character not in the basic source character set (2.3) is replaced by the universal-character-name that designates that character. (An implementation may use any internal encoding, so long as an actual extended character encountered in the source file, and the same extended character expressed in the source file as a universal-character-name (i.e., using the \uXXXX notation), are handled equivalently.)

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. A source file that is not empty and that does not end in a new-line character, or that ends in a new-line character immediately preceded by a backslash character before any such splicing takes place, shall be processed as if an additional new-line character were appended to the file.

3. The source file is decomposed into preprocessing tokens (2.5) and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment.12 Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is unspecified. The process of dividing a source file’s characters into preprocessing tokens is context-dependent. [Example: see the handling of < within a \#include preprocessing directive.
— end example] Within the r-char-sequence of a raw string literal, any transformations performed in phases 1 and 2 (trigraphs, universal-character-names, and line splicing) are reverted.

---

11) Implementations must behave as if these separate phases occur, although in practice different phases might be folded together.
12) A partial preprocessing token would arise from a source file ending in the first portion of a multi-character token that requires a terminating sequence of characters, such as a header-name that is missing the closing * or >. A partial comment would arise from a source file ending with an unclosed /* comment.
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 (2.14.3, 2.14.5); if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.\(^\text{13}\)

6. Adjacent string literal tokens are concatenated.

7. White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token. (2.7). The resulting tokens are syntactically and semantically analyzed and translated as a translation unit. [Note: The process of analyzing and translating the tokens may occasionally result in one token being replaced by a sequence of other tokens (14.2). — end note] [Note: Source files, translation units and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation. — end note]

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

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

2.3 Character sets

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

```
 a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
_ { } [ ] ( ) < > % ; . ? * + - / ~ & | = , \ " '
```

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

\(^{14}\) The glyphs for the members of the basic source character set are intended to identify characters from the subset of ISO/IEC 10646 which corresponds to the ASCII character set. However, because the mapping from source file characters to the source character set (described in translation phase 1) is specified as implementation-defined, an implementation is required to document how the basic source characters are represented in source files.
The universal-character-name construct provides a way to name other characters.

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

The character designated by the universal-character-name \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 0xD800–0xDBFF, inclusive), the program is ill-formed. Additionally, if the hexadecimal value for a universal-character-name outside the c-char-sequence, s-char-sequence, or r-char-sequence of a character or string literal corresponds to a control character (in either of the ranges 0x00–0x1F or 0x7F–0x9F, both inclusive) or to a character in the basic source character set, the program is ill-formed.

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 implementation-defined supersets of the basic execution character set and the basic execution wide-character set, respectively. The values of the members of the execution character sets and the sets of additional members are locale-specific.

### 2.4 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>
</tr>
</thead>
<tbody>
<tr>
<td>??(</td>
<td>]</td>
</tr>
<tr>
<td>??)</td>
<td>]</td>
</tr>
<tr>
<td>??&lt;</td>
<td>{</td>
</tr>
<tr>
<td>??/&gt;</td>
<td>}</td>
</tr>
<tr>
<td>??=</td>
<td>#</td>
</tr>
<tr>
<td>??!</td>
<td></td>
</tr>
<tr>
<td>??-</td>
<td>~</td>
</tr>
</tbody>
</table>
```

[Example:]

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

becomes

```
#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.
2.5 Preprocessing tokens

preprocessing-token:
  header-name
  identifier
  pp-number
  character-literal
  user-defined-character-literal
  string-literal
  user-defined-string-literal
  preprocessing-op-or-punc
  each non-white-space character that cannot be one of the above

1 Each preprocessing token that is converted to a token (2.7) shall have the lexical form of a keyword, an identifier, a literal, an operator, or a punctuator.

2 A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing token are: header names, identifiers, preprocessing numbers, character literals (including user-defined character literals), string literals (including user-defined string literals), preprocessing operators and punctuators, and single non-white-space characters that do not lexically match the other preprocessing token categories. If a ‘ or a ” character matches the last category, the behavior is undefined. Preprocessing tokens can be separated by white space; this consists of comments (2.8), 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.

3 If the input stream has been parsed into preprocessing tokens up to a given character:
   — if the next character begins a sequence of characters that could be the prefix and initial double quote of a raw string literal, such as R", the next preprocessing token shall be a raw string literal;
   — otherwise, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token, even if that would cause further lexical analysis to fail.

[ Example:
#define R "x"
const char* s = R"y"; // ill-formed raw string, not "x" "y"
— end example ]

4 [ 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 ]

5 [ 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 ]

2.6 Alternative tokens

Alternative token representations are provided for some operators and punctuators.\textsuperscript{15}

\textsuperscript{15} These include “digraphs” and additional reserved words. The term “digraph” (token consisting of two characters) is not
In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling. The set of alternative tokens is defined in Table 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><code>&lt;%</code></td>
<td><code>{</code></td>
<td><code>and</code></td>
<td><code>&amp;</code></td>
<td><code>and_eq</code></td>
<td><code>&amp;=</code></td>
</tr>
<tr>
<td><code>%&gt;</code></td>
<td><code>}</code></td>
<td><code>bitor</code></td>
<td>`</td>
<td>`</td>
<td><code>or_eq</code></td>
</tr>
<tr>
<td><code>&lt;:</code></td>
<td><code>[</code></td>
<td><code>or</code></td>
<td>`</td>
<td></td>
<td>`</td>
</tr>
<tr>
<td><code>:</code></td>
<td><code>]</code></td>
<td><code>xor</code></td>
<td><code>~</code></td>
<td><code>not</code></td>
<td><code>!</code></td>
</tr>
<tr>
<td><code>:%</code></td>
<td><code>#</code></td>
<td><code>compl</code></td>
<td><code>~</code></td>
<td><code>not_eq</code></td>
<td><code>!=</code></td>
</tr>
<tr>
<td><code>:%:</code></td>
<td><code>##</code></td>
<td><code>bitand</code></td>
<td><code>&amp;</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 2.7 Tokens

**token:**
- **identifier**
- **keyword**
- **literal**
- **operator**
- **punctuator**

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

### 2.8 Comments

1. 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]*

### 2.9 Header names

**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 `>`

perfectly descriptive, since one of the alternative preprocessing-tokens is `%;%` and of course several primary tokens contain two characters. Nonetheless, those alternative tokens that aren’t lexical keywords are colloquially known as “digraphs”.

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

17) Literals include strings and character and numeric literals.
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.

The appearance of either of the characters `\` or `/` or of either of the character sequences `/*` or `//` in a `q-char-sequence` or an `h-char-sequence` is conditionally supported with implementation-defined semantics, as is the appearance of the character `"` in an `h-char-sequence`.\(^{18}\)

2.10 Preprocessing numbers \([\text{lex.ppnumber}]\)

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

1 Preprocessing number tokens lexically include all integral literal tokens (2.14.2) and all floating literal tokens (2.14.4).

2 A preprocessing number does not have a type or a value; it acquires both after a successful conversion to an integral literal token or a floating literal token.

2.11 Identifiers \([\text{lex.name}]\)

```
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 A of TR 10176:2003. Upper- and lower-case letters are different. All characters are significant.\(^{19}\)

---

\(^{18}\) Thus, a sequence of characters that resembles an escape sequence might result in an error, be interpreted as the character corresponding to the escape sequence, or have a completely different meaning, depending on the implementation.

\(^{19}\) 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
In addition, some identifiers are reserved for use by C++ implementations and standard libraries (17.6.3.3.2) and shall not be used otherwise; no diagnostic is required.

2.12 Keywords

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

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

Furthermore, the alternative representations shown in Table 4 for certain operators and punctuators (2.6) are reserved and shall not be used otherwise:

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

2.13 Operators and punctuators

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

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

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.
Each *preprocessing-op-or-punc* is converted to a single token in translation phase 7 (2.2).

## 2.14 Literals

### 2.14.1 Kinds of literals

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

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

### 2.14.2 Integer literals

```
integer-literal:
  decimal-literal integer-suffix\(_{\text{opt}}\)
  octal-literal integer-suffix\(_{\text{opt}}\)
  hexadecimal-literal integer-suffix\(_{\text{opt}}\)

decimal-literal:
  nonzero-digit
  decimal-literal digit

octal-literal:
  0
  octal-literal octal-digit

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

nonzero-digit: one of
  1 2 3 4 5 6 7 8 9
octal-digit: one of
  0 1 2 3 4 5 6 7
hexadecimal-digit: one of
  0 1 2 3 4 5 6 7 8 9
  a b c d e f
  A B C D E F

integer-suffix:
  unsigned-suffix long-suffix\(_{\text{opt}}\)
  unsigned-suffix long-long-suffix\(_{\text{opt}}\)
  long-suffix unsigned-suffix\(_{\text{opt}}\)
  long-long-suffix unsigned-suffix\(_{\text{opt}}\)

unsigned-suffix: one of
  u U
long-suffix: one of
  l L
long-long-suffix: one of
  ll LL
```

\(^{20}\) The term “literal” generally designates, in this International Standard, those tokens that are called “constants” in ISO C.
An integer literal is a sequence of digits that has no period or exponent part. An integer literal may have a prefix that specifies its base and a suffix that specifies its type. The lexically first digit of the sequence of digits is the most significant. A decimal integer literal (base ten) begins with a digit other than 0 and consists of a sequence of decimal digits. An octal integer literal (base eight) begins with the digit 0 and consists of a sequence of octal digits. A hexadecimal integer literal (base sixteen) begins with 0x or 0X and consists of a sequence of hexadecimal digits, which include the decimal digits and the letters a through f and A through F with decimal values ten through fifteen. [Example: the number twelve can be written 12, 014, or 0XC. — end example]

The type of an integer literal is the first of the corresponding list in Table 5 in which its value can be represented.

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Decimal constant</th>
<th>Octal or hexadecimal constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>long 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></td>
<td></td>
<td>int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned 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></td>
<td></td>
<td>int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>Both u or U and l or L</td>
<td>unsigned long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>ll or LL</td>
<td>long long int</td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long int</td>
</tr>
<tr>
<td>Both u or U and ll or LL</td>
<td>unsigned long int</td>
<td>unsigned 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 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.

2.14.3 Character literals

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

21) The digits 8 and 9 are not octal digits.
c-char-sequence:
c-char
  c-char-sequence c-char

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

escape-sequence
universal-character-name

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

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

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

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

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

2 A character literal that begins with the letter u, such as u’y’, is a character literal of type char16_t. The
value of a char16_t literal containing a single c-char is equal to its ISO 10646 code point value, provided that
the code point is representable with a single 16-bit code unit. (That is, provided it is a basic multi-lingual
plane code point.) If the value is not representable within 16 bits, the program is ill-formed. A char16_t
literal containing multiple c-chars is ill-formed. A character literal that begins with the letter U, such as
U’z’, is a character literal of type char32_t. The value of a char32_t literal containing a single c-char is
equal to its ISO 10646 code point value. A char32_t literal containing multiple c-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.22 The value of a wide-character literal containing a single c-char has value equal
to the numerical value of the encoding of the c-char in the execution wide-character set, unless the c-char
has no representation in the execution wide-character set, in which case the value is implementation-defined.
[Note: the type wchar_t is able to represent all members of the execution wide-character set (see 3.9.1).
—end note]. The value of a wide-character literal containing multiple c-chars is implementation-defined.

3 Certain nongraphic characters, the single quote ‘, the double quote ”, the question mark ?,23 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

---

22) They are intended for character sets where a character does not fit into a single byte.
23) Using an escape sequence for a question mark can avoid accidentally creating a trigraph.
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>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-line</td>
<td>NL(LF) \n</td>
</tr>
<tr>
<td>horizontal tab</td>
<td>HT \t</td>
</tr>
<tr>
<td>vertical tab</td>
<td>VT \v</td>
</tr>
<tr>
<td>backspace</td>
<td>BS \b</td>
</tr>
<tr>
<td>carriage return</td>
<td>CR \r</td>
</tr>
<tr>
<td>form feed</td>
<td>FF \f</td>
</tr>
<tr>
<td>alert</td>
<td>BEL \a</td>
</tr>
<tr>
<td>backslash</td>
<td>\ \</td>
</tr>
<tr>
<td>question mark</td>
<td>? ?</td>
</tr>
<tr>
<td>single quote</td>
<td>’ ’</td>
</tr>
<tr>
<td>double quote</td>
<td>” ”</td>
</tr>
<tr>
<td>octal number</td>
<td>ooo \ooo</td>
</tr>
<tr>
<td>hex number</td>
<td>hhh \xhhh</td>
</tr>
</tbody>
</table>

4 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’).

5 A universal-character-name is translated to the encoding, in the appropriate execution character set, of the character named. If there is no such encoding, the universal-character-name is translated to an implementation-defined encoding. [Note: in translation phase 1, a universal-character-name is introduced whenever an actual extended character is encountered in the source text. Therefore, all extended characters are described in 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]

2.14.4 Floating literals

floating-literal:
  fractional-constant exponent-part_opt floating-suffix_opt
  digit-sequence exponent-part_opt 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
A floating literal consists of an integer part, a decimal point, a fraction part, an \texttt{e} or \texttt{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 \texttt{e} (or \texttt{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.

### 2.14.5 String literals

**(string-literal):**

\texttt{encoding-prefix \texttt{\textquotedblright} s-char-sequence \texttt{\textquotedblright} encoding-prefix \texttt{\textquotedblright} R raw-string}

\textit{encoding-prefix:}

\texttt{u8}

\texttt{u}

\texttt{U}

\texttt{L}

\textit{s-char-sequence:}

\texttt{s-char}

\texttt{s-char-sequence s-char}

\textit{s-char:}

any member of the source character set except

- the double-quote \texttt{"}, backslash \texttt{	extbackslash}, or new-line character
- escape-sequence
- universal-character-name

\textit{raw-string:}

\texttt{" d-char-sequence \texttt{\textquotedblright} ( r-char-sequence \texttt{\textquotedblright}) d-char-sequence \texttt{\textquotedblright} \texttt{\textquotedblright}"

\textit{r-char-sequence:}

\texttt{r-char}

\texttt{r-char-sequence r-char}

\textit{r-char:}

any member of the source character set, except

- a right parenthesis \texttt{)}, followed by the initial \texttt{d-char-sequence}
  (which may be empty) followed by a double quote \texttt{"}.

\textit{d-char-sequence:}

\texttt{d-char}

\texttt{d-char-sequence d-char}

\textit{d-char:}

any member of the basic source character set except:

- space, the left parenthesis \texttt{\texttt{(}}, the right parenthesis \texttt{\texttt{)}}, the backslash \texttt{\textbackslash},
- and the control characters representing horizontal tab,
  vertical tab, form feed, and newline.

§ 2.14.5
A string literal is a sequence of characters (as defined in §2.14.3) surrounded by double quotes, optionally prefixed by \( R \), \( u8 \), \( u8R \), \( u \), \( uR \), \( U \), \( UR \), \( L \), or \( LR \), as in \("."", \( R"(...)" \), \( u8"..." \), \( u8R"**(...)**" \), \( u"..." \), \( uR"*~(...)*~" \), \( U"..." \), \( UR"z\ldots z" \), \( L"..." \), or \( LR"(...)" \), respectively.

A string literal that has an \( R \) in the prefix is a raw string literal. The \( d \)-char-sequence serves as a delimiter. The terminating \( d \)-char-sequence of a raw-string is the same sequence of characters as the initial \( d \)-char-sequence. A \( d \)-char-sequence shall consist of at most 16 characters.\(^{24}\)

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

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

```c
const char *p = R"(a\nb\nc)";
assert(std::strcmp(p, "a\\nb\nc") == 0);
```

— end note]

After translation phase 6, a string literal that does not begin with an encoding-prefix is an ordinary string literal, and is initialized with the given characters.

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.

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 \( u"asdf" \), is a \( \text{char16\_t} \) string literal. A \( \text{char16\_t} \) string literal has type “array of \( n \) const \( \text{char16\_t} \)”, 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 \( \text{c\_char} \) may produce more than one \( \text{char16\_t} \) character in the form of surrogate pairs.

A string literal that begins with \( U \), such as \( U"asdf" \), is a \( \text{char32\_t} \) string literal. A \( \text{char32\_t} \) string literal has type “array of \( n \) const \( \text{char32\_t} \)”, where \( n \) is the size of the string as defined below; it has static storage duration and is initialized with the given characters.

A string literal that begins with \( L \), such as \( L"asdf" \), is a wide string literal. A wide string literal has type “array of \( n \) const \( \text{wchar\_t} \)”, where \( n \) is the size of the string as defined below; it has static storage duration and is initialized with the given characters.

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

In translation phase 6 (2.2), adjacent string literals are concatenated. If both string literals have the same encoding-prefix, the resulting concatenated string literal has that encoding-prefix. If one string literal has no encoding-prefix, it is treated as a string literal of the same encoding-prefix as the other operand. If a UTF-8 string literal token is adjacent to a wide string literal token, the program is ill-formed. Any other concatenations are conditionally supported with implementation-defined behavior. [Note: This concatenation is an interpretation, not a conversion. Because the interpretation happens in translation phase 6 (after each character from a literal has been translated into a value from the appropriate character set), a string

\[ \text{§ 2.14.5} \]
Table 7 — String literal concatenations

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

The literal’s initial rawness has no effect on the interpretation or well-formedness of the concatenation. — end note] Table 7 has some examples of valid concatenations.

Characters in concatenated strings are kept distinct.

[Example:

"\xA" "B"

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

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

14 Escape sequences and universal-character-names in non-raw string literals have the same meaning as in character literals (2.14.3), except that the single quote ‘ is representable either by itself or by the escape sequence \, and the double quote " shall be preceded by a \. 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.

2.14.6 Boolean literals

boolean-literal:
false
true

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

2.14.7 Pointer literals

pointer-literal:
nullptr

24) Use of characters with trigraph equivalents in a d-char-sequence may produce unintended results.
The pointer literal is the keyword `nullptr`. It is a prvalue of type `std::nullptr_t`.

### 2.14.8 User-defined literals

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

user-defined-integer-literal:
  decimal-literal ud-suffix
  octal-literal ud-suffix
  hexadecimal-literal ud-suffix

user-defined-floating-literal:
  fractional-constant exponent-part_opt ud-suffix
  digit-sequence exponent-part ud-suffix

user-defined-string-literal:
  string-literal ud-suffix

user-defined-character-literal:
  character-literal ud-suffix

ud-suffix:
  identifier
```

1. If a token matches both `user-defined-literal` and another literal kind, it is treated as the latter. [Example: `123_km`, `1.2LL`, "Hello"s are all `user-defined-literals`, but `12LL` is an `integer-literal`. — end example]

2. A `user-defined-literal` is treated as a call to a literal operator or literal operator template (13.5.8). To determine the form of this call for a given `user-defined-literal` `L` with `ud-suffix` `X`, the `literal-operator-id` whose literal suffix identifier is `X` is looked up in the context of `L` using the rules for unqualified name lookup (3.4.1). Let `S` be the set of declarations found by this lookup. `S` shall not be empty.

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

   `operator "" X (n ULL)`

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

   `operator "" X ("n")`

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

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

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

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

   `operator "" X (f L)`

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

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

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

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

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

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

6 If \(L\) is a user-defined-character-literal, let \(ch\) be the literal without its ud-suffix. The literal \(L\) is treated as a call of the form

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

7 \([\text{Example:}]\)

```
long double operator " = w(long double);
std::string operator " = w(const char16_t*, size_t);
unsigned operator " = w(const char*);
int main() {
  1.2w;   // calls operator "w(1.2L)
  u"one"w; // calls operator "w(u"one", 3)
  12w;    // calls operator "w("12")
  "two"w; // error: no applicable literal operator
}
```

— end example]

8 In translation phase 6 (2.2), adjacent string literals are concatenated and user-defined-string-literals are considered string literals for that purpose. During concatenation, ud-suffixes are removed and ignored and the concatenation process occurs as described in 2.14.5. At the end of phase 6, if a string literal is the result of a concatenation involving at least one user-defined-string-literal, all the participating user-defined-string-literals shall have the same ud-suffix and that suffix is applied to the result of the concatenation.

9 \([\text{Example:}]\)

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

— end example]
3 Basic concepts [basic]

1 [Note: this Clause presents the basic concepts of the C++ language. It explains the difference between an object and a name and how they relate to the value categories for expressions. It introduces the concepts of a declaration and a definition and presents C++'s notion of type, scope, linkage, and storage duration. The mechanisms for starting and terminating a program are discussed. Finally, this Clause presents the fundamental types of the language and lists the ways of constructing compound types from these. — end note]

2 [Note: this Clause does not cover concepts that affect only a single part of the language. Such concepts are discussed in the relevant Clauses. — end note]

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

4 A name is a use of an identifier (2.11), operator-function-id (13.5), literal-operator-id (13.5.8), conversion-function-id (12.3.2), or template-id (14.2) that denotes an entity or label (6.6.4, 6.1).

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

6 A variable is introduced by the declaration of a reference other than a non-static data member or of an object. The variable's name denotes the reference or object.

7 Some names denote types or templates. In general, whenever a name is encountered it is necessary to determine whether that name denotes one of these entities before continuing to parse the program that contains it. The process that determines this is called name lookup (3.4).

8 Two names are the same if
   — they are identifiers composed of the same character sequence, or
   — they are operator-function-ids formed with the same operator, or
   — they are conversion-function-ids formed with the same type, or
   — they are template-ids that refer to the same class or function (14.4), or
   — they are the names of literal operators (13.5.8) formed with the same literal suffix identifier.

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

3.1 Declarations and definitions [basic.def]

1 A declaration (Clause 7) 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-specification25 (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), it is

25) Appearing inside the braced-enclosed declaration-seq in a linkage-specification does not affect whether a declaration is a definition.

§ 3.1
an opaque-enum-declaration (7.2), 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:

```c
typedef int Int; // declares Int
```

whereas these are just declarations:

```c
typedef int Int; // declares Int
```

— end example]

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

```c
#include <string>

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

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

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

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

— end example ]

4 [ Note: a class name can also be implicitly declared by an elaborated-type-specifier (7.1.6.3). — end note ]

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

3.2 One definition rule [basic.def.odr]

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

2 An expression is potentially evaluated unless it is an unevaluated operand (Clause 5) or a subexpression
thereof. A variable 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. this is used if it appears as a potentially-
evaluated expression (including as the result of the implicit transformation in the body of a non-static member
function (9.3.1)). A virtual member function is 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 or
move 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).26 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 move-assignment
function for a class is used by an implicitly-defined move-assignment function for another class as specified
in 12.8. A default constructor for a class is used by default initialization or value initialization as specified
in 8.5. A constructor for a class is used as specified in 8.5. A destructor for a class is used as specified in
12.4.

3 Every program shall contain exactly one definition of every non-inline function or variable that is used in
that program; no diagnostic required. The definition can appear explicitly in the program, it can be found
in the standard or a user-defined library, or (when appropriate) it is implicitly defined (see 12.1, 12.4 and
12.8). An inline function shall be defined in every translation unit in which it is used.

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

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

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

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

§ 3.2
— 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 lvalue-to-rvalue conversion is applied to a glvalue referring to an object of type \( T \) (4.1), or
— an expression is converted (either implicitly or explicitly) to type \( T \) (Clause 4, 5.2.3, 5.2.7, 5.2.9, 5.4), or
— an expression that is not a null pointer constant, and has type other than \texttt{void*}, is converted to the 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 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 \textit{exception-declaration} has type \( T \), reference to \( T \), or pointer to \( T \) (15.3).

[end note]
If D is a template and is defined in more than one translation unit, then the last four requirements from the list above shall apply to names from the template’s enclosing scope used in the template definition (14.6.3), and also to dependent names at the point of instantiation (14.6.2). If the definitions of D satisfy all these requirements, then the program shall behave as if there were a single definition of D. If the definitions of D do not satisfy these requirements, then the behavior is undefined.

3.3 Scope

3.3.1 Declarative regions and scopes

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

Example: in

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

the identifier j is declared twice as a name (and used twice). The declarative region of the first j includes the entire example. The potential scope of the first j begins immediately after that j and extends to the end of the program, but its (actual) scope excludes the text between the , and the }. The declarative region of the second declaration of j (the j immediately before the semicolon) includes all the text between { and }, but its potential scope excludes the declaration of i. The scope of the second declaration of j is the same as its potential scope. — end example]
The names declared by a declaration are introduced into the scope in which the declaration occurs, except that the presence of a `friend` specifier (11.4), 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 variable or enumerator, or all refer to functions and function templates; in this case the class name or enumeration name is hidden (3.3.10). [Note: a namespace name or a class template name must be unique in its declarative region (7.3.2, Clause 14). — end note]

[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]

### 3.3.2 Point of declaration

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

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

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

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

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

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

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

5 After the point of declaration of a class member, the member name can be looked up in the scope of its class. [Note: this is true even if the class is an incomplete class. For example,
struct X {
    enum E { z = 16 };
    int b[X::z];  // OK
};

— end note]

6 The point of declaration of a class first declared in an elaborated-type-specifier is as follows:
   — for a declaration of the form
     
     class-key attribute-specifier_opt identifier ;

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

     if the elaborated-type-specifier is used in the decl-specifier-seq or parameter-declaration-clause of a
     function defined in namespace scope, the identifier is declared as a class-name in the namespace that
     contains the declaration; otherwise, except as a friend declaration, the identifier is declared in the
     smallest 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]

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

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

9 The point of declaration for a template parameter is immediately after its complete template-parameter.
   [ Example:
     
     typedef unsigned char T;
     template<class T
        = T  // lookup finds the typedef name of unsigned char
        , T  // lookup finds the template parameter
        N = 0> struct A { };

     — end example]

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

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

3.3.3 Block scope [basic.scope.local]

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

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

3 The name declared in an 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, the for-range-declaration, and in the condition of if, while, for, and switch statements are local to the if, while, for, or switch statement (including the controlled statement), and shall not be redeclared in a subsequent condition of that statement nor in the outermost block (or, for the if statement, any of the outermost blocks) of the controlled statement; see 6.4.

### 3.3.4 Function prototype scope

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

### 3.3.5 Function scope

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

### 3.3.6 Namespace scope

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 (3.3.2) onwards; and for each using-directive (7.3.4) that nominates the member’s namespace, the member’s potential scope includes that portion of the potential scope of the using-directive that follows the member’s point of declaration. [Example:

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

§ 3.3.6
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.

The outermost declarative region of a translation unit is also a namespace, called the global namespace. A name declared in the global namespace has global namespace scope (also called global scope). The potential scope of such a name begins at its point of declaration (3.3.2) and ends at the end of the translation unit that is its declarative region. Names with global namespace scope are said to be global.

3.3.7 Class scope

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, brace-or-equal-initializers of non-static data members, 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 declarator-id, including a parameter-declaration-clause and any default arguments (8.3.6)).

[Example:

typedef int c;
enum { i = 1 };

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

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

§ 3.3.7
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.

3.3.8 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.

3.3.9 Template Parameter Scope

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

2 The declarative region of the name of a template parameter of a template is the smallest template-declaration
   in which the name was introduced. Only template parameter names belong to this declarative region; any
   other kind of name introduced by the declaration of a template-declaration is instead introduced into the
   same declarative region where it would be introduced as a result of a non-template declaration of the same
   name. [ Example:
   namespace N {
      template<class T> struct A { };  // #1
      template<class U> void f(U) { }  // #2
      struct B {
         template<class V> friend int g(struct C*);  // #3
      };
   }
   The declarative regions of T, U and V are the template-declarations on lines #1, #2 and #3, respectively.
   But the names A, f, g and C all belong to the same declarative region — namely, the namespace-body of N.
   (g is still considered to belong to this declarative region in spite of its being hidden during qualified and
   unqualified name lookup.) — end example ]

3 The potential scope of a template parameter name begins at its point of declaration (3.3.2) and ends at the
   end of its declarative region. [ Note: this implies that a template-parameter can be used in the declaration
   of subsequent template-parameters and their default arguments but cannot be used in preceding template-
   parameters or their default arguments. For example,
   template<class T, T* p, class U = T> class X { /* ... */ };
   template<class T> void f(T* p = new T);
This also implies that a template-parameter can be used in the specification of base classes. For example,

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

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

4 The declarative region of the name of a template parameter is nested within the immediately-enclosing declarative region. [Note: as a result, a template-parameter hides any entity with the same name in an enclosing scope (3.3.10). [Example:

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

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

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

### 3.3.10 Name hiding [basic.scope.hiding]

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

2 A class name (9.1) or enumeration name (7.2) can be hidden by the name of a variable, data member, function, or enumerator declared in the same scope. If a class or enumeration name and a variable, data member, function, or enumerator are declared in the same scope (in any order) with the same name, the class or enumeration name is hidden wherever the variable, data member, function, or enumerator name is visible.

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

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

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

### 3.4 Name lookup [basic.lookup]

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

§ 3.4
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 [basic.lookup.unqual]

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 &);
    operator int();
    void g(A a) {
      int i = f(a); // f is the typedef, not the friend
      // function: equivalent to int(a)
    }
  };
}
```

Because the expression is not a function call, the argument-dependent name lookup (3.4.2) does not apply and the friend function f is not found. — end note]

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[^28] that is a member of namespace N (where, only for the purpose of exposition, N could represent the global scope) shall be declared before its use in the block in which it is used or in one of its enclosing blocks (6.3) or, shall be declared before its use in namespace N or, if N is a nested namespace, shall be declared before its use in one of N’s enclosing namespaces. [Example:

```c
namespace A {
  namespace N {
```}

[^28]: 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.
void f();
{
}
void A::N::f() {
    i = 5;
    // The following scopes are searched for a declaration of i:
    // 1) outermost block scope of A::N::f, before the use of i
    // 2) scope of namespace N
    // 3) scope of namespace A
    // 4) global scope, before the definition of A::N::f
}

— end example]

7 A name used in the definition of a class \( X \) outside of a member function body or nested class definition\(^{29}\) 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);\(^{30}\) or
— if \( X \) is a local class (9.8) or is a nested class of a local class, before the definition of class \( X \) in a block enclosing the definition of class \( X \), or
— if \( X \) is a member of namespace \( N \), or is a nested class of a class that is a member of \( N \), or is a local class or a nested class within a local class of a function that is a member of \( N \), before the definition of class \( X \) in namespace \( N \) or in one of \( N \)'s enclosing namespaces.

[Example:

namespace M {
    class B {
    }
}
namespace N {
    class Y : public M::B {
        class X {
            int a[i];
        }
    }
// The following scopes are searched for a declaration of i:
// 1) scope of class N::Y::X, before the use of i
// 2) scope of class N::Y, before the definition of N::Y::X
// 3) scope of N::Y's base class M::B
// 4) scope of namespace N, before the definition of N::Y
// 5) global scope, before the definition of N

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

\(^{30}\) This lookup applies whether the definition of \( X \) is nested within \( Y \)'s definition or whether \( X \)'s definition appears in a namespace scope enclosing \( Y \)'s definition (9.7).
8 A name used in the definition of a member function (9.3) of class X following the function’s declarator-id 31 or in the brace-or-equal-initializer of a non-static data member (9.2) of class X 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), 32 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 use of the name, in namespace N or in one of N’s enclosing namespaces.

[Example:

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

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

— end example] [Note: 9.3 and 9.4 further describe the restrictions on the use of names in member function definitions. 9.7 further describes the restrictions on the use of names in the scope of nested classes. 9.8 further describes the restrictions on the use of names in local class definitions. — end note]

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

31) 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.
32) This lookup applies whether the member function is defined within the definition of class X or whether the member function is defined in a namespace scope enclosing X’s definition.
not defined in the class granting friendship, name lookup in the friend function definition shall proceed as described for lookup in namespace member function definitions.

10 In a friend declaration naming a member function, a name used in the function declarator and not part of a template-argument in the declarator-id is first looked up in the scope of the member function’s class (10.2). If it is not found, or if the name is part of a template-argument in the declarator-id, the look up is as described for unqualified names in the definition of the class granting friendship. [Example:

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

11 During the lookup for a name used as a default argument (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.

13 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]

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

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

int i = 2;

int N::j = i; // N::j == 4
— end example]
```

15 A name used in the handler for a function-try-block (Clause 15) is looked up as if the name was used in the outermost block of the function definition. In particular, the function parameter names shall not be redeclared in the exception-declaration nor in the outermost block of a handler for the function-try-block.
Names declared in the outermost block of the function definition are not found when looked up in the scope of a handler for the function-try-block. [Note: but function parameter names are found. — end note]

[Note: the rules for name lookup in template definitions are described in 14.6. — end note]

3.4.2 Argument-dependent name lookup [basic.lookup.argdep]

1 When the postfix-expression in a function call (5.2.2) is an unqualified-id, other namespaces not considered during the usual unqualified lookup (3.4.1) may be searched, and in those namespaces, namespace-scope friend function 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). [Example:

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

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

2 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 namespces and classes are those associated with the function parameter types and those associated with the return type.

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

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

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

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

— Any using-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 denotes its class, namespace, or enumeration. If a :: scope resolution operator in a nested-name-specifier is not preceded by a decltype-specifier, lookup of the name preceding that :: considers only namespaces, types, and templates whose specializations are types. If the name found does not designate a namespace or a class, enumeration, or dependent type, the program is ill-formed. [Example:
class A {
public:
    static int n;
};
int main() {
    int A;
    A::n = 42;  // OK
    A b;        // ill-formed: A does not name a type
}

— end example

[Note: multiply qualified names, such as N1::N2::N3::n, can be used to refer to members of nested classes (9.7) or members of nested namespaces. — end note]

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

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

    — end example]

4 A name prefixed by the unary scope operator :: (5.1) is looked up in global scope, in the translation unit where it is used. The name shall be declared in global namespace scope or shall be a name whose declaration is visible in global scope because of a using-directive (3.4.3.2). The use of :: allows a global name to be referred to even if its identifier has been hidden (3.3.10).

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

6 If a pseudo-destructor-name (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:

    ::opt nested-name-specifier opt class-name :: ~ class-name

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

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

    struct A {
        "A();

§ 3.4.3
typedef A AB;
int main() {
    AB *p;
    p->AB::~AB(); // explicitly calls the destructor for A
}

— end example] [Note: 3.4.5 describes how name lookup proceeds after the . and -> operators. — end note]

3.4.3.1 Class members

1 If the nested-name-specifier of a qualified-id nominates a class, the name specified after the nested-name-specifier is looked up in the scope of the class (10.2), except for the cases listed below. The name shall represent one or more members of that class or of one of its base classes (Clause 10). [Note: a class member can be referred to using a qualified-id at any point in its potential scope (3.3.7). — 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 a conversion-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 (3.3.10).

2 In a lookup in which the constructor is an acceptable lookup result 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
   — in a using-declaration (7.3.3) that is a member-declaration, if the name specified after the nested-name-specifier is the same as the identifier or the simple-template-id’s template-name in the last component of the nested-name-specifier,

   the name is instead considered to name the constructor of class C. [Note: for example, the constructor is not an acceptable lookup result in an elaborated-type-specifier so the constructor would not be used in place of the injected-class-name. — end note] Such a constructor name shall be used only in the declarator-id of a declaration that names a constructor or in a using-declaration. [Example:

        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::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 For a namespace $X$ and name $m$, the namespace-qualified lookup set $S(X, m)$ is defined as follows: Let $S'(X, m)$ be the set of all declarations of $m$ in $X$ and the inline namespace set of $X$ (7.3.1). If $S'(X, m)$ is not empty, $S(X, m)$ is $S'(X, m)$; otherwise, $S(X, m)$ is the union of $S(N_i, m)$ for all namespaces $N_i$ nominated by using-directives in $X$ and its inline namespace set.

3 Given $X::m$ (where $X$ is a user-declared namespace), or given $::m$ (where $X$ is the global namespace), if $S(X, m)$ is the empty set, the program is ill-formed. Otherwise, if $S(X, m)$ has exactly one member, or if the context of the reference is a using-declaration (7.3.3), $S(X, m)$ is the required set of declarations of $m$. Otherwise if the use of $m$ is not one that allows a unique declaration to be chosen from $S(X, m)$, the program is ill-formed. [Example:

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

namespace Z {
  void h(double);
}

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

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

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

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

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

```cpp
namespace A {
  int a;
}
namespace B {
  using namespace A;
}
namespace C {
  using namespace A;
}
namespace BC {
  using namespace B;
  using namespace C;
}
void f() {
  BC::a++; // OK: S is { A::a, A::a }
}
namespace D {
  using A::a;
}
namespace BD {
  using namespace B;
  using namespace D;
}
void g() {
  BD::a++; // OK: S is { A::a, A::a }
}
```

§ 3.4.3.2
Because each referenced namespace is searched at most once, the following is well-defined:

```cpp
namespace B {
    int b;
}

namespace A {
    using namespace B;
    int a;
}

namespace B {
    using namespace A;
}

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

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

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

namespace B {
    struct y { }; 
}

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

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

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

the unqualified-id shall name a member of the namespace designated by the nested-name-specifier or of an element of the inline namespace set (7.3.1) of that namespace. [Example:
namespace A {
    namespace B {
        void f1(int);
    }  // ill-formed, f1 is not a member of A
    using namespace B;
}

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 (3.3.10).

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

    class-key attribute-specifier_opt identifier;

the identifier is looked up according to 3.4.1 but ignoring any non-type names that have been declared. If the elaborated-type-specifier is introduced by the enum keyword and this lookup does not find a previously declared type-name, the elaborated-type-specifier is ill-formed. If the elaborated-type-specifier is introduced by the class-key and this lookup does not find a previously declared type-name, or if the elaborated-type-specifier appears in a declaration with the form:

    class-key attribute-specifier_opt identifier;

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

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
    }

§ 3.4.4
3.4.5 Class member access

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

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

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

— end example]

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

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

the class-name-or-namespace-name following the . or -> operator is 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

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

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

6 If the nested-name-specifier contains a simple-template-id (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.

```
translation-unit:
    declaration-seq_opt
```

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.

A name having namespace scope (3.3.6) has internal linkage if it is the name of

- a variable, function or function template that is explicitly declared static; or,
— a variable 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
— a variable, 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 a variable declared by a block scope `extern` declaration have linkage. If there is a visible declaration of an entity with linkage having the same name and type, ignoring entities declared outside the innermost enclosing namespace scope, the block scope declaration declares that same entity and receives the linkage of the previous declaration. If there is more than one such matching entity, the program is ill-formed. Otherwise, if no matching entity is found, the block scope entity receives external linkage. [Example:

```
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:

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

§ 3.5
8 Names not covered by these rules have no linkage. Moreover, except as noted, a name declared at block scope (3.3.3) has no linkage. A type is said to have linkage if and only if:

- 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 an unnamed class or enumeration member of a class with linkage; or
- it is a specialization of a class template (14); 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 external linkage unless

- the entity has C language linkage (7.5), or
- the entity is declared within an unnamed namespace (7.3.1), or
- the entity is not used (3.2) or is defined in the same translation unit.

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

[Example:

template <class T> struct B {
    void g(T) { }
    void h(T);
    friend void i(B, T) { }
};

void f() {
    struct A { int x; }; // no linkage
    A a = { 1 };  // declares B<A>::g(A) and B<A>::h(A)
    B<A> ba;       // declares B<A>::g(A) and B<A>::h(A)
    ba.g(a);       // OK
}

33) A class template always has external linkage, and the requirements of 14.3.1 and 14.3.2 ensure that the template arguments will also have appropriate linkage.

§ 3.5
Two names that are the same (Clause 3) and that are declared in different scopes shall denote the same variable, function, type, enumerator, template or namespace if

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

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

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

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

After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given variable or function shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (8.3.4). A violation of this rule on type identity does not require a diagnostic.

[Note: linkage to non-C++ declarations can be achieved using a linkage-specification (7.5). — end note]

### 3.6 Start and termination

#### 3.6.1 Main function

A program shall contain a global function called main, which is the designated start of the program. It is implementation-defined whether a program in a freestanding environment is required to define a main function. [Note: in a freestanding environment, start-up and termination is implementation-defined; start-up contains the execution of constructors for objects of namespace scope with static storage duration; termination contains the execution of destructors for objects with static storage duration. — end note]

An implementation shall not redefine the main function. This function shall not be overloaded. It shall have a 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) (17.5.2.1.4.2) and argv[0] shall be the pointer to the initial character of a NTMBS that represents the name used to invoke the program or ".". The value of argc shall be non-negative. The value of argv[argc] shall be 0. [Note: it is recommended that any further (optional) parameters be added after argv. — end note]

The function main shall not be used (3.2) within a program. The linkage (3.5) of main is implementation-defined. A program that defines main as deleted or that declares main to be inline, static, or constexpr
is ill-formed. The name `main` is not otherwise reserved. [Example: member functions, classes, and enumerations can be called `main`, as can entities in other namespaces. —end example]

4 Terminating the program without leaving the current block (e.g., by calling the function `std::exit(int)` (18.5)) does not destroy any objects with automatic storage duration (12.4). If `std::exit` is called to end a program during the destruction of an object with static or thread storage duration, the program has undefined behavior.

5 A return statement in `main` has the effect of leaving the main function (destroying any objects with automatic storage duration) and calling `std::exit` with the return value as the argument. If control reaches the end of `main` without encountering a `return` statement, the effect is that of executing
   ```
   return 0;
   ```

3.6.2 Initialization of non-local variables [basic.start.init]

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

2 Variables with static storage duration (3.7.1) or thread storage duration (3.7.2) shall be zero-initialized (8.5) before any other initialization takes place.

   Constant initialization is performed:

   - if each full-expression (including implicit conversions) that appears in the initializer of a reference with static or thread storage duration is a constant expression (5.19) and the reference is bound to an lvalue designating an object with static storage duration or to a temporary (see 12.2)
   - if an object with static or thread storage duration is initialized such that the initialization satisfies the requirements for the object being declared with `constexpr` (7.1.5).

   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 a non-local variable with static storage duration 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 non-local variables with static storage duration have ordered initialization. Variables with ordered initialization defined within a single translation unit shall be initialized in the order of their definitions in the translation unit. If a program starts a thread (30.3), the subsequent initialization of a variable is indeterminately sequenced with respect to the initialization of a variable defined in a different translation unit. Otherwise, the initialization of a variable is indeterminately sequenced with respect to the initialization of a variable defined in a different translation unit. If a program starts a thread, the subsequent unordered initialization of a variable is unsequenced with respect to every other dynamic initialization. Otherwise, the unordered initialization of a variable is indeterminately sequenced with respect to every other dynamic initialization. [Note: This definition permits initialization of a sequence of ordered variables concurrently with another sequence. —end note] [Note: The initialization of local static variables is described in 6.7. —end note]

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

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

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

```c
inline double fd() { return 1.0; }

extern double d1;

double d2 = d1;  // unspecified:
    // may be statically initialized to 0.0 or
    // dynamically initialized to 1.0

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

— end note]

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

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

— end example

5 It is implementation-defined whether the dynamic initialization of a non-local variable with static or thread storage duration is done before the first statement of the initial function of the thread. If the initialization

§ 3.6.2
is deferred to some point in time after the first statement of the initial function of the thread, it shall occur before the first use of any variable with thread storage duration defined in the same translation unit as the variable to be initialized.

3.6.3 Termination

Destructors (12.4) for initialized objects (that is, objects whose lifetime (3.8) has begun) with static storage duration are called as a result of returning from main and as a result of calling std::exit (18.5). Destructors for initialized objects with thread storage duration within a given thread are called as a result of returning from the initial function of that thread and as a result of that thread calling std::exit. The completions of the destructors for all initialized objects with thread storage duration within that thread are sequenced before the initiation of the destructors of any object with static storage duration. If the completion of the constructor or dynamic initialization of an object with thread storage duration is sequenced before that of another, the completion of the destructor of the second is sequenced before the initiation of the destructor of the first. If the completion of the constructor or dynamic initialization of an object with static storage duration is sequenced before that of another, the completion of the destructor of the second is sequenced before the initiation of the destructor of the first. [Note: this definition permits concurrent destruction. — end note] If an object is initialized statically, the object is destroyed in the same order as if the object was dynamically initialized. For an object of array or class type, all subobjects of that object are destroyed before any block-scope object with static storage duration initialized during the construction of the subobjects is destroyed. [Note: If the destruction of a non-local object with static or thread storage duration terminates by throwing an exception, std::terminate is called (see 15.5.1). — end note]

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

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

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

3.7 Storage duration

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

- static storage duration
- thread storage duration
- automatic storage duration
- dynamic storage duration

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

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

### 3.7.1 Static storage duration

All variables which do not have dynamic storage duration, do not have thread storage duration, and are not local have static storage duration. The storage for these entities shall last for the duration of the program (3.6.2, 3.6.3).

If a variable with static storage duration has initialization or a destructor with side effects, it shall not be eliminated even if it appears to be unused, except that a class object or its copy/move may be eliminated as specified in 12.8.

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

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

### 3.7.2 Thread storage duration

All variables declared with the `thread_local` keyword have thread storage duration. The storage for these entities shall last for the duration of the thread in which they are created. There is a distinct object or reference per thread, and use of the declared name refers to the entity associated with the current thread.

A variable with thread storage duration shall be initialized before its first use and, if constructed, shall be destroyed on thread exit.

### 3.7.3 Automatic storage duration

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

[Note: these variables are initialized and destroyed as described in 6.7. — end note]

If a variable with automatic storage duration has initialization or a destructor with side effects, it shall not be destroyed before the end of its block, nor shall it be eliminated as an optimization even if it appears to be unused, except that a class object or its copy/move may be eliminated as specified in 12.8.

### 3.7.4 Dynamic storage duration

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

2 The library provides default definitions for the global allocation and deallocation functions. Some global allocation and deallocation functions are replaceable (18.6.1). A C++ program shall provide at most one definition of a replaceable allocation or deallocation function. Any such function definition replaces the default version provided in the library (17.6.3.6). The following allocation and deallocation functions (18.6) are implicitly declared in global scope in each translation unit of a program.

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

These implicit declarations introduce only the function names \texttt{operator new}, \texttt{operator new[]}, \texttt{operator delete}, \texttt{operator delete[]}. \[ Note: \] the implicit declarations do not introduce the names \texttt{std}, \texttt{std::bad_alloc}, and \texttt{std::size_t}, or any other names that the library uses to declare these names. Thus, a \texttt{new-expression}, \texttt{delete-expression} or function call that refers to one of these functions without including the header \texttt{<new>} is well-formed. However, referring to \texttt{std}, \texttt{std::bad_alloc}, and \texttt{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 3.7.4.1 and 3.7.4.2.

### 3.7.4.1 Allocation functions

[\texttt{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 \texttt{void*}. The first parameter shall have type \texttt{std::size_t} (18.2). The first parameter shall not have an associated default argument (8.3.6). The value of the first parameter shall be interpreted as the requested size of the allocation. An allocation function can be a function template. Such a template shall declare its return type and first parameter as specified above (that is, template parameter types shall not be used in the return type and first parameter type). Template allocation functions shall have two or more parameters.

2 The allocation function attempts to allocate the requested amount of storage. If it is successful, it shall return the address of the start of a block of storage whose length in bytes shall be at least as large as the requested size. There are no constraints on the contents of the allocated storage on return from the allocation function. The order, contiguity, and initial value of storage allocated by successive calls to an allocation function are unspecified. The pointer returned shall be suitably aligned so that it can be converted to a pointer of any complete object type with a fundamental alignment requirement (3.11) and then used to access the object or array in the storage allocated (until the storage is explicitly deallocated by a call to a corresponding deallocation function). Even if the size of the space requested is zero, the request can fail. If the request succeeds, the value returned shall be a non-null pointer value (4.10) \(p0\) different from any previously returned value \(p1\), unless that value \(p1\) was subsequently passed to an \texttt{operator delete}. The effect of dereferencing a pointer returned as a request for zero size is undefined.\[35\]

3 An allocation function that fails to allocate storage can invoke the currently installed new-handler function (18.6.2.3), if any. \[ Note: \] A program-supplied allocation function can obtain the address of the currently installed \texttt{new_handler} using the \texttt{std::set_new_handler} function (18.6.2.4). \[ end note \] If an allocation function declared with a non-throwing \texttt{exception-specification} (15.4) fails to allocate storage, it shall return

\[35\] The intent is to have \texttt{operator new()} implementable by calling \texttt{std::malloc()} or \texttt{std::calloc()}, so the rules are substantially the same. C++ differs from C in requiring a zero request to return a non-null pointer.
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` (18.6.2.1).

4 A global allocation function is only called as the result of a new expression (5.3.4), or called directly using the
function call syntax (5.2.2), or called indirectly through calls to the functions in the C++ standard library.

[Note: in particular, a global allocation function is not called to allocate storage for objects with static
storage duration (3.7.1), for objects or references with thread storage duration (3.7.2), for objects of type
`std::type_info` (5.2.8), or for the copy of an object thrown by a `throw` expression (15.1). — end note]

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

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

2 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` (18.2), 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.

3 If a deallocation function terminates by throwing an exception, the behavior is undefined. The value of the
first argument supplied to a deallocation function may be a null pointer value; if so, and if the deallocation
function is one supplied in the standard library, the call has no effect. 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.

4 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.36

### 3.7.4.3 Safely-derived pointers [basic.stc.dynamic.safety]

1 A `traceable pointer object` is

---

36) On some implementations, it causes a system-generated runtime fault.
2 A pointer value is a \textit{safely-derived pointer} to a dynamic object only if it has pointer-to-object type and it is one of the following:

- the value returned by a call to the C++ standard library implementation of \texttt{::operator new(std::size\_t)};\textsuperscript{37}
- the result of taking the address of an object (or one of its subobjects) designated by an lvalue resulting from dereferencing a safely-derived pointer value;
- the result of well-defined pointer arithmetic using a safely-derived pointer value;
- the result of a well-defined pointer conversion of a safely-derived pointer value;
- the result of a \texttt{reinterpret\_cast} of a safely-derived pointer value;
- the result of a \texttt{reinterpret\_cast} of an integer representation of a safely-derived pointer value;
- the value of an object whose value was copied from a traceable pointer object, where at the time of the copy the source object contained a copy of a safely-derived pointer value.

3 An integer value is an \textit{integer representation of a safely-derived pointer} only if its type is at least as large as \texttt{std::intptr\_t} and it is one of the following:

- the result of a \texttt{reinterpret\_cast} of a safely-derived pointer value;
- the result of a valid conversion of an integer representation of a safely-derived pointer value;
- the value of an object whose value was copied from a traceable pointer object, where at the time of the copy the source object contained an integer representation of a safely-derived pointer value;
- the result of an additive or bitwise operation, one of whose operands is an integer representation of a safely-derived pointer value \(P\), if that result converted by \texttt{reinterpret\_cast\_void\_t} would compare equal to a safely-derived pointer computable from \texttt{reinterpret\_cast\_void\_t}(\(P\)).

4 An implementation may have \textit{relaxed pointer safety}, in which case the validity of a pointer value does not depend on whether it is a safely-derived pointer value. Alternatively, an implementation may have \textit{strict pointer safety}, in which case, if a pointer value that is not a safely-derived pointer value is dereferenced or deallocated, and the referenced complete object is of dynamic storage duration and has not previously been declared reachable (20.9.12), the behavior is undefined. [\textit{Note: this is true even if the unsafely-derived pointer value might compare equal to some safely-derived pointer value. — end note}] It is implementation defined whether an implementation has relaxed or strict pointer safety.

3.7.5 Duration of subobjects \[basic.stc.inherit]\n
1 The storage duration of member subobjects, base class subobjects and array elements is that of their complete object (1.8).

3.8 Object lifetime \[basic.life]\n
1 The \textit{lifetime} of an object is a runtime property of the object. An object is said to have non-trivial initialization if it is of a class or aggregate type and it or one of its members is initialized by a constructor other than a trivial default constructor. [\textit{Note: initialization by a trivial copy/move constructor is non-trivial initialization. — end note}] The lifetime of an object of type \(T\) begins when:

- storage with the proper alignment and size for type \(T\) is obtained, and

\textsuperscript{37} This section does not impose restrictions on dereferencing pointers to memory not allocated by \texttt{::operator new}. This maintains the ability of many C++ implementations to use binary libraries and components written in other languages. In particular, this applies to C binaries, because dereferencing pointers to memory allocated by \texttt{malloc} is not restricted.
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 starts as soon as storage with proper size and alignment is obtained, and its lifetime ends when the storage which the array occupies is reused or released. 12.6.2 describes the lifetime of base and member subobjects. — end note]

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

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

Before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that refers to the storage location where the object will be or was located may be used but only in limited ways. For an object under construction or destruction, see 12.7. Otherwise, such a pointer refers to allocated storage (3.7.4.2), and using the pointer as if the pointer were of type `void*`, is well-defined. Such a pointer may be dereferenced but the resulting lvalue may only be used in limited ways, as described below. The program has undefined behavior if:

- the object will be or was of a class type with a non-trivial destructor and the pointer is used as the operand of a delete-expression,
- the pointer is used to access a non-static data member or call a non-static member function of the object, or
- the pointer is implicitly converted (4.10) to a pointer to a base class type, or
- the pointer is used as the operand of a static_cast (5.2.9) (except when the conversion is to void*, or to void* and subsequently to char*, or unsigned char*), or
- the pointer is used as the operand of a dynamic_cast (5.2.7). [Example:

```
#include <cstdlib>

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

38) 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]

Similarly, before the lifetime of an object has started but after the storage which the object will occupy
has been allocated or, after the lifetime of an object has ended and before the storage which the object
occupied is reused or released, any glvalue that refers to the original object may be used but only in limited
ways. For an object under construction or destruction, see 12.7. Otherwise, such a glvalue refers to allocated
storage (3.7.4.2), and using the properties of the glvalue that do not depend on its value is well-defined. The
program has undefined behavior if:

— an lvalue-to-rvalue conversion (4.1) is applied to such a glvalue,
— the glvalue is used to access a non-static data member or call a non-static member function of the
  object, or
— the glvalue is implicitly converted (4.10) to a reference to a base class type, or
— the glvalue 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 glvalue is used as the operand of a dynamic_cast (5.2.7) or as the operand of typeid.

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:

struct C {
  int i;

§ 3.8
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), thread (3.7.2), or automatic (3.7.3) storage duration and if T has a non-trivial destructor, the program must ensure that an object of the original type occupies that same storage location when the implicit destructor call takes place; otherwise the behavior of the program is undefined. This is true even if the block is exited with an exception. [ Example:

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

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

const B b;

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

39) That is, an object for which a destructor will be called implicitly—upon exit from the block for an object with automatic storage duration, upon exit from the thread for an object with thread storage duration, or upon exit from the program for an object with static storage duration.
In this section, “before” and “after” refer to the “happens before” relation (1.10). [Note: Therefore, undefined behavior results if an object that is being constructed in one thread is referenced from another thread without adequate synchronization. —end note]

### 3.9 Types [basic.types]

1 [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).]

2 For any object (other than a base-class subobject) of trivially copyable type \( T \), whether or not the object holds a valid value of type \( T \), the underlying bytes (1.7) making up the object can be copied into an array of \texttt{char} or \texttt{unsigned char}.\(^{40}\) If the content of the array of \texttt{char} or \texttt{unsigned char} is copied back into the object, the object shall subsequently hold its original value. [Example:

```c
#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 trivially copyable type \( T \), if two pointers to \( T \) point to distinct \( T \) objects \( \text{obj1} \) and \( \text{obj2} \), where neither \( \text{obj1} \) nor \( \text{obj2} \) is a base-class subobject, if the underlying bytes (1.7) making up \( \text{obj1} \) are copied into \( \text{obj2} \),\(^{41}\) \( \text{obj2} \) shall subsequently hold the same value as \( \text{obj1} \). [Example:

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

—end example]

4 The object representation of an object of type \( T \) is the sequence of \( N \) \texttt{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 trivially copyable types, the value representation is a set of bits in the object representation that determines a value, which is one discrete element of an implementation-defined set of values.\(^{42}\)

5 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.\(^{43}\) Incompletely-defined object types and the void types are incomplete types (3.9.1). Objects shall not be defined to have an incomplete type.

6 A class type (such as “class \( X \)”) might be incomplete at one point in a translation unit and complete later on; the type “class \( X \)” is the same type at both points. The declared type of an array object might be

---

\(^{40}\) By using, for example, the library functions (17.6.1.2) `std::memcpy` or `std::memmove`.  
\(^{41}\) By using, for example, the library functions (17.6.1.2) `std::memcpy` or `std::memmove`.  
\(^{42}\) The intent is that the memory model of C++ is compatible with that of ISO/IEC 9899 Programming Language C.  
\(^{43}\) The size and layout of an instance of an incompletely-defined object type is unknown.
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:

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

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

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

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

— end example]

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

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

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

10 A type is a literal type if it is:
   — a scalar type; or
   — a class type (Clause 9) with
     — a trivial copy constructor,
     — no non-trivial move constructor,
— a trivial destructor,
— a trivial default constructor or at least one constexpr constructor other than the copy or move constructor, 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

1 Objects declared as characters (char) shall be large enough to store any member of the implementation’s basic character set. If a character from this set is stored in a character object, the integral value of that character object is equal to the value of the single character literal form of that character. It is implementation-defined whether a char object can hold negative values. Characters can be explicitly declared *unsigned or signed*. Plain char, signed char, and unsigned char are three distinct types. A char, a signed char, and an unsigned char occupy the same amount of storage and have the same alignment requirements (3.11); that is, they have the same object representation. For character types, all bits of the object representation participate in the value representation. For unsigned character types, all possible bit patterns of the value representation represent numbers. These requirements do not hold for other types. In any particular implementation, a plain char object can take on either the same values as a signed char or an unsigned char; which one is implementation-defined.

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

3 For each of the standard signed integer types, there exists a corresponding (but different) *standard unsigned integer type*: “unsigned char”, “unsigned short int”, “unsigned int”, “unsigned long int”, and “unsigned long long int”, each of which occupies the same amount of storage and has the same alignment requirements (3.11) as the corresponding signed integer type; that is, each signed integer type has the same object representation as its corresponding unsigned integer type. Likewise, for each of the extended signed integer types there exists a corresponding extended unsigned integer type with the same amount of storage and alignment requirements. The standard and extended unsigned integer types are collectively called *unsigned integer types*. The range of non-negative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the value representation of each corresponding signed/unsigned type shall be the same. The standard signed integer types and standard unsigned integer types are collectively called the *standard integer types*, and the extended signed integer types and extended unsigned integer types are collectively called the *extended integer types*.

4 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.

---

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

45) See 7.1.6.2 regarding the correspondence between types and the sequences of type-specifiers that designate them.

46) This implies that unsigned arithmetic does not overflow because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting unsigned integer type.
Type \texttt{wchar\_t} is a distinct type whose values can represent distinct codes for all members of the largest extended character set specified among the supported locales (22.3.1). Type \texttt{wchar\_t} shall have the same size, signedness, and alignment requirements (3.11) as one of the other integral types, called its \textit{underlying type}. Types \texttt{char16\_t} and \texttt{char32\_t} denote distinct types with the same size, signedness, and alignment as \texttt{uint\_least16\_t} and \texttt{uint\_least32\_t}, respectively, in <\texttt{stdint.h}>, called the underlying types.

Values of type \texttt{bool} are either \texttt{true} or \texttt{false}.\footnote{Using a \texttt{bool} value in ways described by this International Standard as “undefined,” such as by examining the value of an uninitialized automatic object, might cause it to behave as if it is neither \texttt{true} nor \texttt{false}.} [\textit{Note:} there are no \texttt{signed}, \texttt{unsigned}, \texttt{short}, or \texttt{long} \texttt{bool} types or values. — end note] Values of type \texttt{bool} participate in integral promotions (4.5).

Types \texttt{bool}, \texttt{char}, \texttt{char16\_t}, \texttt{char32\_t}, \texttt{wchar\_t}, and the signed and unsigned integer types are collectively called \textit{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 \textit{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 \textit{American National Dictionary for Information Processing Systems}.)} [\textit{Example:} this International Standard permits 2’s complement, 1’s complement and signed magnitude representations for integral types. — end example]

There are three \textit{floating point} types: \texttt{float}, \texttt{double}, and \texttt{long\ double}. The type \texttt{double} provides at least as much precision as \texttt{float}, and the type \texttt{long\ double} provides at least as much precision as \texttt{double}. The set of values of the type \texttt{float} is a subset of the set of values of the type \texttt{double}; the set of values of the type \texttt{double} is a subset of the set of values of the type \texttt{long\ double}. The value representation of floating-point types is implementation-defined. \textit{Integral} and \textit{floating} types are collectively called \textit{arithmetic} types. Specializations of the standard template std::numeric\_limits (18.3) shall specify the maximum and minimum values of each arithmetic type for an implementation.

The \texttt{void} type has an empty set of values. The \texttt{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 \textit{cv void} (5.4). An expression of type \texttt{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 \texttt{?:} (5.16), as the operand of \texttt{typeid}, or as the expression in a return statement (6.6.3) for a function with the return type \texttt{void}.

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

[\textit{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

[\texttt{basic.compound}]

Compound types can be constructed in the following ways:

- \textit{arrays} of objects of a given type, 8.3.4;
- \textit{functions}, which have parameters of given types and return \texttt{void} or references or objects of a given type, 8.3.5;
- \textit{pointers} to \texttt{void} or objects or functions (including static members of classes) of a given type, 8.3.1;
- \textit{references} to objects or functions of a given type, 8.3.2. There are two types of references:
  - \textit{lvalue reference}
  - \textit{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 class members, which identify members of a given type within objects of a given class, 8.3.3.

2 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.

3 A pointer to objects of type T is referred to as a “pointer to T.” [Example: a pointer to an object of type int is referred to as “pointer to int” and a pointer to an object of class X is called a “pointer to X.” — end example] Except for pointers to static members, text referring to “pointers” does not apply to pointers to members. Pointers to incomplete types are allowed although there are restrictions on what can be done with them (3.11). A valid value of an object pointer type represents either the address of a byte in memory (1.7) or a null pointer (4.10). If an object of type T is located at an address A, a pointer of type cv T* whose value is the address A is said to point to that object, regardless of how the value was obtained. [Note: for instance, the address one past the end of an array (5.7) would be considered to point to an unrelated object of the array’s element type that might be located at that address. There are further restrictions on pointers to objects with dynamic storage duration; see 3.7.4.3. — end note] The value representation of pointer types is implementation-defined. Pointers to 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 (3.11) have no special representation, but their range of valid values is restricted by the extended alignment requirement. This International Standard specifies only two ways of obtaining such a pointer: taking the address of a valid object with an over-aligned type, and using one of the runtime pointer alignment functions. An implementation may provide other means of obtaining a valid pointer value for an over-aligned type. — end note]

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*.

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).51

50) Static class members are objects or functions, and pointers to them are ordinary pointers to objects or functions.  
51) The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and non-static data members of unions.
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).

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 function types that have cv-qualifiers.

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

Table 8 — Relations on const and volatile

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

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 \{\texttt{const}\}, \{\texttt{volatile}\}, \{\texttt{const, volatile}\}, or the empty set. Cv-qualifiers applied to an array type attach to the underlying element type, so the notation “cv T,” where T is an array type, refers to an array whose elements are so-qualified. Such array types can be said to be more (or less) cv-qualified than other types based on the cv-qualification of the underlying element types.

### 3.10 Lvalues and rvalues [basic.lval]

Expressions are categorized according to the taxonomy in Figure 1.

![Expression category taxonomy](image)

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

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

— A glvalue (“generalized” lvalue) is an lvalue or an xvalue.
— An *rvalue* (so called, historically, because rvalues could appear on the right-hand side of an assignment expressions) is an xvalue, a temporary object (12.2) or subobject thereof, or a value that is not associated with an object.

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

Every expression belongs to exactly one of the fundamental classifications in this taxonomy: lvalue, xvalue, or prvalue. This property of an expression is called its *value category*. [Note: The discussion of each built-in operator in Clause 5 indicates the category of the value it yields and the value categories of the operands it expects. For example, the built-in assignment operators expect that the left operand is an lvalue and that the right operand is a prvalue and yield an lvalue as the result. User-defined operators are functions, and the categories of values they expect and yield are determined by their parameter and return types. — end note]

1. Whenever a glvalue appears in a context where a prvalue is expected, the glvalue is converted to a prvalue; see 4.1, 4.2, and 4.3.

2. 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.

3. Class prvalues can have cv-qualified types; non-class prvalues always have cv-unqualified types. Prvalues shall always have complete types or the void type; in addition to these types, glvalues can also have incomplete types.

4. 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]

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

6. 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.

7. 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).

8. 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.

9. If a program attempts to access the stored value of an object through a glvalue of other than one of the following types the behavior is undefined52

   — the dynamic type of the object,
   — a cv-qualified version of the dynamic type of the object,
   — a type similar (as defined in 4.4) to the dynamic type of the object,
   — a type that is the signed or unsigned type corresponding to the dynamic type of the object,
   — a type that is the signed or unsigned type corresponding to a cv-qualified version of the dynamic type of the object,
   — an aggregate or union type that includes one of the aforementioned types among its elements or non-static data members (including, recursively, an element or non-static data member of a subaggregate or contained union),

52) The intent of this list is to specify those circumstances in which an object may or may not be aliased.
— a type that is a (possibly cv-qualified) base class type of the dynamic type of the object,
— a char or unsigned char type.

### 3.11 Alignment

Object types have alignment requirements (3.9.1, 3.9.2) which place restrictions on the addresses at which an object of that type may be allocated. An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type; stricter alignment can be requested using the alignment attribute (7.6.2).

A fundamental alignment is represented by an alignment less than or equal to the greatest alignment supported by the implementation in all contexts, which is equal to `alignof(std::max_align_t)` (18.2).

An extended alignment is represented by an alignment greater than `alignof(std::max_align_t)`. It is implementation-defined whether any extended alignments are supported and the contexts in which they are supported (7.6.2). A type having an extended alignment requirement is an over-aligned type. [Note: every over-aligned type is or contains a class type 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.53

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 (7.6.2). — end note]

Comparing alignments is meaningful and provides the obvious results:

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

[Note: the runtime pointer alignment function (20.9.13) can be used to obtain an aligned pointer within a buffer; the aligned-storage templates in the library (20.7.6.6) can be used to obtain aligned storage. — end note]

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

53 It is intended that every valid alignment value be an integral power of two.
4 Standard conversions [conv]

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 or an rvalue reference to function type (8.3.2), an xvalue if T is an rvalue reference to object type, and a prvalue otherwise. The expression e is used as a glvalue if and only if the initialization uses it as a glvalue.

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

1 A glvalue (3.10) of a non-function, non-array type \( T \) can be converted to a prvalue.\(^{54} \) If \( T \) is an incomplete type, a program that necessitates this conversion is ill-formed. If the object to which the glvalue 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 prvalue is the cv-unqualified version of \( T \). Otherwise, the type of the prvalue is \( T \).\(^{55} \)

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 glvalue has a class type, the conversion copy-initializes a temporary of type \( T \) from the glvalue and the result of the conversion is a prvalue for the temporary. Otherwise, if the glvalue has (possibly cv-qualified) type \( \text{std}::\text{nullptr}_t \), the prvalue result is a null pointer constant (4.10). Otherwise, the value contained in the object indicated by the glvalue is the prvalue result.

[Note: See also 3.10. — end note]

4.2 Array-to-pointer conversion

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

4.3 Function-to-pointer conversion

1 An lvalue of function type \( T \) can be converted to a prvalue of type “pointer to \( T \)” The result is a pointer to the function.\(^{56} \)

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

4.4 Qualification conversions

1 A prvalue of type “pointer to \( \text{cv1} T \)” can be converted to a prvalue of type “pointer to \( \text{cv2} T \)” if “\( \text{cv2} T \)” is more cv-qualified than “\( \text{cv1} T \)”.

2 A prvalue of type “pointer to member of \( X \) of type \( \text{cv1} T \)” can be converted to a prvalue of type “pointer to member of \( X \) of type \( \text{cv2} T \)” if “\( \text{cv2} T \)” is more cv-qualified than “\( \text{cv1} T \)”.

[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:\(^{57} \)

Two pointer types \( T1 \) and \( T2 \) are similar if there exists a type \( T \) and integer \( n > 0 \) such that:

\[ T1 \text{ is } \text{cv1}_{1,0} \text{ pointer to } \text{cv1}_{1,1} \text{ pointer to } \cdots \text{cv1}_{1,n-1} \text{ pointer to } \text{cv1}_{1,n} T \]

and

\[ T2 \text{ is } \text{cv2}_{2,0} \text{ pointer to } \text{cv2}_{2,1} \text{ pointer to } \cdots \text{cv2}_{2,n-1} \text{ pointer to } \text{cv2}_{2,n} T \]

\(^{54} \) For historical reasons, this conversion is called the “lvalue-to-rvalue” conversion, even though that name does not accurately reflect the taxonomy of expressions described in 3.10.

\(^{55} \) In C++ class prvalues can have cv-qualified types (because they are objects). This differs from ISO C, in which non-lvalues never have cv-qualified types.

\(^{56} \) This conversion never applies to non-static member functions because an lvalue that refers to a non-static member function cannot be obtained.

\(^{57} \) These rules ensure that const-safety is preserved by the conversion.
where each \( cv_{i,j} \) is \texttt{const}, \texttt{volatile}, \texttt{const volatile}, or nothing. The n-tuple of cv-qualifiers after the first in a pointer type, e.g., \( cv_{1,1}, cv_{1,2}, \ldots, cv_{1,n} \) in the pointer type \( T_1 \), is called the \textit{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 \texttt{const} is in \( cv_{1,j} \) then \texttt{const} is in \( cv_{2,j} \), and similarly for \texttt{volatile}.
- if the \( cv_{1,j} \) and \( cv_{2,j} \) are different, then \texttt{const} is in every \( cv_{2,k} \) for \( 0 < k < j \).

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

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 \texttt{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:

\[
\text{cv}_i P_0 \text{ to } \text{cv}_1 P_1 \text{ to } \cdots \text{cv}_{n-1} P_{n-1} \text{ to } \text{cv}_n T
\]

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:

\[
\begin{align*}
T_1 \text{ is } & \text{cv}_{1,0} P_0 \text{ to } \text{cv}_{1,1} P_1 \text{ to } \cdots \text{cv}_{1,n-1} P_{n-1} \text{ to } \text{cv}_{1,n} T \\
T_2 \text{ is } & \text{cv}_{2,0} P_0 \text{ to } \text{cv}_{2,1} P_1 \text{ to } \cdots \text{cv}_{2,n-1} P_{n-1} \text{ to } \text{cv}_{2,n} T
\end{align*}
\]

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.

### 4.5 Integral promotions \([\text{conv.prom}]\)

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

2 A prvalue of type \texttt{char16_t}, \texttt{char32_t}, or \texttt{wchar_t} (3.9.1) can be converted to a prvalue of the first of the following types that can represent all the values of its underlying type: \texttt{int}, \texttt{unsigned int}, \texttt{long int}, \texttt{unsigned long int}, \texttt{long long int}, or \texttt{unsigned long long int}. If none of the types in that list can represent all the values of its underlying type, a prvalue of type \texttt{char16_t}, \texttt{char32_t}, or \texttt{wchar_t} can be converted to a prvalue of its underlying type.
3 A prvalue of an unscoped enumeration type whose underlying type is not fixed (7.2) can be converted to a prvalue of the first of the following types that can represent all the values of the enumeration (i.e., the values in the range $b_{\text{min}}$ to $b_{\text{max}}$ as described in 7.2): int, unsigned int, long int, unsigned long int, long long int, or unsigned long long int. If none of the types in that list can represent all the values of the enumeration, a prvalue of an unscoped enumeration type can be converted to a prvalue of the extended integer type with lowest integer conversion rank (4.13) greater than the rank of long long in which all the values of the enumeration can be represented. If there are two such extended types, the signed one is chosen.

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

5 A prvalue for an integral bit-field (9.6) can be converted to a prvalue of type int if int can represent all the values of the bit-field; otherwise, it can be converted to unsigned int if unsigned int can represent all the values of the bit-field. If the bit-field is larger yet, no integral promotion applies to it. If the bit-field has an enumerated type, it is treated as any other value of that type for promotion purposes.

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

7 These conversions are called integral promotions.

4.6 Floating point promotion  

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

2 This conversion is called floating point promotion.

4.7 Integral conversions

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

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

3 If the destination type is signed, the value is unchanged if it can be represented in the destination type (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 A prvalue of floating point type can be converted to a prvalue of another floating point type. If the source value can be exactly represented in the destination type, the result of the conversion is that exact representation. If the source value is between two adjacent destination values, the result of the conversion is an implementation-defined choice of either of those values. Otherwise, the behavior is undefined.
The conversions allowed as floating point promotions are excluded from the set of floating point conversions.

4.9 Floating-integral conversions

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

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

4.10 Pointer conversions

1 A null pointer constant is an integral constant expression (5.19) prvalue of integer type that evaluates to zero or a prvalue of type \text{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). A null pointer constant of integral type can be converted to a prvalue of type \text{std::nullptr_t}. \([\text{Note: The resulting prvalue is not a null pointer value. — end note}]\)

2 A prvalue of type “pointer to cv \(T\),” where \(T\) is an object type, can be converted to a prvalue of type “pointer to cv void.” The result of converting a “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 A prvalue of type “pointer to cv \(D\),” where \(D\) is a class type, can be converted to a prvalue of type “pointer to cv \(B\),” where \(B\) is a base class (Clause 10) of \(D\). If \(B\) is an inaccessible (Clause 11) or ambiguous (10.2) base class of \(D\), a program that necessitates this conversion is ill-formed. The result of the conversion is a pointer to the base class subobject of the derived class object. The null pointer value is converted to the null pointer value of the destination type.

4.11 Pointer to member conversions

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

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

§ 4.11
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.\textsuperscript{58}

4.12 Boolean conversions

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

4.13 Integer conversion rank

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

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

— 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 ]

\textsuperscript{58} The rule for conversion of pointers to members (from pointer to member of base to pointer to member of derived) appears inverted compared to the rule for pointers to objects (from pointer to derived to pointer to base) (4.10, Clause 10). This inversion is necessary to ensure type safety. Note that a pointer to member is not a pointer to object or a pointer to function and the rules for conversions of such pointers do not apply to pointers to members. In particular, a pointer to member cannot be converted to a void*.

§ 4.13
5 Expressions

1 [Note: Clause 5 defines the syntax, order of evaluation, and meaning of expressions. An expression is a sequence of operators and operands that specifies a computation. An expression can result in a value and can cause side effects. — end note]

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

3 Clause 5 defines the effects of operators when applied to types for which they have not been overloaded. Operator overloading shall not modify the rules for the built-in operators, that is, for operators applied to types for which they are defined by this Standard. However, these built-in operators participate in overload resolution, and as part of that process user-defined conversions will be considered where necessary to convert the operands to types appropriate for the built-in operator. If a built-in operator is selected, such conversions will be applied to the operands before the operation is considered further according to the rules in Clause 5; see 13.3.1.2, 13.6.

4 If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined. [Note: most existing implementations of C++ ignore integer overflows. Treatment of division by zero, forming a remainder using a zero divisor, and all floating point exceptions vary among machines, and is usually adjustable by a library function. — end note]

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

6 [Note: An expression is an xvalue if it is:
— the result of calling a function, whether implicitly or explicitly, whose return type is an rvalue reference to object type,
— a cast to an rvalue reference to object type,
— a class member access expression designating a non-static data member in which the object expression is an xvalue, or
— a . * pointer-to-member expression in which the first operand is an xvalue and the second operand is a pointer to data member.

In general, the effect of this rule is that named rvalue references are treated as lvalues and unnamed rvalue references to objects are treated as xvalues; rvalue references to functions are treated as lvalues whether named or not. — end note]

Example:

```c
struct A {
    int m;
}
```

59) The precedence of operators is not directly specified, but it can be derived from the syntax.
The expressions \( f() \), \( f().m \), \( \text{static\_cast\<A\&\>(a)} \), and \( a + a \) are xvalues. The expression \( \text{ar} \) is an lvalue.

--- end example

An expression designating an object is called an **object-expression**.

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

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

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

1. If either operand is of scoped enumeration type (7.2), no conversions are performed; if the other operand does not have the same type, the expression is ill-formed.
2. If either operand is of type \texttt{long double}, the other shall be converted to \texttt{long double}.
3. Otherwise, if either operand is \texttt{double}, the other shall be converted to \texttt{double}.
4. Otherwise, if either operand is \texttt{float}, the other shall be converted to \texttt{float}.
5. Otherwise, the integral promotions (4.5) shall be performed on both operands.\(^{60}\) Then the following rules shall be applied to the promoted operands:
   1. If both operands have the same type, no further conversion is needed.
   2. Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank shall be converted to the type of the operand with greater rank.
   3. Otherwise, if the operand that has unsigned integer type has rank greater than or equal to the rank of the type of the other operand, the operand with signed integer type shall be converted to the type of the operand with unsigned integer type.
   4. Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type shall be converted to the type of the operand with signed integer type.
   5. Otherwise, both operands shall be converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

\(^{60}\) 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.
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.\footnote{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 [expr.prim]

5.1.1 General [expr.prim.general]

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

id-expression:
  unqualified-id
  qualified-id

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

A literal is a primary expression. Its type depends on its form (2.14). A string literal is an lvalue; all other literals are prvalues.

The keyword this names a pointer to the object for which a non-static member function (9.3.2) is invoked or a non-static data member’s initializer (9.2) is evaluated. The keyword this shall be used only inside the body of a non-static member function (9.3) of the nearest enclosing class or in a brace-or-equal-initializer for a non-static data member (9.2). The type of the expression is a pointer to the class of the function or non-static data member, possibly with cv-qualifiers on the class type. The expression is a prvalue.

[Example:

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

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

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

The operator :: followed by an identifier, a qualified-id, an operator-function-id, or a literal-operator-id is a primary-expression. Its type is specified by the declaration of the identifier, qualified-id, operator-function-id, or literal-operator-id. The result is the entity denoted by the identifier, qualified-id, operator-function-id,
or literal-operator-id. The result is an lvalue if the entity is a function or variable and a prvalue otherwise. The identifier, qualified-id, operator-function-id, or literal-operator-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 an entity declared in the global namespace to be referred to even if its name 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 -> 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 literal-operator-ids, see 13.5.8; for template-ids, see 14.2. A class-name or decltype-specifier prefixed by ~ denotes a destructor; see 12.4. Within the definition of a non-static member function, an identifier that names a non-static member is transformed to a class member access expression (9.3.1). — end note] The type of the expression is the type of the identifier. The result is the entity denoted by the identifier. The result is an lvalue if the entity is a function, variable, or data member and a prvalue otherwise.

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

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

A nested-name-specifier that denotes a class, optionally followed by the keyword template (14.2), and then followed by the name of a member of either that class (9.2) or one of its base classes (Clause 10), is a qualified-id; 3.4.3.1 describes name lookup for class members that appear in qualified-ids. The result is the member. The type of the result is the type of the member. The result is an lvalue if the member is a static member function or a data member and a prvalue otherwise. [Note: a class member can be referred to using a qualified-id at any point in its potential scope (3.3.7). — end note] Where class-name :: class-name is used, 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). The form ~ decltype-specifier also denotes the destructor, but it shall not be used as the unqualified-id in a qualified-id. [Note: a typedef-name that names a class is a class-name (9.1). — end note]

7 A nested-name-specifier that names a namespace (7.3), followed by the name of a member of that namespace (or the name of a member of a namespace made visible by a using-directive) is a qualified-id; 3.4.3.2 describes name lookup for namespace members that appear in qualified-ids. The result is the member. The type of the result is the type of the member. The result is an lvalue if the member is a function or a variable and a prvalue otherwise.

8 A nested-name-specifier that denotes an enumeration (7.2), followed by the name of an enumerator of that enumeration, is a qualified-id that refers to the enumerator. The result is the enumerator. The type of the result is the type of the enumeration. The result is a prvalue.
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
— in a brace-or-equal-initializer for a non-static data member of that class or of a class derived from that class (12.6.2), or
— if that id-expression denotes a non-static data member and it appears in an unevaluated operand.

[Example:

```cpp
struct S {
    int m;
};
int i = sizeof(S::m); // OK
int j = sizeof(S::m + 42); // OK
```
— end example]

5.1.2 Lambda expressions

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

```cpp
#include <algorithm>
#include <cmath>
void abssort(float *x, unsigned N) {
    std::sort(x, x + N,
        [](float a, float b) {
            return std::abs(a) < std::abs(b);
        });
}
```
— end example]

lambda-expression:
    lambda-introducer lambda-declarator_opt compound-statement

lambda-introducer:
    [ lambda-capture_opt ]

lambda-capture:
    capture-default
capture-list
capture-default, capture-list

capture-default:
    &

§ 5.1.2
The evaluation of a lambda-expression results in a prvalue temporary (12.2). This temporary is called the closure object. A lambda-expression shall not appear in an unevaluated operand (Clause 5). [Note: a closure object behaves like a function object (20.8). — end note]

The type of the lambda-expression (which is also the type of the closure object) is a unique, unnamed non-union class type — called the closure type — whose properties are described below. This class type is not an aggregate (8.5.1). The closure type is declared in the smallest block scope, class scope, or namespace scope that contains the corresponding lambda-expression. [Note: this determines the set of namespaces and classes associated with the closure type (3.4.2). The parameter types of a lambda-declarator do not affect these associated namespaces and classes. — end note] An implementation may define the closure type differently from what is described below provided this does not alter the observable behavior of the program other than by changing:

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

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

If a lambda-expression does not include a lambda-declarator, it is as if the lambda-declarator were (). If a lambda-expression does not include a trailing-return-type, it is as if the trailing-return-type denotes the following type:

— if the compound-statement if of the form

```c
{ return attribute-specifier_opt expression ; }
```

the type of the returned expression after lvalue-to-rvalue conversion (4.1), array-to-pointer conversion (4.2), and function-to-pointer conversion (4.3);
— otherwise, void.

[Example:

```c
auto x1 = [](int i){ return i; }; // OK: return type is int
auto x2 = []{ return { 1, 2 }; }; // error: the return type is void (a braced-init-list is not an expression)
```

— end example]

The closure type for a lambda-expression has a public inline function call operator (13.5.4) whose parameters and return type are described by the lambda-expression’s parameter-declaration-clause and trailing-return-type respectively. This function call operator is declared const (9.3.1) if and only if the lambda-expression’s parameter-declaration-clause is not followed by mutable. It is neither virtual nor declared

§ 5.1.2
volatile. Default arguments (8.3.6) shall not be specified in the parameter-declaration-clause of a lambda-declarator. Any exception-specification specified on a lambda-expression applies to the corresponding function call operator. Any attribute-specifiers appearing immediately after the lambda-expression’s parameter-declaration-clause appertain to the type of the corresponding function call operator. [Note: names referenced in the lambda-declarator are looked up in the context in which the lambda-expression appears. — end note]

6 The closure type for a lambda-expression with no lambda-capture has a public non-virtual non-explicit const conversion function to pointer to function having the same parameter and return types as the closure type’s function call operator. The value returned by this conversion function shall be the address of a function that, when invoked, has the same effect as invoking the closure type’s function call operator.

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

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

— end example]

8 If a lambda-capture includes a capture-default that is &, the identifiers in the lambda-capture shall not be preceded by &. If a lambda-capture includes a capture-default that is =, the lambda-capture shall not contain this and each identifier it contains shall be preceded by &. An identifier or this shall not appear more than once in a lambda-capture. [Example:

```cpp
struct S2 { void f(int i); }
void S2::f(int i) {
    [&, i]( ); // OK
    [&, &i]{}; // error: i preceded by & when & is the default
    [=, this]{}; // error: this when = is the default
    [i, i]{}; // error: i repeated
}

— end example]

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

10 The identifiers in a capture-list are looked up using the usual rules for unqualified name lookup (3.4.1); each such lookup shall find a variable with automatic storage duration declared in the reaching scope of the local lambda expression. An entity (i.e. a variable or this) is said to be explicitly captured if it appears in the lambda-expression’s capture-list.

11 If a lambda-expression has an associated capture-default and its compound-statement uses (3.2) this or a variable with automatic storage duration and the used entity is not explicitly captured, then the used entity
is said to be *implicitly captured*; such entities shall be declared within the reaching scope of the lambda expression. [Note: the implicit capture of an entity by a nested lambda-expression can cause its implicit capture by the containing lambda-expression (see below). Implicit uses of this can result in implicit capture. — end note]

12 An entity is *captured* if it is captured explicitly or implicitly. An entity captured by a lambda-expression is used (3.2) in the scope containing the lambda-expression. If this is captured by a local lambda expression, its nearest enclosing function shall be a non-static member function. If a lambda-expression uses (3.2) this or a variable with automatic storage duration from its reaching scope, that entity shall be captured by the lambda-expression. If a lambda-expression captures an entity and that entity is not defined or captured in the immediately enclosing lambda expression or function, the program is ill-formed. [Example:

```c
void f1(int i) {
    int const N = 20;
    auto m1 = [=]{
        int const M = 30;
        auto m2 = [i]{
            int x[N][M];
            x[0][0] = i;
        };
        int x[N][M];
        x[0][0] = i;
    };
    struct s1 {
        int f;
        int work(int n) {
            int m = n*n;
            int j = 40;
            auto m3 = [this,m] {
                auto m4 = [&j] {
                    int x = n;
                    // OK: N and M are not "used"
                    x += m;
                    // OK: i is explicitly captured by m2
                    // and implicitly captured by m1
                    x += i;
                    // OK: this captured implicitly by m4
                    // and explicitly captured by m3
                    x += f;
                    // OK: this captured implicitly by m4
                    // and explicitly captured by m3
                };
            };
            return x;
        }
    };
}
```

— end example]

13 A lambda-expression appearing in a default argument shall not implicitly or explicitly capture any entity. [Example:

```c
void f2() {
    int i = 1;
    void g1(int i = (i){ return i; })(); // ill-formed
    void g2(int = (i){ return 0; })();    // ill-formed
    void g3(int = (i){ return i; })();    // ill-formed
    void g4(int = (i){ return 0; })();    // OK
    void g5(int = (i){ return sizeof i; })(); // OK
}
```

§ 5.1.2
An entity is captured by copy if it is implicitly captured and the capture-default is = or if it is explicitly captured with a capture that does not include an &. For each entity captured by copy, an unnamed non-static data member is declared in the closure type. The declaration order of these members is unspecified. The type of such a data member is the type of the corresponding captured entity if the entity is not a reference to an object, or the referenced type otherwise. [Note: if the captured entity is a reference to a function, the corresponding data member is also a reference to a function. — end note]

An entity is captured by reference if it is implicitly or explicitly captured but not captured by copy. It is unspecified whether additional unnamed non-static data members are declared in the closure type for entities captured by reference.

If a lambda-expression \texttt{m1} captures an entity and that entity is captured by an immediately enclosing lambda-expression \texttt{m2}, then \texttt{m1}'s capture is transformed as follows:

- if \texttt{m2} captures the entity by copy, \texttt{m1} captures the corresponding non-static data member of \texttt{m2}'s closure type;
- if \texttt{m2} captures the entity by reference, \texttt{m1} captures the same entity captured by \texttt{m2}.

[Example: the nested lambda expressions and invocations below will output 123234.]

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

— end example]

Every \texttt{id-expression} that is a use (3.2) of an entity captured by copy is transformed into an access to the corresponding unnamed data member of the closure type. If \texttt{this} is captured, each use of \texttt{this} is transformed into an access to the corresponding unnamed data member of the closure type, cast (5.4) to the type of \texttt{this}. [Note: the cast ensures that the transformed expression is a prvalue. — end note]

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

```cpp
void f3() {
    float x, &r = x;
    // x and r are not captured (appearance in a decltype operand is not a “use”)
    decltype(x) y1; // y1 has type float
    decltype(x) y2 = y1; // y2 has type float const because this lambda
    // is not mutable and x is an lvalue
    decltype(r) r1 = y1; // r1 has type float const (transformation not considered)
    decltype(r) r2 = y2; // r2 has type float const
}
```

§ 5.1.2
The closure type associated with a \textit{lambda-expression} has a deleted (8.4.3) default constructor and a deleted copy assignment operator. It has an implicitly-declared copy constructor (12.8) and may have an implicitly-declared move constructor (12.8). \[ \text{Note: the copy/move constructor is implicitly defined in the same way as any other implicitly declared copy/move constructor would be implicitly defined.} \] The closure type associated with a \textit{lambda-expression} has an implicitly-declared destructor (12.4).

When the \textit{lambda-expression} is evaluated, the entities that are captured by copy are used to direct-initialize each corresponding non-static data member of the resulting closure object. (For array members, the array elements are direct-initialized in increasing subscript order.) These initializations are performed in the (unspecified) order in which the non-static data members are declared. \[ \text{Note: this ensures that the destructions will occur in the reverse order of the constructions.} \] A \textit{capture} followed by an ellipsis is a pack expansion (14.5.3). \[ \text{Example:} \]

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

Postfix expressions group left-to-right.

```
postfix-expression:
    primary-expression
    postfix-expression [ expression ]
    postfix-expression [ braced-init-list ]
    postfix-expression ( expression-list opt )
    simple-type-specifier ( expression-list opt )
    typename-specifier ( expression-list opt )
    simple-type-specifier braced-init-list
    typename-specifier braced-init-list
    postfix-expression . template opt id-expression
    postfix-expression -> template opt id-expression
    postfix-expression . pseudo-destructor-name
    postfix-expression -> pseudo-destructor-name
    postfix-expression ++
    postfix-expression --
    dynamic_cast < type-id > ( expression )
    static_cast < type-id > ( expression )
    reinterpret_cast < type-id > ( expression )
    const_cast < type-id > ( expression )
    typeid ( expression )
    typeid ( type-id )
```

expression-list:
    initializer-list

§ 5.2
2 [Note: The > token following the type-id in a dynamic_cast, static_cast, reinterpret_cast, or const_cast may be the product of replacing a >> token by two consecutive > tokens (14.2). — end note]

### 5.2.1 Subscripting

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 unscoped enumeration or integral type. The result is an lvalue of type “T.” The type “T” shall be a completely-defined object type.\(^{62}\) The expression \(E1[E2]\) is identical (by definition) to \(*((E1)+(E2))\) \(^{62}\) [Note: see 5.3 and 5.7 for details of * and + and 8.3.4 for details of arrays. — end note]

A braced-init-list may appear as a subscript for a user-defined \(\text{operator[]}\). In that case, the initializer list is treated as the initializer for the subscript argument of the \(\text{operator[]}\). An initializer list shall not be used with the built-in subscript operator.

[Example:

```cpp
struct X {
    Z operator[](std::initializer_list<int>);
};
X x;
x[{1,2,3}] = 7; // OK: meaning x.operator[](1,2,3)
int a[10];
a[{1,2,3}] = 7; // error: built-in subscript operator
```

— end example]

### 5.2.2 Function call

There are two kinds of function call: ordinary function call and member function\(^{63}\) (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 call is as a member of the object pointed to or referred to by the object expression (5.2.5, 5.5).

In the case of an implicit class member access, the implied object is the one pointed by \this.\(^{63}\) [Note: a member function call of the form \(f()\) is interpreted as \((*\text{this}).\text{f()}\) (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. If the selected function is non-virtual, or if the id-expression in the class member access expression is a qualified-id, that function is called. Otherwise, its final overrider (10.3) in the dynamic type of the object expression is called. \(^{63}\) [Note: the dynamic type is the type of the object pointed or referred to by the current value of the object expression. 12.7 describes

\(^{62}\) This is true even if the subscript operator is used in the following common idiom: \&x[0].

\(^{63}\) A static member function (9.4) is an ordinary function.
the behavior of virtual function calls when the object-expression refers to an object under construction or destruction. — end note]

2 [Note: if a function or member function name is used, and name lookup (3.4) does not find a declaration of that name, the program is ill-formed. No function is implicitly declared by such a call. — end note]

3 The type of the function call expression is the return type of the statically chosen function (i.e., ignoring the virtual keyword), even if the type of the function actually called is different. This type shall be a complete object type, a reference type or the type void.

4 When a function is called, each parameter (8.3.5) shall be initialized (8.5, 12.8, 12.1) with its corresponding argument. If the function is a non-static member function, the this parameter of the function (9.3.2) shall be initialized with a pointer to the object of the call, converted as if by an explicit type conversion (5.4). [Note: There is no access or ambiguity checking on this conversion; the access checking and disambiguation are done as part of the (possibly implicit) class member access operator. See 10.2, 11.2, and 5.2.5. — end note] When a function is called, the parameters that have object type shall have completely-defined object type. [Note: this still allows a parameter to be a pointer or reference to an incomplete class type. However, it prevents a passed-by-value parameter to have an incomplete class type. — end note] During the initialization of a parameter, an implementation may avoid the construction of extra temporaries by combining the conversions on the associated argument and/or the construction of temporaries with the initialization of the parameter (see 12.2). The lifetime of a parameter ends when the function in which it is defined returns. The initialization and destruction of each parameter occurs within the context of the calling function. [Example: the access of the constructor, conversion functions or destructor is checked at the point of call in the calling function. If a constructor or destructor for a function parameter throws an exception, the search for a handler starts in the scope of the calling function; in particular, if the function called has a function-try-block (Clause 15) with a handler that could handle the exception, this handler is not considered. — end example] The value of a function call is the value returned by the called function except in a virtual function call if the return type of the final overrider is different from the return type of the statically chosen function, the value returned from the final overrider is converted to the return type of the statically chosen function.

5 [Note: a function can change the values of its non-const parameters, but these changes cannot affect the values of the arguments except where a parameter is of a reference type (8.3.2); if the reference is to a const-qualified type, const_cast is required to be used to cast away the constness in order to modify the argument’s value. Where a parameter is of const reference type a temporary object is introduced if needed (7.1.6, 2.14, 2.14.5, 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 (18.10). [Note: This paragraph does not apply to arguments passed to a function parameter pack. Function parameter packs are expanded during template instantiation (14.5.3), thus each such argument has a corresponding parameter when a function template specialization is actually called. — end note] The lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the argument expression. An argument that has (possibly cv-qualified) type std::nullptr_t is converted to type void* (4.10). After these conversions, if the argument does not have arithmetic, enumeration, pointer, pointer to member, or class type, the program is ill-formed. Passing a potentially-evaluated argument of class type (Clause 9) with a non-trivial copy constructor or a non-trivial destructor with no corresponding parameter is conditionally-supported, with implementation-defined semantics. If the argument has integral or enumeration type that is subject to

§ 5.2.2
the integral promotions (4.5), or a floating point type that is subject to the floating point promotion (4.6),
the value of the argument is converted to the promoted type before the call. These promotions are referred
to as the default argument promotions.

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

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

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

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

A simple-type-specifier (7.1.6.2) or typename-specifier (14.6) followed by a parenthesized expression-list
constructs a value of the specified type given the expression list. If the expression list is a single expression, the
type conversion expression is equivalent (in definedness, and if defined in meaning) to the corresponding cast
eexpression (5.4). If the type specified is a class type, the class type shall be complete. If the expression list
specifies more than a single value, the type shall be a class with a suitably declared constructor (8.5, 12.1),
and the expression T(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 a prvalue.

The expression T(), where T is a simple-type-specifier or typename-specifier for a non-array complete ob-
ject type or the (possibly cv-qualified) void type, creates a prvalue 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 prvalue (3.10). — end
note]

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

5.2.4 Pseudo destructor call [expr.pseudo]

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

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

::opt nested-name-specifier_opt type-name ::~ type-name

shall designate the same scalar type.

5.2.5 Class member access [expr.ref]

A postfix expression followed by a dot . or an arrow ->, optionally followed by the keyword template (14.8.1),
and then followed by an id-expression, is a postfix expression. The postfix expression before the dot or arrow

§ 5.2.5
is evaluated, the result of that evaluation, together with the id-expression, determines the result of the entire postfix expression.

2 For the first option (dot) the type of the first expression (the object expression) shall be “class object” (of a complete type). For the second option (arrow) the type of the first expression (the pointer expression) shall be “pointer to class object” (of a complete type). In these cases, the id-expression shall name a member of the class or of one of its base classes. [Note: because the name of a class is inserted in its class scope (Clause 9), the name of a class is also considered a nested member of that class. — end note] [Note: 3.4.5 describes how names are looked up after the . and -> operators. — end note]

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

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

— 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; if E1 is an xvalue, then E1.E2 is an xvalue; otherwise, it is a prvalue. Let the notation vq12 stand for the “union” of vq1 and vq2; that is, if vq1 or vq2 is volatile, then vq12 is volatile. Similarly, let the notation cq12 stand for the “union” of cq1 and cq2; that is, if cq1 or cq2 is const, then cq12 is const. If E2 is declared to be a mutable member, then the type of E1.E2 is “vq12 T”. If E2 is not declared to be a mutable member, then the type of E1.E2 is “cq12 vq12 T”.

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

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

— Otherwise, if E1.E2 refers to a non-static member function and the type of E2 is “function of parameter-type-list cv ref-qualifier.opt returning T”, then E1.E2 is a prvalue. The expression designates a non-static member function. The expression can be used only as the left-hand operand of a member function call (9.3). [Note: any redundant set of parentheses surrounding the expression is ignored (5.1). — end note] The type of E1.E2 is “function of parameter-type-list cv returning T”.

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

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

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

---

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

65) Note that if E1 has the type “pointer to class X,” then (*E1) is an lvalue.
5.2.6 Increment and decrement

[expr.post.incr]

1 The value of a postfix ++ expression is the value of its operand. [Note: the value obtained is a copy of the original value — end note] The operand shall be a modifiable lvalue. The type of the operand shall be an arithmetic type or a pointer to a complete object type. The value of the operand object is modified by adding 1 to it, unless the object is of type bool, in which case it is set to true. [Note: this use is deprecated, see Annex D. — end note] The value computation of the ++ expression is sequenced before the modification of the operand object. With respect to an indeterminately-sequenced function call, the operation of postfix ++ is a single evaluation. [Note: Therefore, a function call shall not intervene between the lvalue-to-rvalue conversion and the side effect associated with any single postfix ++ operator. — end note] The result is a prvalue. The type of the result is the cv-unqualified version of the type of the operand. See also 5.7 and 5.17.

2 The operand of postfix -- is decremented analogously to the postfix ++ operator, except that the operand shall not be of type bool. [Note: For prefix increment and decrement, see 5.3.2. — end note]

5.2.7 Dynamic cast

[expr.dynamic.cast]

1 The result of the expression dynamic_cast<T>(v) is the result of converting the expression v to type T. T shall be a pointer or reference to a complete class type, or “pointer to cv void.” The dynamic_cast operator shall not cast away constness (5.2.11).

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

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

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

5 If T is “pointer to 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. 66 The result is an lvalue if T is an lvalue reference, or an xvalue if T is an rvalue reference. In both the pointer and reference cases, the program is ill-formed if cv2 has greater cv-qualification than cv1 or if B is an inaccessible or ambiguous base class of D. [Example:

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

— end example]

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

7 If T is “pointer to cvvoid,” 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 If C is the class type to which T points or refers, the run-time check logically executes as follows:

66) The most derived object (1.8) pointed or referred to by v can contain other B objects as base classes, but these are ignored.
— If, in the most derived object pointed (referred) to by \( v \), \( v \) points (refers) to a public base class subobject of a C object, and if only one object of type C is derived from the subobject pointed (referred) to by \( v \) the result points (refers) to that C object.

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

— Otherwise, the run-time check fails.

9 The value of a failed cast to pointer type is the null pointer value of the required result type. A failed cast to reference type throws std::bad_cast (18.7.2).

[Example:

```cpp
class A { virtual void f();};
class B { virtual void g();};
class D : public virtual A, private B { ;

void g() {
  D d;
  B* bp = (B*)&d; // cast needed to break protection
  A* ap = &d; // public derivation, no cast needed
  D& dr = dynamic_cast<D&>(bp); // fails
  ap = dynamic_cast<A*>(bp); // fails
  bp = dynamic_cast<B*>(ap); // fails
  ap = dynamic_cast<A*>(&d); // succeeds
  bp = dynamic_cast<B*>(&d); // ill-formed (not a run-time check)
}

class E : public D, public B { ;
class F : public E, public D { ;

void h() {
  F f;
  A* ap = &f; // succeeds: finds unique A
  D* dp = dynamic_cast<D*>(ap); // fails: yields 0
  // f has two D subobjects
  E* ep = (E*)ap; // ill-formed: cast from virtual base
  E* ep1 = dynamic_cast<E*>(ap); // succeeds
}
```

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

### 5.2.8 Type identification

1 The result of a typeid expression is an lvalue of static type const std::type_info (18.7.1) and dynamic type const std::type_info or const name where name is an implementation-defined class publicly derived from std::type_info which preserves the behavior described in 18.7.1. The lifetime of the object referred to by the lvalue extends to the end of the program. Whether or not the destructor is called for the std::type_info object at the end of the program is unspecified.

2 When typeid is applied to a glvalue expression whose type is a polymorphic class type (10.3), the result refers to a std::type_info object representing the type of the most derived object (1.8) (that is, the dynamic type) to which the glvalue refers. If the glvalue expression is obtained by applying the unary * operator to a

---

67 The recommended name for such a class is extended_type_info.

§ 5.2.8
pointer and the pointer is a null pointer value (4.10), the typeid expression throws the std::bad_typeid exception (18.7.3).

3 When typeid is applied to an expression other than a glvalue of a polymorphic class type, the result refers to a std::type_info object representing the static type of the expression. Lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) conversions are not applied to the expression. If the type of the expression is a class type, the class shall be completely-defined. The expression is an unevaluated operand (Clause 5).

4 When typeid is applied to a type-id, the result refers to a std::type_info object representing the type of the type-id. If the type of the type-id is a reference to a possibly cv-qualified type, the result of the typeid expression refers to a std::type_info object representing the cv-unqualified referenced type. If the type of the type-id is a class type or a reference to a class type, the class shall be completely-defined.

5 The top-level cv-qualifiers of the glvalue expression or the type-id that is the operand of typeid are always ignored. [Example:

```c
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> (18.7.1) is not included prior to a use of typeid, the program is ill-formed.

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

5.2.9 Static cast [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 or an rvalue reference to function type, the result is an lvalue; if T is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue. The static_cast operator shall not cast away constness (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.” An xvalue of type “cv1 B” may be cast to type “rvalue reference to cv2 D” with the same constraints as for an lvalue of type “cv1 B.” If the object of type “cv1 B” is actually a subobject of an object of type D, the result refers to the enclosing object of type D. Otherwise, the result of the cast is undefined. [Example:

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

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

68) If p is an expression of pointer type, then *p, (**p), *(p), (**(p)), and so on all meet this requirement.
A glvalue of type "cv1 T1" can be cast to type "rvalue reference to cv2 T2" if "cv2 T2" is reference-compatible with "cv1 T1" (8.5.3). The result refers to the object or the specified base class subobject thereof. If T2 is an inaccessible (Clause 11) or ambiguous (10.2) base class of T1, a program that necessitates such a cast is ill-formed.

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 expression e is used as a glvalue if and only if the initialization uses it as a glvalue.

Otherwise, the static_cast shall perform one of the conversions listed below. No other conversion shall be performed explicitly using a static_cast.

Any expression can be explicitly converted to type cv void. The expression value is discarded. [Note: however, if the value is in a temporary object (12.2), the destructor for that object is not executed until the usual time, and the value of the object is preserved for the purpose of executing the destructor. — end note] 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.

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:

```
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 a scoped enumeration type (7.2) can be explicitly converted to an integral type. The value is unchanged if the original value can be represented by the specified type. Otherwise, the resulting value is unspecified. A value of a scoped enumeration type can also be explicitly converted to a floating-point type; the result is the same as that of converting from the original value to the floating-point type.

A value of integral or enumeration type can be explicitly converted to 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.

A prvalue of type “pointer to cv1 B,” where B is a class type, can be converted to a prvalue of type “pointer to cv2 D,” where D is a class derived (Clause 10) from B, if a valid standard conversion from “pointer to D” to “pointer to B” exists (4.10), cv2 is the same cv-qualification as, or greater cv-qualification than, cv1, and B is neither a virtual base class of D nor a base class of a virtual base class of D. The null pointer value (4.10) is converted to the null pointer value of the destination type. If the prvalue of type “pointer to cv1 B” points to a B that is actually a subobject of an object of type D, the resulting pointer points to the enclosing object of type D. Otherwise, the result of the cast is undefined.
A prvalue of type “pointer to member of D of type cv1 T” can be converted to a prvalue of type “pointer to member of B of type cv2 T,” where B is a base class (Clause 10) of D, if a valid standard conversion from “pointer to member of B of type T” to “pointer to member of D of type T” exists (4.11), and cv2 is the same cv-qualification as, or greater cv-qualification than, cv1. The null member pointer value (4.11) is converted to the null member pointer value of the destination type. If 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]

A prvalue of type “pointer to cv1 void” can be converted to a prvalue of type “pointer to cv2 T,” where T is an object type and cv2 is the same cv-qualification as, or greater cv-qualification than, cv1. The null pointer value is converted to the null pointer value of the destination type. 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.
```
— end example]

### 5.2.10 Reinterpret cast

The result of the expression `reinterpret_cast<T>(v)` is the result of converting the expression v to type T. If T is an lvalue reference type or an rvalue reference to function type, the result is an lvalue; if T is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue and the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the expression v. Conversions that can be performed explicitly using `reinterpret_cast` are listed below. No other conversion can be performed explicitly using `reinterpret_cast`.

1. The `reinterpret_cast` operator shall not cast away constness (5.2.11). An expression of integral, enumeration, pointer, or pointer-to-member type can be explicitly converted to its own type; such a cast yields the value of its operand. [Note: The mapping performed by `reinterpret_cast` might, or might not, produce a representation different from the original value. — end note]

2. A pointer can be explicitly converted to any integral type large enough to hold it. The mapping function is implementation-defined. [Note: it is intended to be unsurprising to those who know the addressing structure of the underlying machine. — end note] A value of type `std::nullptr_t` can be converted to an integral type; the conversion has the same meaning and validity as a conversion of `(void*)0` to the integral type. [Note: a `reinterpret_cast` cannot be used to convert a value of any type to the type `std::nullptr_t`. — end note]

3. A value of integral type or enumeration type can be explicitly converted to a pointer. A pointer converted to an integer of sufficient size (if any such exists on the implementation) and back to the same pointer type will have its original value; mappings between pointers and integers are otherwise implementation-defined. [Note: Except as described in 3.7.4.3, the result of such a conversion will not be a safely-derived pointer value. — end note]

4. 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

---

69) Function types (including those used in pointer to member function types) are never cv-qualified; see 8.3.5.
the definition of the function is undefined. Except that converting a prvalue of type “pointer to \( T_1 \)” to the type “pointer to \( T_2 \)” (where \( T_1 \) and \( T_2 \) are function types) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified. [Note: see also 4.10 for more details of pointer conversions. — end note]

7 A pointer to an object can be explicitly converted to a pointer to a different object type.\(^{70}\) When a prvalue \( v \) of type “pointer to \( T_1 \)” is converted to the type “pointer to \( cvT_2 \)”\(^{71}\), the result is `\( \text{static	extunderscore cast}\langle cvT_2\rangle(\text{static	extunderscore cast}\langle cvvoid\rangle(v)) \)` if both \( T_1 \) and \( T_2 \) are standard-layout types (3.9) and the alignment requirements of \( T_2 \) are no stricter than those of \( T_1 \). Converting a prvalue of type “pointer to \( T_1 \)” to the type “pointer to \( T_2 \)” (where \( T_1 \) and \( T_2 \) are object types and where the alignment requirements of \( T_2 \) are no stricter than those of \( T_1 \)) and back to its original type yields the original pointer value. The result of any other such pointer conversion is unspecified.

8 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 a prvalue of one type to the other type and back, possibly with different cv-qualification, shall yield the original pointer value.

9 The null pointer value (4.10) is converted to the null pointer value of the destination type. [Note: A null pointer constant of type `std::nullptr_t` cannot be converted to a pointer type, and a null pointer constant of integral type is not necessarily converted to a null pointer value. — end note]

10 A prvalue of type “pointer to member of \( X \) of type \( T_1 \)” can be explicitly converted to a prvalue of a different type “pointer to member of \( Y \) of type \( T_2 \)” if \( T_1 \) and \( T_2 \) are both function types or both object types.\(^{71}\) The null member pointer value (4.11) is converted to the null member pointer value of the destination type. The result of this conversion is unspecified, except in the following cases:

- converting a prvalue of type “pointer to member function” to a different pointer to member function type and back to its original type yields the original pointer to member value.

- converting a prvalue of type “pointer to data member of \( X \) of type \( T_1 \)” to the type “pointer to data member of \( Y \) of type \( T_2 \)” (where the alignment requirements of \( T_2 \) are no stricter than those of \( T_1 \)) and back to its original type yields the original pointer to member value.

11 An lvalue expression of type \( T_1 \) can be cast to the type “reference to \( T_2 \)” if an expression of type “pointer to \( T_1 \)” can be explicitly converted to the type “pointer to \( T_2 \)” using a `\texttt{reinterpret	extunderscore cast}`. That is, a reference cast `\texttt{reinterpret	extunderscore cast}\langle T\&\rangle(x)` has the same effect as the conversion `\texttt{*reinterpret	extunderscore cast}\langle T\rangle(\&x)` with the built-in `\&` and `\*` operators (and similarly for `\texttt{reinterpret	extunderscore cast}\langle T&&\rangle(x)`). The result refers to the same object as the source lvalue, but with a different type. The result is an lvalue for an lvalue reference type or an rvalue reference to function type and an xvalue for an rvalue reference to object type. No temporary is created, no copy is made, and constructors (12.1) or conversion functions (12.3) are not called.\(^{72}\)

### § 5.2.11 Const cast

The result of the expression `\texttt{const	extunderscore cast}\langle T\rangle(v)` is of type \( T \). If \( T \) is an lvalue reference type or an rvalue reference to function type, the result is an lvalue; if \( T \) is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue and the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the expression \( v \). Conversions that can be

---

\(^{70}\) The types may have different cv-qualifiers, subject to the overall restriction that a `\texttt{reinterpret	extunderscore cast}` cannot cast away constness.

\(^{71}\) \( T_1 \) and \( T_2 \) may have different cv-qualifiers, subject to the overall restriction that a `\texttt{reinterpret	extunderscore cast}` cannot cast away constness.

\(^{72}\) This is sometimes referred to as a type pun.
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

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

and

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

where `T` is any object type or the `void` type and where `cv_{1,k}` and `cv_{2,k}` may be different cv-qualifications, a prvalue 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 For two object types `T1` and `T2`, if a pointer to `T1` can be explicitly converted to the type “pointer to `T2`” using a `const_cast`, then the following conversions can also be made:

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

The result of a reference `const_cast` refers to the original object.

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

6 A null pointer value (4.10) is converted to the null pointer value of the destination type. The null member pointer value (4.11) is converted to the null member pointer value of the destination type.

7 [Note: Depending on the type of the object, a write operation through the pointer, lvalue or pointer to data member resulting from a `const_cast` that casts away a const-qualifier may produce undefined behavior (7.1.6.1). — end note]

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

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

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

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

casting from `X1` to `X2` casts away constness if, for a non-pointer type `T` there does not exist an implicit conversion (Clause 4) from:

\[ Tcv_{1,(N-K+1)} \star cv_{1,(N-K+2)} \star \cdots cv_{1,N} \star \]

to

\[ 73) \text{ `const_cast` is not limited to conversions that cast away a const-qualifier.} \]
\[ T^{cv_2(M-K+1)} \ast cv_2(M-K+2) \ast \cdots \ast cv_2 M \ast \]

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

10 Casting from a prvalue of type “pointer to data member of \( X \) of type \( T_1 \)” to the type “pointer to data member of \( Y \) of type \( T_2 \)” casts away constness if a cast from a prvalue of type “pointer to \( T_1 \)” to the type “pointer to \( T_2 \)” casts away constness.

11 For multi-level pointer to members and multi-level mixed pointers and pointer to members (4.4), the “member” aspect of a pointer to member level is ignored when determining if a \texttt{const} cv-qualifier has been cast away.

12 \[ \text{Note: some conversions which involve only changes in cv-qualification cannot be done using \texttt{const\_cast}.} \]

For instance, conversions between pointers to functions are not covered because such conversions lead to values whose use causes undefined behavior. For the same reasons, conversions between pointers to member functions, and in particular, the conversion from a pointer to a const member function to a pointer to a non-const member function, are not covered. \[ \text{— end note} \]

### 5.3 Unary expressions

1 Expressions with unary operators group right-to-left.

\[
\begin{aligned}
\text{unary-expression:} & \\
\text{postfix-expression} & \\
++ & \text{cast-expression} \\
-- & \text{cast-expression} \\
\text{unary-operator cast-expression} & \\
\text{sizeof unary-expression} & \\
\text{sizeof ( type-id )} & \\
\text{sizeof ... ( identifier )} & \\
\text{alignof ( type-id )} & \\
\text{noexcept-expression} & \\
\text{new-expression} & \\
\text{delete-expression} & \\
\end{aligned}
\]

\[
\begin{aligned}
\text{unary-operator: one of} & \\
* & k + - ! \sim \\
\end{aligned}
\]

#### 5.3.1 Unary operators

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

\[ \text{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 a prvalue, see 4.1.} \[ \text{— end note} \]

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

3 The result of the unary \& operator is a pointer to its operand. The operand shall be an lvalue or a \textit{qualified-id}. If the operand is a \textit{qualified-id} naming a non-static member \( m \) of some class \( C \) with type \( T \), the result has type “pointer to member of class \( C \) of type \( T \)” and is a prvalue designating \( C::m \); the program is ill formed if \( C \) is an ambiguous base (10.2) of the class designated by the \textit{nested-name-specifier} of the \textit{qualified-id}. Otherwise, if the type of the expression is \( T \), the result has type “pointer to \( T \)” and is a prvalue that is the address of

\ § 5.3.1 104
the designated object (1.7) or a pointer to the designated function. [Note: In particular, the address of an object of type “cv T” is “pointer to cv T”, with the same cv-qualification. — end note] [Example:

```cpp
class A {
  int i;
};
class B : A {
};
... &B::i ...
// has type A::*
```

— end example] [Note: a pointer to member formed from a mutable non-static data member (7.1.1) does not reflect the mutable specifier associated with the non-static data member. — end note]

A pointer to member is only formed when an explicit & is used and its operand is a qualified-id not enclosed in parentheses. [Note: that is, the expression &qual, where the qualified-id is enclosed in parentheses, does not form an expression of type “pointer to member.” Neither does qualified-id, because there is no implicit conversion from a qualified-id for a non-static member function to the type “pointer to member function” as there is from an lvalue of function type to the type “pointer to function” (4.3). Nor is &unqualified-id a pointer to member, even within the scope of the unqualified-id’s class. — end note]

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

The address of an overloaded function (Clause 13) can be taken only in a context that uniquely determines which version of the overloaded function is referred to (see 13.4). [Note: since the context might determine whether the operand is a static or non-static member function, the context can also affect whether the expression has type “pointer to function” or “pointer to member function.” — end note]

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

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

The operand of the logical negation operator ! is contextually converted to bool (Clause 4); its value is true if the converted operand is false and false otherwise. The type of the result is bool.

The operand of ~ shall have integral or unscoped enumeration type; the result is the one’s complement of its operand. Integral promotions are performed. The type of the result is the type of the promoted operand. There is an ambiguity in the unary-expression ~X(), where X is a class-name or decltype-specifier. The ambiguity is resolved in favor of treating ~ as a unary complement rather than treating ~X as referring to a destructor.

5.3.2 Increment and decrement [expr.pre.incr]

1 The operand of prefix ++ is modified by adding 1, or set to true if it is bool (this use is deprecated). The operand shall be a modifiable lvalue. The type of the operand shall be an arithmetic type or a pointer to a completely-defined object type. The result is the updated operand; it is an lvalue, and it is a bit-field if the operand is a bit-field. If x is not of type bool, the expression ++x is equivalent to x+=1 [Note: see the discussions of addition (5.7) and assignment operators (5.17) for information on conversions. — end note]

2 The operand of prefix -- is modified by subtracting 1. The operand shall not be of type bool. The requirements on the operand of prefix -- and the properties of its result are otherwise the same as those of
prefix ++. [Note: For postfix increment and decrement, see 5.2.6. — end note]

5.3.3 sizeof [expr.sizeof]

1 The sizeof operator yields the number of bytes in the object representation of its operand. The operand is either an expression, which is an unevaluated operand (Clause 5), or a parenthesized type-id. The sizeof operator shall not be applied to an expression that has function or incomplete type, to an enumeration type whose underlying type is not fixed before all its enumerators have been declared, to the parenthesized name of such types, or to an lvalue that designates a bit-field. sizeof(char), sizeof(signed char) and sizeof(unsigned char) are 1. The result of sizeof applied to any other fundamental type (3.9.1) is implementation-defined. [Note: in particular, sizeof(bool), sizeof(char16_t), sizeof(char32_t), and sizeof(wchar_t) are implementation-defined. — end note] [Note: See 1.7 for the definition of byte and 3.9 for the definition of object representation. — end note]

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

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

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

5 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]

6 The result of sizeof and sizeof... is a constant of type std::size_t. [Note: std::size_t is defined in the standard header <cstdint> (18.2). — end note]

5.3.4 New [expr.new]

1 The new-expression attempts to create an object of the type-id (8.1) or new-type-id to which it is applied. The type of that object is the allocated type. This type shall be a complete object type, but not an abstract class type or array 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]

74) sizeof(bool) is not required to be 1.
75) The actual size of a base class subobject may be less than the result of applying sizeof to the subobject, due to virtual base classes and less strict padding requirements on base class subobjects.
new-expression:
:: opt new new-placement\_opt new-type-id new-initializer\_opt
:: opt new new-placement\_opt ( type-id ) new-initializer\_opt

new-placement:
( expression-list )

new-type-id:
type-specifier-seq new-declarator\_opt

new-declarator:
ptr-operator new-declarator\_opt
noptr-new-declarator

noptr-new-declarator:
[ expression ] attribute-specifier\_opt
noptr-new-declarator [ constant-expression ] attribute-specifier\_opt

new-initializer:
( expression-list\_opt )
braced-init-list

Entities created by a `new-expression` have dynamic storage duration (3.7.4). [Note: the lifetime of such an entity is not necessarily restricted to the scope in which it is created. — end note] If the entity is a non-array object, the `new-expression` returns a pointer to the object created. If it is an array, the `new-expression` returns a pointer to the initial element of the array.

2 If the `auto` type-specifier appears in the type-specifier-seq of a `new-type-id` or type-id of a `new-expression`, the `new-expression` shall contain a `new-initializer` of the form

( assignment-expression )

The allocated type is deduced from the `new-initializer` as follows: Let \( e \) be the assignment-expression in the `new-initializer` and \( T \) be the `new-type-id` or type-id of the `new-expression`, then the allocated type is the type deduced for the variable \( x \) in the invented declaration (7.1.6.4):

\[ T x(e); \]

[Example:
\[
\text{new auto(1); // allocated type is int}
\text{auto x = new auto('a'); // allocated type is char, x is of type char*}
\]

— end example]

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

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

is ill-formed because the binding is
\[
\text{(new int) (*[10])(); // error}
\]

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

§ 5.3.4
new (int (*[10])());

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

5 When the allocated object is an array (that is, the noptr-new-declarator syntax is used or the new-type-id or type-id denotes an array type), the new-expression yields a pointer to the initial element (if any) of the array. [Note: both new int and new int[10] have type int* and the type of new int[1] [10] is int (**) [10] — end note] The attribute-specifier in a noptr-new-declarator appertains to the associated array type.

6 Every constant-expression in a noptr-new-declarator shall be an integral constant expression (5.19) and evaluate to a strictly positive value. The expression in a noptr-new-declarator shall be of integral type, unscoped enumeration type, or a class type for which a single non-explicit conversion function to integral or unscoped 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 noptr-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]

7 When the value of the expression in a noptr-new-declarator is zero, the allocation function is called to allocate an array with no elements. If the value of that expression is such that the size of the allocated object would exceed the implementation-defined limit, no storage is obtained and the new-expression terminates by throwing an exception of a type that would match a handler (15.3) of type std::bad_array_new_length (18.6.2.2).

8 A new-expression obtains storage for the object by calling an allocation function (3.7.4.1). If the new-expression terminates by throwing an exception, it may release storage by calling a deallocation function (3.7.4.2). If the allocated type is a non-array type, the allocation function’s name is operator new and the deallocation function’s name is operator delete. If the allocated type is an array type, the allocation function’s name is operator new[] and the deallocation function’s name is operator delete[]. [Note: an implementation shall provide default definitions for the global allocation functions (3.7.4, 18.6.1.1, 18.6.1.2). A C++ program can provide alternative definitions of these functions (17.6.3.6) and/or class-specific versions (12.5). — end note]

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

10 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]

11 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

§ 5.3.4 108
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.

12 [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]

13 [Note: unless an allocation function is declared with a non-throwing exception-specification (15.4), it indicates failure to allocate storage by throwing a `std::bad_alloc` exception (Clause 15, 18.6.2.1); it returns a non-null pointer otherwise. If the allocation function is declared with a non-throwing exception-specification, it returns null to indicate failure to allocate storage and a non-null pointer otherwise. — end note] If the allocation function returns null, initialization shall not be done, the deallocation function shall not be called, and the value of the new-expression shall be null.

14 [Note: when the allocation function returns a value other than null, it must be a pointer to a block of storage in which space for the object has been reserved. The block of storage is assumed to be appropriately aligned and of the requested size. The address of the created object will not necessarily be the same as that of the block if the object is an array. — end note] A new-expression that creates an object of type `T` initializes that object as follows:
— If the new-initializer is omitted, the object is default-initialized (8.5); if no initialization is performed, the object has indeterminate value.
— Otherwise, the new-initializer is interpreted according to the initialization rules of 8.5 for direct-initialization.

15 The invocation of the allocation function is indeterminately sequenced with respect to the evaluations of expressions in the new-initializer. Initialization of the allocated object is sequenced before the value computation of the new-expression. It is unspecified whether expressions in the new-initializer are evaluated if the allocation function returns the null pointer or exits using an exception.

16 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 above76 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]

76) This may include evaluating a new-initializer and/or calling a constructor.
If the new-expression begins with a unary :: operator, the deallocation function’s name is looked up in the global scope. Otherwise, if the allocated type is a class type $T$ or an array thereof, the deallocation function’s name is looked up in the scope of $T$. If this lookup fails to find the name, or if the allocated type is not a class type or array thereof, the deallocation function’s name is looked up in the global scope.

A declaration of a placement deallocation function matches the declaration of a placement allocation function if it has the same number of parameters and, after parameter transformations (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 (3.7.4.2) and that function, considered as a placement deallocation function, would have been selected as a match for the allocation function, the program is ill-formed. [Example:

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

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

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

If a new-expression calls a deallocation function, it passes the value returned from the allocation function call as the first argument of type void*. If a placement deallocation function is called, it is passed the same additional arguments as were passed to the placement allocation function, that is, the same arguments as those specified with the new-placement syntax. If the implementation is allowed to make a copy of any argument as part of the call to the allocation function, it is allowed to make a copy (of the same original value) as part of the call to the deallocation function or to reuse the copy made as part of the call to the allocation function. If the copy is elided in one place, it need not be elided in the other.

### 5.3.5 Delete

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

```c
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 to object type, or a class type having a single non-explicit conversion function (12.3.2) to a pointer to object type. The result has type void. If the operand has a class type, the operand is converted to a pointer type by calling the above-mentioned conversion function, and the converted operand is used in place of the original operand for the remainder of this section. In either alternative, the value of the operand of delete may be a null pointer value. If it is not a null pointer value, in the first alternative (delete object), the value of the operand of delete shall be a pointer to a non-array object or a pointer to a subobject (1.8) representing a base class of such an object.

---

77) 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.

78) This implies that an object cannot be deleted using a pointer of type void* because void is not an object type.
(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 to a `const` type can be the operand of a `delete-expression`; it is not necessary to cast away the constness (5.2.11) of the pointer expression before it is used as the operand of the `delete-expression`. — end note]

3 In the first alternative (`delete object`), if the static type of the object to be deleted is different from its dynamic type, the static type shall be a base class of the dynamic type of the object to be deleted and the static type shall have a virtual destructor or the behavior is undefined. In the second alternative (`delete array`) if the dynamic type of the object to be deleted differs from its static type, the behavior is undefined.

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

5 If the object being deleted has incomplete class type at the point of deletion and the complete class has a non-trivial destructor or a deallocation function, the behavior is undefined.

6 If the value of the operand of the `delete-expression` is not a null pointer value, the `delete-expression` will invoke the destructor (if any) for the object or the elements of the array being deleted. In the case of an array, the elements will be destroyed in order of decreasing address (that is, in reverse order of the completion of their constructor; see 12.6.2).

7 If the value of the operand of the `delete-expression` is not a null pointer value, the `delete-expression` will call a deallocation function (3.7.4.2). Otherwise, it is unspecified whether the deallocation function will be called. [Note: The deallocation function is called regardless of whether the destructor for the object or some element of the array throws an exception. — end note]

8 [Note: An implementation provides default definitions of the global deallocation functions `operator delete()` for non-arrays (18.6.1.1) and `operator delete[]()` for arrays (18.6.1.2). A C++ program can provide alternative definitions of these functions (17.6.3.6), and/or class-specific versions (12.5). — end note]

9 When the keyword `delete` in a `delete-expression` is preceded by the unary `::` operator, the global deallocation function is used to deallocate the storage.

10 Access and ambiguity control are done for both the deallocation function and the destructor (12.4, 12.5).

5.3.6 Alignof

1 An `alignof` expression yields the alignment requirement of its operand type. The operand shall be a `type-id` representing a complete object type or an array thereof or a reference to a complete object type.

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

3 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.

5.3.7 noexcept operator

1 The `noexcept` operator determines whether the evaluation of its operand, which is an unevaluated operand (Clause 5), can throw an exception (15.1).

```plaintext
noexcept-expression:
    noexcept ( expression )
```

2 The result of the `noexcept` operator is a constant of type `bool` and is an rvalue.

---

79) 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.
The result of the `noexcept` operator is `false` if in a potentially-evaluated context the `expression` would contain

- a potentially evaluated call\(^{80}\) to a function, member function, function pointer, or member function pointer that does not have a non-throwing `exception-specification` (15.4),
- a potentially evaluated `throw-expression` (15.1),
- a potentially evaluated `dynamic_cast` expression `dynamic_cast<T>(v)`, where T is a reference type, that requires a run-time check (5.2.7), or
- a potentially evaluated `typeid` expression (5.2.8) applied to a glvalue expression whose type is a polymorphic class type (10.3).

Otherwise, the result is `true`.

### 5.4 Explicit type conversion (cast notation)

The result of the expression `(T) cast-expression` is of type `T`. The result is an lvalue if `T` is an lvalue reference type or an rvalue reference to function type and an xvalue if `T` is an rvalue reference to object type; otherwise the result is a prvalue. [Note: if `T` is a non-class type that is `cv-qualified`, the `cv-qualifiers` are ignored when determining the type of the resulting prvalue; 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
```

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, a glvalue of an unambiguous non-virtual base class type, or a pointer to member of an unambiguous non-virtual base class type may be explicitly converted to a pointer, a reference, or a pointer to member of a derived class type, respectively.

---

\(^{80}\) This includes implicit calls such as the call to an allocation function in a `new-expression`. 

§ 5.4
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 a prvalue of type “pointer to incomplete class type”. The destination type of a cast using the cast notation can be “pointer to incomplete class type”. If both the operand and destination types are class types and one or both are incomplete, it is unspecified whether the static_cast or the reinterpret_cast interpretation is used, even if there is an inheritance relationship between the two classes. [Note: For example, if the classes were defined later in the translation unit, a multi-pass compiler would be permitted to interpret a cast between pointers to the classes as if the class types were complete at the point of the cast. — end note]

5.5 Pointer-to-member operators [expr.mptr.oper]

1 The pointer-to-member operators ->* and .* group left-to-right.

```c
pm-expression:
  cast-expression
  pm-expression .* cast-expression
  pm-expression ->* cast-expression
```

2 The binary operator .* binds its second operand, which shall be of type “pointer to member of T” (where T is a completely-defined class type) to its first operand, which shall be of class T or of a class of which T is an unambiguous and accessible base class. The result is an object or a function of the type specified by the second operand.

3 The binary operator ->* binds its second operand, which shall be of type “pointer to member of T” (where T is a completely-defined class type) to its first operand, which shall be of type “pointer to T” or “pointer to a class of which T is an unambiguous and accessible base class.” The result is an object or a function of the type specified by the second operand.

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

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

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

§ 5.5
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 ]

6 If the result of .* or ->* is a function, then that result can be used only as the operand for the function call operator (). [Example:
(ptr_to_obj->*ptr_to_mfct)(10);

calls the member function denoted by ptr_to_mfct for the object pointed to by ptr_to_obj. — end example] In a .* expression whose object expression is an rvalue, the program is ill-formed if the second operand is a pointer to member function with ref-qualifier &. In a ->* expression or in a .* expression whose object expression is an lvalue, the program is ill-formed if the second operand is a pointer to member function with ref-qualifier &&. The result of a .* expression whose second operand is a pointer to a data member is of the same value category (3.10) as its first operand. The result of a .* expression whose second operand is a pointer to a member function is a prvalue. The result of an ->* expression is an lvalue if its second operand is a pointer to data member and a prvalue otherwise. If the second operand is the null pointer to member value (4.11), the behavior is undefined.

5.6 Multiplicative operators  [expr.mul]

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

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

2 The operands of * and / shall have arithmetic or unscoped enumeration type; the operands of % shall have integral or unscoped enumeration type. The usual arithmetic conversions are performed on the operands and determine the type of the result.

3 The binary * operator indicates multiplication.

4 The binary / operator yields the quotient, and the binary % operator yields the remainder from the division of the first expression by the second. If the second operand of / or % is zero the behavior is undefined. For integral operands the / operator yields the algebraic quotient with any fractional part discarded;81 if the quotient a/b is representable in the type of the result, (a/b)*b + a%b is equal to a.

5.7 Additive operators  [expr.add]

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 unscoped enumeration type, or one operand shall be a pointer to a completely-defined object type and the other shall have integral or unscoped enumeration type.

81) This is often called truncation towards zero.
2 For subtraction, one of the following shall hold:
   — both operands have arithmetic or unscoped 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
     unscoped enumeration type.

3 The result of the binary + operator is the sum of the operands. The result of the binary - operator is the
   difference resulting from the subtraction of the second operand from the first.

4 For the purposes of these operators, a pointer to a nonarray object behaves the same as a pointer to the
   first element of an array of length one with the type of the object as its element type.

5 When an expression that has integral type is added to or subtracted from a pointer, the result has the type
   of the pointer operand. If the pointer operand points to an element of an array object, and the array is
   large enough, the result points to an element offset from the original element such that the difference of
   the subscripts of the resulting and original array elements equals the integral expression. In other words, if
   the expression P points to the i-th element of an array object, the expressions (P)+N (equivalently, N+(P))
   and (P)-N (where N has the value n) point to, respectively, the i + n-th and i − n-th elements of the array
   object, provided they exist. Moreover, if the expression P points to the last element of an array object,
   the expression (P)+1 points one past the last element of the array object, and if the expression Q points
   one past the last element of an array object, the expression (Q)-1 points to the last element of the array
   object. If both the pointer operand and the result point to elements of the same array object, or one past
   the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is
   undefined.

6 When two pointers to elements of the same array object are subtracted, the result is the difference of the
   subscripts of the two array elements. The type of the result is an implementation-defined signed integral
   type; this type shall be the same type that is defined as std::ptrdiff_t in the <cstdint> header (18.2). As
   with any other arithmetic overflow, if the result does not fit in the space provided, the behavior is undefined.
   In other words, if the expressions P and Q point to, respectively, the i-th and j-th elements of an array object,
   the expression (P)-(Q) has the value i − j provided the value fits in an object of type std::ptrdiff_t.
   Moreover, if the expression P points either to an element of an array object or one past the last element of
   an array object, and the expression Q points to the last element of the same array object, the expression
   (Q+1)-(P) has the same value as ((Q)-(P))+1 and as −((P)-((Q)+1)), and has the value zero if the
   expression P points one past the last element of the array object, even though the expression (Q)+1 does not
   point to an element of the array object. Unless both pointers point to elements of the same array object, or
   one past the last element of the array object, the behavior is undefined.\footnote{Another way to approach
   pointer arithmetic is first to convert the pointer(s) to character pointer(s): In this scheme the
   integral value of the expression added to or subtracted from the converted pointer is first multiplied by the
   size of the object originally pointed to, and the resulting pointer is converted back to the original type. For
   pointer subtraction, the result of the difference between the character pointers is similarly divided by the
   size of the object originally pointed to. When viewed in this way, an implementation need only provide
   one extra byte (which might overlap another object in the program) just after the end of the object in order to
   satisfy the “one past the last element” requirements.}

7 If the value 0 is added to or subtracted from a pointer value, the result compares equal to the original pointer
   value. If two pointers point to the same object or both point one past the end of the same array or both
   are null, and the two pointers are subtracted, the result compares equal to the value 0 converted to the type
   std::ptrdiff_t.

\footnote{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.}
5.8 Shift operators

1 The shift operators `<<` and `>>` group left-to-right.

```
shift-expression:
  additive-expression
  shift-expression << additive-expression
  shift-expression >> additive-expression
```

The operands shall be of integral or unscoped enumeration type and integral promotions are performed. The type of the result is that of the promoted left operand. The behavior is undefined if the right operand is negative, or greater than or equal to the length in bits of the promoted left operand.

2 The value of `E1 << E2` is `E1` left-shifted `E2` bit positions; vacated bits are zero-filled. If `E1` has an unsigned type, the value of the result is `E1 \times 2^{E2}`, reduced modulo one more than the maximum value representable in the result type. Otherwise, if `E1` has a signed type and non-negative value, and `E1 \times 2^{E2}` is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

3 The value of `E1 >> E2` is `E1` right-shifted `E2` bit positions. If `E1` has an unsigned type or if `E1` has a signed type and a non-negative value, the value of the result is the integral part of the quotient of `E1/2^{E2}`. If `E1` has a signed type and a negative value, the resulting value is implementation-defined.

5.9 Relational operators

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

```
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`.

2 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, or on two null pointer constants, at least one of which is non-integral) to bring them to their `composite pointer type`. If one operand is a null pointer constant, the composite pointer type is `std::nullptr_t` if the other operand is also a null pointer constant or, if the other operand is a pointer, 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:]

```
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<=q and p>=q both yield true and p<q and p>q both yield false.

— If two pointers p and q of the same type point to different objects that are not members of the same object or elements of the same array or to different functions, or if only one of them is null, the results of p<q, p>q, p=q, and p=q are unspecified.

— If two pointers point to non-static data members of the same object, or to subobjects or array elements of such members, recursively, the pointer to the later declared member compares greater provided the two members have the same access control (Clause 11) and provided their class is not a union.

— If two pointers point to non-static data members of the same object with different access control (Clause 11) the result is unspecified.

— If two pointers point to non-static 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.

3 Pointers to void (after pointer conversions) can be compared, with a result defined as follows: If both pointers represent the same address or are both the null pointer value, the result is true if the operator is <= or >= and false otherwise; otherwise the result is unspecified.

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

5 If both operands (after conversions) are of arithmetic or enumeration type, each of the operators shall yield true if the specified relationship is true and false if it is false.

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 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.

\[\text{Example:}\]

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

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

--- end example

3 If two operands of type \texttt{std::nullptr_t} are compared, the result is \texttt{true} if the operator is \texttt{==}, and \texttt{false} otherwise.

4 Each of the operators shall yield \texttt{true} if the specified relationship is true and \texttt{false} if it is false.

5.11 Bitwise AND operator \hfill [expr.bit.and]

```
and-expression:
    equality-expression
    and-expression & equality-expression
```

1 The usual arithmetic conversions are performed; the result is the bitwise AND function of the operands. The operator applies only to integral or unscoped enumeration operands.

5.12 Bitwise exclusive OR operator \hfill [expr.xor]

```
exclusive-or-expression:
    and-expression
    exclusive-or-expression ^ and-expression
```

1 The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral or unscoped enumeration operands.

5.13 Bitwise inclusive OR operator \hfill [expr.or]

```
inclusive-or-expression:
    exclusive-or-expression
    inclusive-or-expression | exclusive-or-expression
```

1 The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral or unscoped enumeration operands.

5.14 Logical AND operator \hfill [expr.log.and]

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

§ 5.14 118
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`.

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

```plaintext
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`.

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

5.16 Conditional operator

```plaintext
conditional-expression:
  logical-or-expression
  logical-or-expression ? expression : assignment-expression
```

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.

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 a prvalue.
- Both the second and the third operands have type `void`; the result is of type `void` and is a prvalue. [Note: this includes the case where both operands are `throw-expressions`. — end note]

Otherwise, if the second and third operand have different types and either has (possibly cv-qualified) class type, or if both are glvalues of the same value category and the same type except for cv-qualification, an attempt is made to convert each of those operands to the type of the other. The process for determining whether an operand expression `E1` of type `T1` can be converted to match an operand expression `E2` of type `T2` is defined as follows:

- 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 an lvalue.
- If `E2` is an xvalue: `E1` can be converted to match `E2` if `E1` can be implicitly converted to the type “rvalue reference to `T2`”, subject to the constraint that the reference must bind directly.
— If \( \text{E2} \) is an rvalue or if neither of the conversions above can be done and at least one of the operands has (possibly cv-qualified) class type:

— if \( \text{E1} \) and \( \text{E2} \) have class type, and the underlying class types are the same or one is a base class of the other: \( \text{E1} \) can be converted to match \( \text{E2} \) if the class of \( \text{T2} \) is the same type as, or a base class of, the class of \( \text{T1} \), and the cv-qualification of \( \text{T2} \) is the same cv-qualification as, or a greater cv-qualification than, the cv-qualification of \( \text{T1} \). If the conversion is applied, \( \text{E1} \) is changed to a prvalue of type \( \text{T2} \) by copy-initializing a temporary of type \( \text{T2} \) from \( \text{E1} \) and using that temporary as the converted operand.

— Otherwise (i.e., if \( \text{E1} \) or \( \text{E2} \) 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): \( \text{E1} \) can be converted to match \( \text{E2} \) if \( \text{E1} \) can be implicitly converted to the type that expression \( \text{E2} \) would have if \( \text{E2} \) were converted to a prvalue (or the type it has, if \( \text{E2} \) is a prvalue).

Using this process, it is determined whether the second operand can be converted to match the third operand, and whether the third operand can be converted to match the second operand. If both can be converted, or one can be converted but the conversion is ambiguous, the program is ill-formed. If neither can be converted, the operands are left unchanged and further checking is performed as described below. If exactly one conversion is possible, that conversion is applied to the chosen operand and the converted operand is used in place of the original operand for the remainder of this section.

4 If the second and third operands are glvalues of the same value category and have the same type, the result is of that type and value category and it is a bit-field if the second or the third operand is a bit-field, or if both are bit-fields.

5 Otherwise, the result is a prvalue. If the second and third operands do not have the same type, and either has (possibly cv-qualified) class type, overload resolution is used to determine the conversions (if any) to be applied to the operands (13.3.1.2, 13.6). If the overload resolution fails, the program is ill-formed. Otherwise, the conversions thus determined are applied, and the converted operands are used in place of the original operands for the remainder of this section.

6 Lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the second and third operands. After those conversions, one of the following shall hold:

— The second and third operands have the same type; the result is of that type. If the operands have class type, the result is a prvalue temporary of the result type, which is copy-initialized from either the second operand or the third operand depending on the value of the first operand.

— 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, or both are null pointer constants, at least one of which is non-integral; pointer conversions (4.10) and qualification conversions (4.4) are performed to bring them to their composite pointer type (5.9). The result is of the composite pointer type.

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

5.17 Assignment and compound assignment operators

The assignment operator (=) and the compound assignment operators all group right-to-left. All require a
modifiable lvalue as their left operand and return an lvalue referring to the left operand. The result in all cases is a bit-field if the left operand is a bit-field. In all cases, the assignment is sequenced after the value computation of the right and left operands, and before the value computation of the assignment expression. With respect to an indeterminately-sequenced function call, the operation of a compound assignment is a single evaluation. [Note: Therefore, a function call shall not intervene between the lvalue-to-rvalue conversion and the side effect associated with any single compound assignment operator. — end note]

**assignment-expression:**

`conditional-expression`

`logical-or-expression assignment-operator initializer-clause`

`throw-expression`

**assignment-operator:** one of

`== /= %= += -= >>= <<= &= ^= |=`

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

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

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

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

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

7 The behavior of an expression of the form `E1 op = E2` is equivalent to `E1 = E1 op E2` except that `E1` is evaluated only once. In `+=` and `-=`, `E1` shall either have arithmetic type or be a pointer to a possibly cv-qualified completely-defined object type. In all other cases, `E1` shall have arithmetic type.

8 If the value being stored in an object is accessed from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have the same type, otherwise the behavior is undefined. [Note: This restriction applies to the relationship between the left and right sides of the assignment operation; it is not a statement about how the target of the assignment may be aliased in general. See 3.10. — end note]

9 A **braced-init-list** may appear on the right-hand side of

— an assignment to a scalar, in which case the initializer list shall have at most a single element. The meaning of `x={v}`, where `T` is the scalar type of the expression `x`, is that of `x=T(v)` except that no narrowing conversion (8.5.4) is allowed. The meaning of `x={}` is `x=T()`.

— an assignment defined by a user-defined assignment operator, in which case the initializer list is passed as the argument to the operator function.

**Example:**

```cpp
complex<double> z;
z = { 1, 2 };  // meaning z.operator=({1,2})
z += { 1, 2 };  // meaning z.operator+=({1,2})
int a, b;
a = b = { 1 };  // meaning a=b=1;
a = { 1 } = b;  // syntax error
```

— end example]
5.18 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 of the same value category as its right operand, and is a bit-field if its right operand is a glvalue and a bit-field.

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]:

— this (5.1) unless it appears as the postfix-expression in a class member access expression, including the result of the implicit transformation in the body of a non-static member function (9.3.1);

— an invocation of a function other than a constexpr function or a constexpr constructor [Note: overload resolution (13.3) is applied as usual — end note];

— an invocation of an undefined constexpr function or an undefined constexpr constructor outside the definition of a constexpr function or a constexpr constructor;

— an invocation of a constexpr function with arguments that, when implicitly converted to the corresponding parameter types and substituted for the corresponding parameters in the potential constant expression of the constexpr function, and the resulting expression implicitly converted to the return type, do not produce a constant expression; [Example:

constexpr const int* addr(const int& ir) { return &ir; } // OK
static const int x = 5;
constexpr const int* xp = addr(x); // OK: (const int*)&(const int&)x is an address constant expression
constexpr const int* tp = addr(5); // error, initializer for constexpr variable not a constant

(83) However, an invocation of an overloaded comma operator is an ordinary function call; hence, the evaluations of its argument expressions are unsequenced relative to one another (see 1.9).
// expression; (const int*)&(const int&)5 is not a constant
// expression because it takes the address of a temporary

— end example ]
— a result that is not mathematically defined or not in the range of representable values for its type;
— a lambda-expression (5.1.2);
— an lvalue-to-rvalue conversion (4.1) unless it is applied to
   — a glvalue of integral or enumeration type that refers to a non-volatile const object with a preceding
     initialization, initialized with a constant expression, or
   — a glvalue of literal type that refers to a non-volatile object defined with constexpr, or that refers
     to a sub-object of such an object, or
   — a glvalue of literal type that refers to a non-volatile temporary object initialized with a constant
     expression,84
   — an array-to-pointer conversion (4.2) that is applied to a glvalue that does not designate an object with
     static storage duration;
— a unary operator & (5.3.1) that is applied to an lvalue that does not designate an object with static
   storage duration;
— an id-expression that refers to a variable or data member of reference type;85
— 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);
— 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 initializers (7.2), and as integral or enumeration non-type template arguments (14.3). — end
note ]

4 [ Note: Although in some contexts constant expressions must be evaluated during program translation, others
may be evaluated during program execution. Since this International Standard imposes no restrictions on the

84) The temporary must be part of the constant expression, as any longer-lived temporary would have to be bound to a
reference, and reference variables cannot appear in a constant expression.
85) Use of a reference parameter of a constexpr function does not prevent the body from being a potential constant expression
because the parameters are replaced by constant expressions during that determination, and later by arguments to a call.
accuracy of floating-point operations, it is unspecified whether the evaluation of a floating-point expression during translation yields the same result as the evaluation of the same expression (or the same operations on the same values) during program execution.\[^{86}\][Example:

```c
bool f() {
    char array[1 + int(1 + 0.2 - 0.1 - 0.1)];  // Must be evaluated during translation
    int size = 1 + int(1 + 0.2 - 0.1 - 0.1);  // May be evaluated at runtime
    return sizeof(array) == size;
}
```

It is unspecified whether the value of f() will be true or false. — end example] — end note]

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:

```c
struct A {
    constexpr A(int i) : val(i) { }
    constexpr operator int() { return val; }
    constexpr operator long() { return 43; }
private:
    int val;
};
template<int> struct X { };
constexpr A a = 42;
X<a> x;  // OK: unique conversion to int
int ary[a];  // error: ambiguous conversion
```

— end example]

An expression is a potential constant expression if it is a constant expression when all occurrences of function parameters are replaced as follows:

— for non-reference parameters, by arbitrary prvalue constant expressions of the appropriate types;
— for lvalue reference parameters, by arbitrary variables of the referred-to types with static storage duration initialized with constant expressions; or
— for rvalue reference parameters, by arbitrary prvalue constant expressions of the referred-to types implicitly converted to the types of the parameters.

---

\[^{86}\) Nonetheless, implementations are encouraged to provide consistent results, irrespective of whether the evaluation was actually performed during translation or during program execution.

§ 5.19
6 Statements

1 Except as indicated, statements are executed in sequence.

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

   The optional attribute-specifier appertains to the respective statement.

6.1 Labeled statement

1 A statement can be labeled.

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

   The optional attribute-specifier appertains to the label. An identifier label declares the identifier. The only use of an identifier label is as the target of a goto. The scope of a label is the function in which it appears. Labels shall not be redeclared within a function. A label can be used in a goto statement before its definition. Labels have their own name space and do not interfere with other identifiers.

2 Case labels and default labels shall occur only in switch statements.

6.2 Expression statement

1 Expression statements have the form

   expression-statement:
   expression_opt ;

   The expression is evaluated and its value is discarded. The lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are not applied to the expression. All side effects from an expression statement are completed before the next statement is executed. An expression statement with the expression missing is called a null statement. [Note: Most statements are expression statements — usually assignments or function calls. A null statement is useful to carry a label just before the } of a compound statement and to supply a null body to an iteration statement such as a while statement (6.5.1). — end note]

6.3 Compound statement or block

1 So that several statements can be used where one is expected, the compound statement (also, and equivalently, called “block”) is provided.

   compound-statement:
   { statement-seq_opt }
6.4 Selection statements

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

Selection statements choose one of several flows of control.

1 Selection statements choose one of several flows of control.

- **if (condition) statement**
- **if (condition) statement else statement**
- **switch (condition) statement**

**condition:**

- **expression**
- **attribute-specifier_opt type-specifier-seq declarator = initializer-clause**
- **attribute-specifier_opt type-specifier-seq declarator braced-init-list**

See 8.3 for the optional attribute-specifier in a condition. In Clause 6, the term substatement refers to the contained statement or statements that appear in the syntax notation. The substatement in a selection-statement (each substatement, in the else form of the if statement) implicitly defines a block scope (3.3). If the substatement in a selection-statement is a single statement and not a compound-statement, it is as if it was rewritten to be a compound-statement containing the original substatement. [Example:

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

can be equivalently rewritten as

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

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

2 The rules for conditions apply both to selection-statements and to the for and while statements (6.5). The declarator shall not specify a function or an array. If the auto type-specifier appears in the type-specifier-seq, the type of the identifier being declared is deduced from the initializer as described in 7.1.6.4.

3 A name introduced by a declaration in a condition (either introduced by the type-specifier-seq or the declarator of the condition) is in scope from its point of declaration until the end of the substatements controlled by the condition. If the name is re-declared in the outermost block of a substatement controlled by the condition, the declaration that re-declares the name is ill-formed. [Example:

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

— end example]

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

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

6.4.1 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.87

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.

6 case and default labels in themselves do not alter the flow of control, which continues unimpeded across such labels. To exit from a switch, see break, 6.6.1. [Note: usually, the substatement that is the subject of a switch is compound and case and default labels appear on the top-level statements contained within

87) In other words, the else is associated with the nearest un-elsed if.
the (compound) substatement, but this is not required. Declarations can appear in the substatement of a
switch-statement. — end note]

6.5 Iteration statements [stmt.iter]

1 Iteration statements specify looping.

\[
\begin{align*}
\text{iteration-statement:} & \\
& \text{while ( condition ) statement} \\
& \text{do statement while ( expression ) ;} \\
& \text{for ( for-init-statement condition\textsubscript{opt} ; expression\textsubscript{opt} ) statement} \\
& \text{for ( for-range-declaration : expression ) statement}
\end{align*}
\]

\[for-init-statement: \]

\[
\begin{align*}
\text{expression-statement} \\
\text{simple-declaration}
\end{align*}
\]

\[for-range-declaration: \]

\[
\begin{align*}
\text{attribute-specifier\textsubscript{opt} type-specifier-seq declarator}
\end{align*}
\]

See 8.3 for the optional attribute-specifier in a for-range-declaration. [Note: a for-init-statement ends with a semicolon. — end note]

2 The substatement in an iteration-statement implicitly defines a block scope (3.3) which is entered and exited each time through the loop.

If the substatement in an iteration-statement is a single statement and not a compound-statement, it is as if it was rewritten to be a compound-statement containing the original statement. [Example:

\[
\begin{align*}
\text{while (--x >= 0)} \\
& \text{int i;}
\end{align*}
\]

can be equivalently rewritten as

\[
\begin{align*}
\text{while (--x >= 0) } \\
& \text{int i;}
\end{align*}
\]

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

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

5 A loop that, outside of the for-init-statement in the case of a for statement,

\[
\begin{align*}
& \text{— makes no calls to library I/O functions, and} \\
& \text{— does not access or modify volatile objects, and} \\
& \text{— performs no synchronization operations (1.10) or atomic operations (Clause 29)}
\end{align*}
\]

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 [stmt.while]

1 In the while statement the substatement is executed repeatedly until the value of the condition (6.4) becomes false. The test takes place before each execution of the substatement.

2 When the condition of a while statement is a declaration, the scope of the variable that is declared extends from its point of declaration (3.3.2) to the end of the while statement. A while statement of the form
while (T t = x) statement

is equivalent to

```
label: {                  // start of condition scope
   T t = x;
   if (t) {
      statement
      goto label;
   }
}                         // end of condition scope
```

The variable created in a condition is destroyed and created with each iteration of the loop. [Example:

```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 [stmt.do]
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 [stmt.for]
1 The for statement

```
for ( for-init-statement condition_{opt} ; expression_{opt} ) statement
```

is equivalent to

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

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

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

If the for-init-statement is a declaration, the scope of the name(s) declared extends to the end of the for-statement. [Example:

```c
int i = 42;
int a[10];

for (int i = 0; i < 10; i++)
    a[i] = i;

int j = i;       // j = 42

— end example]
```

### 6.5.4 The range-based for statement

The range-based for statement

```
for ( for-range-declaration : expression ) statement
```

is equivalent to

```
{
    auto && __range = ( expression );
    for ( auto __begin = begin-expr,
          __end = end-expr;
         __begin != __end;
         ++__begin ) {
        for-range-declaration = *__begin;
        statement
    }
}
```

where __range, __begin, and __end are variables defined for exposition only, and _RangeT is the type of the expression, and begin-expr and end-expr are determined as follows:

— if _RangeT is an array type, begin-expr and end-expr are __range and __range + __bound, respectively, where __bound is the array bound. If _RangeT is an array of unknown size or an array of incomplete type, the program is ill-formed.

— otherwise, begin-expr and end-expr are begin(__range) and end(__range), respectively, where begin and end are looked up with argument-dependent lookup (3.4.2). For the purposes of this name lookup, namespace std is an associated namespace.

[Example:

```c
int array[5] = { 1, 2, 3, 4, 5 };
for (int& x : array)
    x *= 2;

— end example]
```

§ 6.5.4
6.6 Jump statements

Jump statements unconditionally transfer control.

\[
\text{jump-statement:} \\
\quad \text{break ;} \\
\quad \text{continue ;} \\
\quad \text{return expression_{opt} ;} \\
\quad \text{return braced-init-list ;} \\
\quad \text{goto identifier ;}
\]

On exit from a scope (however accomplished), objects with automatic storage duration (3.7.3) that have been constructed in that scope are destroyed in the reverse order of their construction. [Note: For temporaries, see 12.2. — end note] Transfer out of a loop, out of a block, or back past an initialized variable with automatic storage duration involves the destruction of objects with automatic storage duration that are in scope at the point transferred from but not at the point transferred to. (See 6.7 for transfers into blocks). [Note: However, the program can be terminated (by calling \texttt{std::exit()} or \texttt{std::abort()} (18.5), for example) without destroying class objects with automatic storage duration. — end note]

6.6.1 The \textit{break} statement

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

6.6.2 The \textit{continue} statement

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

\[
\begin{align*}
\text{while (foo) } & \quad \text{do } \quad \text{for (;;) } \\
\{ & \quad \{ & \quad \{ \\
\quad \text{contin: ;} & \quad \text{contin: ;} & \quad \text{contin: ;} \\
\} & \quad \} & \quad \}
\end{align*}
\]

a \textit{continue} not contained in an enclosed iteration statement is equivalent to \textit{goto contin}.

6.6.3 The \textit{return} statement

A function returns to its caller by the \textit{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 \texttt{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 value of 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 or move of a temporary object (12.2). [Note: A copy or move operation associated with a return statement may be elided or considered as an rvalue for the purpose of overload resolution in selecting a constructor (12.8). — end note] A return statement with a \textit{braced-init-list} initializes the object or reference to be returned from the function by copy-list-initialization (8.5.4) from the specified initializer list. [Example:

\[
\begin{align*}
\text{std::pair}\langle\text{std::string, int}\rangle & \quad \text{f(const char* p, int x) } \\
& \quad \text{return \{p, x\};}
\end{align*}
\]
Flowing off the end of a function is equivalent to a return with no value; this results in undefined behavior in a value-returning function.

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

6.6.4 The goto statement

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

6.7 Declaration statement

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

   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.

2 Variables with automatic storage duration (3.7.3) are initialized each time their declaration-statement is executed. Variables with automatic storage duration declared in the block are destroyed on exit from the block (6.6).

3 It is possible to transfer into a block, but not in a way that bypasses declarations with initialization. A program that jumps\(^88\) from a point where a variable with automatic storage duration is not in scope to a point where it is in scope is ill-formed unless the variable has scalar type, class type with a trivial default constructor and a trivial destructor, a cv-qualified version of one of these types, or an array of one of the preceding types and is declared without an initializer (8.5). [Example:

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

   — end example]

4 The zero-initialization (8.5) of all block-scope variables with static storage duration (3.7.1) or thread storage duration (3.7.2) is performed before any other initialization takes place. Constant initialization (3.6.2) of a block-scope entity with static storage duration, if applicable, is performed before its block is first entered. An implementation is permitted to perform early initialization of other block-scope variables with static or thread storage duration under the same conditions that an implementation is permitted to statically initialize a variable with static or thread storage duration in namespace scope (3.6.2). Otherwise such a variable is

\(^{88}\) The transfer from the condition of a switch statement to a case label is considered a jump in this respect.
initialized the first time control passes through its declaration; such a variable is considered initialized upon
the completion of its initialization. If the initialization exits by throwing an exception, the initialization
is not complete, so it will be tried again the next time control enters the declaration. If control enters
the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for
completion of the initialization.\footnote{The implementation must not introduce any deadlock around execution of the initializer.} If control re-enters the declaration recursively while the variable is being
initialized, the behavior is undefined. [Example:

```c
int foo(int i) {
    static int s = foo(2*i); // recursive call - undefined
    return i+1;
}
```
— end example]

The destructor for a block-scope object with static or thread storage duration will be executed if and only
if it was constructed. [Note: 3.6.3 describes the order in which block-scope objects with static and thread
storage duration are destroyed. — end note]

### 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 indistin-
guishable 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),

```c
T(a)->m = 7;    // expression-statement
T(a)++;         // expression-statement
T(a,5)<<(c;    // expression-statement

T(*d)(int);     // declaration
T(e)[5];        // declaration
T(f) = { 1, 2 }; // declaration
T(*g)(double(3)); // declaration
```

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

The remaining cases are declarations. [Example:

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

\section{6.8}
3 The disambiguation is purely syntactic; that is, the meaning of the names occurring in such a statement, beyond whether they are type-names or not, is not generally used in or changed by the disambiguation. Class templates are instantiated as necessary to determine if a qualified name is a type-name. Disambiguation precedes parsing, and a statement disambiguated as a declaration may be an ill-formed declaration. If, during parsing, a name in a template parameter is bound differently than it would be bound during a trial parse, the program is ill-formed. No diagnostic is required. [Note: This can occur only when the name is declared earlier in the declaration. — end note] [Example:

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

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

— end example]
7 Declarations [dcl.dcl]

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

```plaintext
declaration-seq:
declaration
    declaration-seq declaration

declaration:
    block-declaration
    function-definition
    template-declaration
    explicit-instantiation
    explicit-specialization
    linkage-specification
    namespace-definition
    empty-declaration
    attribute-declaration

block-declaration:
    simple-declaration
    asm-definition
    namespace-alias-definition
    using-declaration
    using-directive
    static_assert-declaration
    alias-declaration
    opaque-enum-declaration

alias-declaration:
    using identifier = type-id ;

simple-declaration:
    attribute-specifier_opt decl-specifier-seq_opt init-declarator-list_opt ;

static_assert-declaration:
    static_assert ( constant-expression , string-literal ) ;

empty-declaration:
    ;

attribute-declaration:
    attribute-specifier ;
```

[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

```plaintext
attribute-specifier_opt decl-specifier-seq_opt init-declarator-list_opt ;
```

is divided into three parts. Attributes are described in 7.6. decl-specifiers, the principal components of a decl-specifier-seq, are described in 7.1. declarators, the components of an init-declarator-list, are described in Clause 8. The optional attribute-specifier in a simple-declaration appertains to each of the entities declared by the declarators; it shall not appear if the optional init-declarator-list is omitted. [Note: In the declaration
for an entity, attributes appertaining to that entity may appear at the start of the declaration and after the 
declarator-id for that declaration. — end note] [Example:

[[noreturn, nothrow]] void f [[noreturn]] () ; // OK

— end example

Except where otherwise specified, the meaning of an attribute-declaration is implementation-defined.

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 turn, can have declarations nested within them. Unless otherwise stated, utterances 
in Clause 7 about components in, of, or contained by a declaration or subcomponent thereof refer only to 
those components of the declaration that are not nested within scopes nested within the declaration.

3 In a simple-declaration, the optional init-declarator-list can be omitted only when declaring a class (Clause 9) 
or enumeration (7.2), that is, when the decl-specifier-seq contains either a class-specifier, an elaborated-
type-specifier with a class-key (9.1), or an enum-specifier. In these cases and whenever a class-specifier or 
enum-specifier is present in the decl-specifier-seq, the identifiers in these specifiers are among the names being 
declared by the declaration (as class-names, enum-names, or enumerators, depending on the syntax). In such 
cases, and except for the declaration of an unnamed bit-field (9.6), the decl-specifier-seq shall introduce one 
or more names into the program, or shall redefine a name introduced by a previous declaration. [Example:

enum { } ; // ill-formed
typedef class { } ; // ill-formed

— end example]

4 In a static_assert-declaration the constant-expression shall be a constant expression (5.19) that can be 
contextually converted to bool (Clause 4). If the value of the expression when so converted is true, the 
declaration has no effect. Otherwise, the program is ill-formed, and the resulting diagnostic message (1.4) 
shall include the text of the string-literal, except that characters not in the basic source character set (2.3) 
are not required to appear in the diagnostic message. [Example:

static_assert(sizeof(long) >= 8, "64-bit code generation required for this library.");

— end example]

5 An empty-declaration has no effect.

6 Each init-declarator in the init-declarator-list contains exactly one declarator-id, which is the name declared 
by that init-declarator and hence one of the names declared by the declaration. The type-specifiers (7.1.6) 
in the decl-specifier-seq and the recursive declarator structure of the init-declarator describe a type (8.3), 
which is then associated with the name being declared by the init-declarator.

7 If the decl-specifier-seq contains the typedef specifier, the declaration is called a typedef declaration and the 
name of each init-declarator is declared to be a typedef-name, synonymous with its associated type (7.1.3). 
If the decl-specifier-seq contains no typedef specifier, the declaration is called a function declaration if the 
type associated with the name is a function type (8.3.5) and an object declaration otherwise.

8 Syntactic components beyond those found in the general form of declaration are added to a function decla-
ration to make a function-definition. An object declaration, however, is also a definition unless it contains 
the extern specifier and has no initializer (3.1). A definition causes the appropriate amount of storage to 
be reserved and any appropriate initialization (8.5) to be done.
Only in function declarations for constructors, destructors, and type conversions can the _decl-specifier-seq_ be omitted.\(^90\)

### 7.1 Specifiers  \[dcl.spec\]

1. The specifiers that can be used in a declaration are
   
   \[
   \begin{align*}
   &\textit{decl-specifier:} \\
   &\quad \textit{storage-class-specifier} \\
   &\quad \textit{type-specifier} \\
   &\quad \textit{function-specifier} \\
   &\quad \textit{friend} \\
   &\quad \textit{typedef} \\
   &\quad \textit{constexpr} \\
   &\quad \textit{decl-specifier-seq} \\
   &\quad \text{attribute-specifier}_\text{opt} \\
   &\quad \textit{decl-specifier} \ \textit{decl-specifier-seq}
   \end{align*}
   \]

   The optional _attribute-specifier_ in a _decl-specifier-seq_ appertains to the type determined by the _decl-specifier-seq_ (8.3). The _attribute-specifier_ affects the type only for the declaration it appears in, not other declarations involving the same type.

2. If a _type-name_ is encountered while parsing a _decl-specifier-seq_, it is interpreted as part of the _decl-specifier-seq_ if and only if there is no previous _type-specifier_ other than a _cv-qualifier_ in the _decl-specifier-seq_. The sequence shall be self-consistent as described below. _[Example:]
   
   ```
   \begin{align*}
   \text{typedef char* Pc;} & \quad \text{static Pc;} & \quad \text{// error: name missing}
   \end{align*}
   ```
   
   Here, the declaration \texttt{static Pc} is ill-formed because no name was specified for the static variable of type \texttt{Pc}. To get a variable called \texttt{Pc}, a _type-specifier_ (other than \textit{const} or \textit{volatile}) has to be present to indicate that the _typedef-name_ \texttt{Pc} is the name being (re)declared, rather than being part of the _decl-specifier_ sequence. For another example,

   ```
   \begin{align*}
   \text{void f(const Pc);} & \quad \text{// void f(char* const) (not const char*)} \\
   \text{void g(const int Pc);} & \quad \text{// void g(const int)}
   \end{align*}
   ```
   
   _— end example _

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

   ```
   \begin{align*}
   \text{void h(unsigned Pc);} & \quad \text{// void h(unsigned int)} \\
   \text{void k(unsigned int Pc);} & \quad \text{// void k(unsigned int)}
   \end{align*}
   ```

   _— end example _ — end note _

### 7.1.1 Storage class specifiers  \[dcl.stc\]

1. The storage class specifiers are
   
   \[
   \begin{align*}
   &\textit{storage-class-specifier:} \\
   &\quad \textit{register} \\
   &\quad \textit{static} \\
   &\quad \textit{thread_local} \\
   &\quad \textit{extern} \\
   &\quad \textit{mutable}
   \end{align*}
   \]

---

90) The “implicit int” rule of C is no longer supported.
At most one storage-class-specifier shall appear in a given decl-specifier-seq, except that thread_local may appear with static or extern. If thread_local appears in any declaration of a variable it shall be present in all declarations of that entity. If a storage-class-specifier appears in a decl-specifier-seq, there can be no typedef specifier in the same decl-specifier-seq and the init-declarator-list of the declaration shall not be empty (except for an anonymous union declared in a named namespace or in the global namespace, which shall be declared static (9.5)). The storage-class-specifier applies to the name declared by each init-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.

2 The register specifier shall be applied only to names of variables declared in a block (6.3) or to function parameters (8.4). It specifies that the named variable has automatic storage duration (3.7.3). A variable declared without a storage-class-specifier at block scope or declared as a function parameter has automatic storage duration by default.

3 A register specifier is a hint to the implementation that the variable so declared will be heavily used. [Note: the hint can be ignored and in most implementations it will be ignored if the address of the variable is taken. This use is deprecated (see D.4). — end note]

4 The thread_local specifier indicates that the named entity has thread storage duration (3.7.2). It shall be applied only to the names of variables of namespace or block scope and to the names of static data members. When thread_local is applied to a variable of block scope the storage-class-specifier static is implied if it does not appear explicitly.

5 The static specifier can be applied only to names of variables and functions and to anonymous unions (9.5). There can be no static function declarations within a block, nor any static function parameters. A static specifier used in the declaration of a variable declares the variable to have static storage duration (3.7.1), unless accompanied by the thread_local specifier, which declares the variable to have thread storage duration (3.7.2). A static specifier can be used in declarations of class members; 9.4 describes its effect. For the linkage of a name declared with a static specifier, see 3.5.

6 The extern specifier can be applied only to the names of variables and functions. The extern specifier cannot be used in the declaration of class members or function parameters. For the linkage of a name declared with an extern specifier, see 3.5. [Note: The extern keyword can also be used in explicit-instantiations and linkage-specifications, but it is not a storage-class-specifier in such contexts. — end note]

7 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.

8 The linkages implied by successive declarations for a given entity shall agree. That is, within a given scope, each declaration declaring the same variable name or the same overloading of a function name shall imply the same linkage. Each function in a given set of overloaded functions can have a different linkage, however.

[ Example:

```
static char* f(); //f() has internal linkage
char* f(); //f() still has internal linkage
{ /* ... */ }

cchar* g(); //g() has external linkage
static char* g() //error: inconsistent linkage
{ /* ... */ }

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

inline void l();
```

§ 7.1.1 138
void l();  // external linkage

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

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

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

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

int c;          // c has external linkage
static int c;   // error: inconsistent linkage

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

— end example]

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]

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]

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

7.1.2 Function specifiers

Function-specifiers can be used only in function declarations.

function-specifier:
  inline
  virtual
  explicit

§ 7.1.2
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. 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.

4 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] A type defined within the body of an `extern inline` function is the same type in every translation unit.

5 The `virtual` specifier shall be used only in the initial declaration of a non-static class member function; see 10.3.

6 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**

1 Declarations containing the `decl-specifier` `typedef` declare identifiers that can be used later for naming fundamental (3.9.1) or compound (3.9.2) types. The `typedef` specifier shall not be combined in a `decl-specifier-seq` with any other kind of specifier except a `type-specifier`, and it shall not be used in the `decl-specifier-seq` of a `parameter-declaration` (8.3.5) nor in the `decl-specifier-seq` of a `function-definition` (8.4).

```
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;
typedef int metricp;
```

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

2 A `typedef-name` can also be introduced by an `alias-declaration`. The `identifier` following the `using` keyword becomes a `typedef-name`. 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]

3 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

§ 7.1.3
If the typedef declaration defines an unnamed class (or enum), the first `typedef-name` declared by the declaration to be that class type (or enum type) is used to denote the class type (or enum type) for linkage purposes only (3.5).  

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

— 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, the declaration of a function or function template, or the declaration of a static data member of a literal type (3.9). If any declaration of a function or function template has `constexpr` specifier, then all its declarations shall contain the `constexpr` specifier.  

[Note: an explicit specialization can differ from the template declaration with respect to the `constexpr` specifier. — end note] [Note: function parameters cannot be declared `constexpr`. — end note]

Example:
```
constexpr int square(int x);  // OK: declaration
constexpr int bufsz = 1024;  // OK: definition
constexpr struct pixel {
    int x;
    int y;
    constexpr pixel(int);
};
constexpr pixel pixel::pixel(int a)
    : x(square(a)), y(square(a))  // OK: definition
{ }
constexpr pixel small(2);  // error: square not defined, so small(2)
    // not constant (5.19) so constexpr not satisfied
constexpr int square(int x) {
    return x * x;
}  // OK: definition
constexpr pixel large(4);  // OK: square defined
int next(constexpr int x) {
    return x + 1;
}  // error: not for parameters
extern constexpr int memsz;  // error: not a definition
```

— end example]

2 A `constexpr` specifier used in the declaration of a function that is not a constructor declares that function to be a `constexpr function`. Similarly, a `constexpr` specifier used in a constructor declaration declares that constructor to be a `constexpr constructor`. Constexpr functions and constexpr constructors are implicitly inline (7.1.2).

3 The definition of a constexpr function shall satisfy the following constraints:
   — it shall not be virtual (10.3)
   — its return type shall be a literal type or a reference to literal type

§ 7.1.5
— each of its parameter types shall be a literal type or a reference to literal type
— its function-body shall be a compound-statement of the form

\{ return expression ; \}

where expression is a potential constant expression (5.19)
— every implicit conversion used in converting expression to the function return type (8.5) shall be one of those allowed in a constant expression (5.19).

[Example:

```cpp
constexpr int square(int x)
{ return x * x; } // OK
constexpr long long_max()
{ return 2147483647; } // OK
constexpr int abs(int x)
{ return x < 0 ? -x : x; } // OK
constexpr void f(int x)
// error: return type is void
{ /* ... */ }
constexpr int prev(int x)
{ return --x; } // error: use of decrement
constexpr int g(int x, int n) {
    int r = 1;
    while (--n > 0) r *= x;
    return r;
}
```

— end example]

4 The definition of a constexpr constructor shall satisfy the following constraints:
— each of its parameter types shall be a literal type or a reference to literal type
— its function-body shall not be a function-try-block
— 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 invoked by a mem-initializer shall be a constexpr constructor.
— every constructor argument and full-expression in a mem-initializer shall be a potential constant expression
— every implicit conversion used in converting a constructor argument to the corresponding parameter type and converting a full-expression to the corresponding member type shall be one of those allowed in a constant expression.

A trivial copy/move constructor is also a constexpr constructor.

[Example:

```cpp
struct Length {
    explicit constexpr Length(int i = 0) : val(i) { }
private:
    int val;
};
```
If the instantiated template specialization of a `constexpr` function template would fail to satisfy the requirements for a constexpr function or constexpr constructor, the `constexpr` specifier is ignored.

A call to a constexpr function produces the same result as a call to an equivalent non-constexpr function in all respects except that a call to a constexpr function can appear in a constant expression.

A `constexpr` specifier for a non-static member function that is not a constructor declares that member function to be `const` (9.3.1). [Note: the `constexpr` specifier has no other effect on the function type. — end note] The class of which that function is a member shall be a literal type (3.9). [Example:

```cpp
class debug_flag {
public:
  explicit debug_flag(bool);
  constexpr bool is_on(); // error: debug_flag not const
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. If it is initialized by a constructor call, the constructor shall be a constexpr constructor and every argument to the constructor shall be a constant expression. Otherwise, every full-expression that appears in its initializer shall be a constant expression. Each implicit conversion used in converting the initializer expressions and each constructor call used for the initialization 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

```cpp
 type-specifier:
  trailing-type-specifier
  class-specifier
  enum-specifier

 trailing-type-specifier:
  simple-type-specifier
  elaborated-type-specifier
  typename-specifier
  cv-qualifier
```

§ 7.1.6
The optional `attribute-specifier` in a `type-specifier-seq` or a `trailing-type-specifier-seq` appertains to the type denoted by the preceding `type-specifiers` (8.3). The `attribute-specifier` affects the type only for the declaration it appears in, not other declarations involving the same type.

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

- `const` can be combined with any `type-specifier` except itself.
- `volatile` can be combined with any `type-specifier` except itself.
- `signed` or `unsigned` can be combined with `char`, `long`, `short`, or `int`.
- `short` or `long` can be combined with `int`.
- `long` can be combined with `double`.
- `long` can be combined with `long`.

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. A `type-specifier-seq` shall not define a class or enumeration unless it appears in the `type-id` of an `alias-declaration` (7.1.3).

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 and 8.3.5 describe how `cv-qualifiers` affect object and function types. — end note] Redundant `cv-qualifications` are ignored. [Note: for example, these could be introduced by typedefs. — end note]

[Note: Declaring a variable `const` can affect its linkage (7.1.1) and its usability in constant expressions (5.19). As described in 8.5, the definition of an object or subobject of `const-qualified type` must specify an initializer or be subject to default-initialization. — end note]

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

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

```c
const int ci = 3; // cv-qualified (initialized as required)
ci = 4; // ill-formed: attempt to modify const
int i = 2; // not cv-qualified
```

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.
const int* cip;  // pointer to const int
cip = &i;       // OK: cv-qualified access path to unqualified
*cip = 4;       // ill-formed: attempt to modify through ptr to const

int* ip;
ip = const_cast<int*>(cip); // cast needed to convert const int* to int*
*ip = 4;             // defined: *ip points to i, a non-const object

const int* ciq = new const int (3); // initialized as required
int* iq = const_cast<int*>(ciq); // cast required
*iq = 4;               // undefined: modifies a const object

5 For another example

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]

6 If an attempt is made to refer to an object defined with a volatile-qualified type through the use of a glvalue with a non-volatile-qualified type, the program behavior is undefined.

7 [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 [dcl.type.simple]

1 The simple type specifiers are

§ 7.1.6.2
simple-type-specifier:
  ::opt nested-name-specifieropt 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
decltype-specifier

type-name:
class-name
cenum-name
typedef-name
dcltype-specifier:
dcltype ( expression )

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 9 — 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>simple-template-id</td>
<td>the type as defined in 14.2</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>cchar16_t</td>
<td>&quot;cchar16_t&quot;</td>
</tr>
<tr>
<td>cchar32_t</td>
<td>&quot;cchar32_t&quot;</td>
</tr>
<tr>
<td>bool</td>
<td>&quot;bool&quot;</td>
</tr>
<tr>
<td>unsigned 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 int</td>
<td>&quot;unsigned long long int&quot;</td>
</tr>
<tr>
<td>signed long int</td>
<td>&quot;long int&quot;</td>
</tr>
<tr>
<td>signed long</td>
<td>&quot;long int&quot;</td>
</tr>
</tbody>
</table>
Table 9 — simple-type-specifiers and the types they specify (continued)

<table>
<thead>
<tr>
<th>Specifier(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<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</td>
<td>“long long int”</td>
</tr>
<tr>
<td>long int</td>
<td>“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</td>
<td>“short int”</td>
</tr>
<tr>
<td>short int</td>
<td>“short int”</td>
</tr>
<tr>
<td>short</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 unparenthesized 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 the statically chosen function;

— otherwise, if e is an lvalue, decltype(e) is T&, where T is the type of e;

— otherwise, decltype(e) is the type of e.

The operand of the decltype specifier is an unevaluated operand (Clause 5).

[Example:

```c
const int&& foo();
int i;
struct A { double x; }; const A* a = new A();
decltype(foo()) x1 = i; // type is const int&&
decltype(i) x2; // type is int
decltype(a->x) x3; // type is double
decltype((a->x)) x4 = x3; // type is const double&
```

— end example]
7.1.6.3 Elaborated type specifiers

elaborated-type-specifier:
  class-key attribute-specifier opt :: opt nested-name-specifier opt identifier
  class-key opt nested-name-specifier opt template opt simple-template-id
  enum opt nested-name-specifier opt identifier

d  An attribute-specifier shall not appear in an elaborated-type-specifier unless the latter is the sole constituent of a declaration. If an elaborated-type-specifier is the sole constituent of a declaration, the declaration is ill-formed unless it is an explicit specialization (14.7.3), an explicit instantiation (14.7.2) or it has one of the following forms:

  class-key attribute-specifier opt 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;

  In the first case, the attribute-specifier, if any, appertains to the class being declared; the attributes in the attribute-specifier are thereafter considered attributes of the class whenever it is named.

2 3.4.4 describes how name lookup proceeds for the identifier in an elaborated-type-specifier. If the identifier resolves to a class-name or enum-name, the elaborated-type-specifier introduces it into the declaration the same way a simple-type-specifier introduces its type-name. If the identifier resolves to a typedef-name, the elaborated-type-specifier is ill-formed. [Note: this implies that, within a class template with a template type-parameter T, the declaration

friend class T;

is ill-formed. However, the similar declaration friend T; is allowed (11.4). — end note]

3  The class-key or enum keyword present in the elaborated-type-specifier shall agree in kind with the declaration to which the name in the elaborated-type-specifier refers. This rule also applies to the form of elaborated-type-specifier that declares a class-name or friend class since it can be construed as referring to the definition of the class. Thus, in any elaborated-type-specifier, the enum keyword shall be used to refer to an enumeration (7.2), the union class-key shall be used to refer to a union (Clause 9), and either the class or struct class-key shall be used to refer to a class (Clause 9) declared using the class or struct class-key.

[Example:

  enum class E { a, b };
  enum E x = E::a; // OK

  — end example]

7.1.6.4 auto specifier

1  The auto type-specifier signifies that the type of a variable being declared shall be deduced from its initializer or that a function declarator shall include a trailing-return-type.

2  The auto type-specifier may appear with a function declarator with a trailing-return-type (8.3.5) in any context where such a declarator is valid.

3  Otherwise, the type of the variable is deduced from its initializer. The name of the variable being declared shall not appear in the initializer expression. This use of auto is allowed when declaring variables in a block (6.3), in namespace scope (3.3.6), and in a for-init-statement (6.5.3). auto shall appear as one of the decl-specifiers in the decl-specifier-seq and the decl-specifier-seq shall be followed by one or more init-declarators, each of which shall have a non-empty initializer.

[Example:

§ 7.1.6.4]
auto x = 5; // OK: x has type int
const auto *v = &x, u = 6; // OK: v has type const int*, u has type const int
static auto y = 0.0; // OK: y has type double
auto int r; // error: auto is not a storage-class-specifier

— end example

4 The auto type-specifier can also be used in declaring a variable in the condition of a selection statement (6.4) or an iteration statement (6.5), in the type-specifier-seq in the new-type-id or type-id of a new-expression (5.3.4), in a for-range-declaration, and in declaring a static data member with a brace-or-equal-initializer that appears within the member-specification of a class definition (9.4.2).

5 A program that uses auto in a context not explicitly allowed in this section is ill-formed.

6 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 either a new invented type template parameter U or, if the initializer is a braced-init-list (8.5.4), with std::initializer_list<U>. The type deduced for the variable d is then the deduced A 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 the initializer for d is the corresponding argument. If the deduction fails, the declaration is ill-formed. [Example:

    auto x1 = {1, 2}; // decltype(x1) is std::initializer_list<int>
    auto x2 = {1, 2.0}; // error: cannot deduce element type

— end example]

7 If the list of declarators contains more than one declarator, the type of each declared variable is determined as described above. If the type deduced for the template parameter U is not the same in each deduction, the program is ill-formed.

    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

7.2 Enumeration declarations [dcl.enum]

1 An enumeration is a distinct type (3.9.1) with named constants. Its name becomes an enum-name, within its scope.

    enum-name:
        identifier
    enum-specifier:
        enum-head { enumerator-list_opt }
    enum-head:
        enum-key attribute-specifier_opt identifier_opt enum-base_opt
        enum-key attribute-specifier_opt nested-name-specifier identifier
        enum-base_opt

§ 7.2
opaque-enum-declaration:
  enum-key attribute-specifier\_opt identifier enum-base\_opt ;

enum-key:
  enum
  enum class
  enum struct
enum-base:
  : type-specifier-seq
enumerator-list:
  enumerator-definition
  enumerator-list , enumerator-definition
enumerator-definition:
  enumerator
  enumerator = constant-expression
enumerator:
  identifier

The optional attribute-specifier in the enum-head and the opaque-enum-declaration appertains to the enumeration; the attributes in that attribute-specifier are thereafter considered attributes of the enumeration whenever it is named.

2 The enumeration type declared with an enum-key of only enum is an unscoped enumeration, and its enumerators are unscoped enumerators. The enum-keys enum class and enum struct are semantically equivalent; an enumeration type declared with one of these is a scoped enumeration, and its enumerators are scoped enumerators. The optional identifier shall not be omitted in the declaration of a scoped enumeration. The type-specifier-seq of an enum-base shall name an integral type; any cv-qualification is ignored. An opaque-enum-declaration declaring an unscoped enumeration shall not omit the enum-base. The identifiers in an 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 (5.19). 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 ]

3 An opaque-enum-declaration is either a redeclaration of an enumeration in the current scope or a declaration of a new enumeration. [ Note: an enumeration declared by an opaque-enum-declaration has fixed underlying type and is a complete type. The list of enumerators can be provided in a later redeclaration with an enum-specifier. — end note ] A scoped enumeration shall not be later redeclared as unscoped or with a different underlying type. An unscoped enumeration shall not be later redeclared as scoped and each redeclaration shall include an enum-base specifying the same underlying type as in the original declaration.

4 If the enum-key is followed by a nested-name-specifier, the enum-specifier shall refer to an enumeration that was previously declared directly in the class or namespace to which the nested-name-specifier refers (i.e., neither inherited nor introduced by a using-declaration), and the enum-specifier shall appear in a namespace enclosing the previous declaration.

5 Each enumeration defines a type that is different from all other types. Each enumeration also has an underlying type. The underlying type can be explicitly specified using 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

§ 7.2
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. If no such type exists, the program is ill-formed.

For an enumeration whose underlying type is not fixed, the underlying type is an integral type that can represent all the enumerator values defined in the enumeration. If no integral type can represent all the enumerator values, the enumeration is ill-formed. It is implementation-defined which integral type is used as the underlying type except that the underlying type shall not be larger than int unless the value of an enumerator cannot fit in an int or unsigned int. If the enumerator-list is empty, the underlying type is as if the enumeration had a single enumerator with value 0.

For an enumeration whose underlying type is fixed, the values of the enumeration are the values of the underlying type. Otherwise, for an enumeration where \( e_{\text{min}} \) is the smallest enumerator and \( e_{\text{max}} \) is the largest, the values of the enumeration are the values in the range \( b_{\text{min}} \) to \( b_{\text{max}} \), defined as follows: Let \( K \) be 1 for a two’s complement representation and 0 for a one’s complement or sign-magnitude representation. \( b_{\text{max}} \) is the smallest value greater than or equal to \( \max(|e_{\text{min}}| - K, |e_{\text{max}}|) \) and equal to \( 2^M - 1 \), where \( M \) is a non-negative integer. \( b_{\text{min}} \) is zero if \( e_{\text{min}} \) is non-negative and \(-b_{\text{max}} + K\) otherwise. The size of the smallest bit-field large enough to hold all the values of the enumeration type is \( \max(M, 1) \) if \( b_{\text{min}} \) is zero and \( M + 1 \) otherwise. It is possible to define an enumeration that has values not defined by any of its enumerators. If the enumerator-list is empty, the values of the enumeration are as if the enumeration had a single enumerator with value 0.

Two enumeration types are layout-compatible if they have the same underlying type.

The value of an enumerator or an object of an unscoped enumeration type is converted to an integer by integral promotion (4.5). [Example:

```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:
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:

```cpp
enum direction { left='l', right='r' };

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

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

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

— end example

An enumerator declared in class scope can be referred to using the class member access operators (`::`, `. (dot) and -> (arrow)`, see 5.2.5.

Example:

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

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

— end example

### 7.3 Namespaces

A namespace is an optionally-named declarative region. The name of a namespace can be used to access entities declared in that namespace; that is, the members of the namespace. Unlike other declarative regions, the definition of a namespace can be split over several parts of one or more translation units.
The outermost declarative region of a translation unit is a namespace; see 3.3.6.

### 7.3.1 Namespace definition

The grammar for a namespace-definition is

```
namespace-name:
  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
```

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

2. The identifier in an extension-namespace-definition shall have previously been defined in an original-namespace-definition in the same declarative region.

3. Every namespace-definition shall appear in the global scope or in a namespace scope (3.3.6).

4. Because a namespace-definition contains declarations in its namespace-body and a namespace-definition is itself a declaration, it follows that namespace-definitions can be nested. [Example:

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

— end example]

5. The enclosing namespaces of a declaration are those namespaces in which the declaration lexically appears, except for a redeclaration of a namespace member outside its original namespace (e.g., a definition as specified in 7.3.1.2). Such a redeclaration has the same enclosing namespaces as the original declaration. [Example:
namespace Q {
    namespace V {
        void f();  // enclosing namespaces are the global namespace, Q, and Q::V
        class C { void m(); }
    }
    void V::f() { // enclosing namespaces are the global namespace, Q, and Q::V
        extern void h();  // ... so this declares Q::V::h
    }
    void V::C::m() { // enclosing namespaces are the global namespace, Q, and Q::V
    }
}

— end example]
The use of the `static` keyword is deprecated when declaring variables in a namespace scope (see annex D); the `unnamed-namespace` provides a superior alternative.

### 7.3.1.2 Namespace member definitions

Members (including explicit specializations of templates (14.7.3)) of a namespace can be defined within that namespace. [Example:]

```cpp
namespace X {
    void f() { /* ... */ }
}

— end example]
```

Members 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();
    }
}

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

---

94) this implies that the name of the class or function is unqualified.
name in a friend declaration is neither qualified nor a template-id and the declaration is a function or an elaborated-type-specifier, the lookup to determine whether the entity has been previously declared shall not consider any scopes outside the innermost enclosing namespace. [Note: the other forms of friend declarations cannot declare a new member of the innermost enclosing namespace and thus follow the usual lookup rules. — end note] [Example:

```cpp
// Assume f and g have not yet been defined.
void h(int);
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;
}

// definition of A::f
void f() { f(x); }
void f2()<int> { /* ... */ } // definition of A::f2<> 

// A::f, A::g and A::h are visible here and known to be friends
using A::x;

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

— end example]

### 7.3.2 Namespace alias

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

- namespace-alias-definition:
  - identifier
  - namespace-alias-definition:
    - namespace alias identifier = qualified-namespace-specifier ;

- qualified-namespace-specifier:
  - ::opt nested-name-specifieropt namespace-name

1. The identifier in a namespace-alias-definition is a synonym for the name of the namespace denoted by the qualified-namespace-specifier and becomes a namespace-alias. [Note: when looking up a namespace-name in a namespace-alias-definition, only namespace names are considered, see 3.4.6. — end note]

2. In a declarative region, a namespace-alias-definition can be used to redefine a namespace-alias declared in that declarative region to refer only to the namespace to which it already refers. [Example: the following declarations are well-formed:

§ 7.3.2
4 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

1 A using-declaration introduces a name into the declarative region in which the using-declaration appears.

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.

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

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

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

— end example]

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

   class C {
   int g();
   };

   class D2 : public B {
    using B::f;
    // OK: B is a base of D2
    using B::e;
    // OK: e is an enumerator of base B

§ 7.3.3
using B::x;  // OK: x is a union member of base B
using C::g;  // error: C isn’t a base of D2
};

— end example]

4 [ Note: Since destructors do not have names, a using-declaration cannot refer to a destructor for a base class. Since specializations of member templates for conversion functions are not found by name lookup, they are not considered when a using-declaration specifies a conversion function (14.5.2). — end note ] If an assignment operator brought from a base class into a derived class scope has the signature of a copy/move assignment operator for the derived class (12.8), the using-declaration does not by itself suppress the implicit declaration of the derived class assignment operator; the copy/move assignment operator from the base class is hidden or overridden by the implicitly-declared copy/move assignment operator of the derived class, as described below.

5 A using-declaration shall not name a template-id. [ Example:

    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.
    }

    — end example ]

9 Members declared by a using-declaration can be referred to by explicit qualification just like other member names (3.4.3.2). In a using-declaration, a prefix :: refers to the global namespace. [ Example:

    void f();

    namespace A {
      void g();
    }

    namespace X {

§ 7.3.3 159
using ::f;  // global f
using A::g;  // A’s g

void h()
{
    X::f();  // calls ::f
    X::g();  // calls A::g
}

/* end example */

A using-declaration is a declaration and can therefore be used repeatedly where (and only where) multiple declarations are allowed. [Example:

namespace A {
    int i;
}

namespace A1 {
    using A::i;
    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 */

The entity declared by a using-declaration shall be known in the context using it according to its definition at the point of the using-declaration. Definitions added to the namespace after the using-declaration are not considered when a use of the name is made. [Example:

namespace A {
    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),

§ 7.3.3
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]

12 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:

    namespace A {
        int x;
    }

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

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

    — end example]

13 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('l');      // error: ambiguous: B::f(int) or C::f(char)?
    void f(int);  // error: f(int) conflicts with C::f(int) and B::f(int)
}

— end example — end note —

15 When a using-declaration brings names from a base class into a derived class scope, member functions and member function templates in the derived class override and/or hide member functions and member function templates with the same name, parameter-type-list (8.3.5), cv-qualification, and ref-qualifier (if any) in a base class (rather than conflicting). [Note: For using-declarations that name a constructor, see 12.9. — end note] [Example:

struct B {
    virtual void f(int);
    virtual void f(char);
    void g(int);
    void h(int);
};

struct D : B {
    using B::f;
    void f(int);  // OK: D::f(int) overrides B::f(int);

    using B::g;
    void g(char);  // OK

    using B::h;
    void h(int);   // OK: D::h(int) hides B::h(int)
};

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

— end example —

16 For the purpose of overload resolution, the functions which are introduced by a using-declaration into a derived class will be treated as though they were members of the derived class. In particular, the implicit this parameter shall be treated as if it were a pointer to the derived class rather than to the base class.
This has no effect on the type of the function, and in all other respects the function remains a member of the base class.

The access rules for inheriting constructors are specified in 12.9; otherwise all instances of the name mentioned in a *using-declaration* shall be accessible. In particular, if a derived class uses a *using-declaration* to access a member of a base class, the member name shall be accessible. If the name is that of an overloaded member function, then all functions named shall be accessible. The base class members mentioned by a *using-declaration* shall be visible in the scope of at least one of the direct base classes of the class where the *using-declaration* is specified. [Note: because a *using-declaration* designates a base class member (and not a member subobject or a member function of a base class subobject), a *using-declaration* cannot be used to resolve inherited member ambiguities. For example,

```c
struct A { int x(); };  // CoD: 11.3.3.3 163
struct B : A { };       // CoD: 11.3.3.3 163
struct C : A {
    using A::x;     // CoD: 11.3.3.3 163
    int x(int);     // CoD: 11.3.3.3 163
};                     // CoD: 11.3.3.3 163
struct D : B, C {
    using C::x;     // CoD: 11.3.3.3 163
    int x(double);  // CoD: 11.3.3.3 163
};                     // CoD: 11.3.3.3 163
int f(D* d) {
    return d->x();  // ambiguous: B::x or C::x
}                   // CoD: 11.3.3.3 163
```

— end note]

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

```c
class A {
private:
    void f(char);  // CoD: 11.3.3.3 163
public:
    void f(int);  // CoD: 11.3.3.3 163
protected:
    void g();    // CoD: 11.3.3.3 163
};                           // CoD: 11.3.3.3 163

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
};                  // CoD: 11.3.3.3 163
```

— end example]

19 [Note: use of access-declarations (11.3) is deprecated; member *using-declarations* provide a better alternative. — end note]
If a using-declaration uses the keyword typename and specifies a dependent name (14.6.2), the name introduced by the using-declaration is treated as a typedef-name (7.1.3).

### 7.3.4 Using directive

**using-directive:**

```plaintext
attribute-specifieropt using namespace ::opt nested-name-specifieropt namespace-name;
```

1. A using-directive shall not appear in class scope, but may appear in namespace scope or in block scope. [Note: when looking up a namespace-name in a using-directive, only namespace names are considered, see 3.4.6. — end note] The optional attribute-specifier appertains to the using-directive.

2. A using-directive specifies that the names in the nominated namespace can be used in the scope in which the using-directive appears after the using-directive. During unqualified name lookup (3.4.1), the names appear as if they were declared in the nearest enclosing namespace which contains both the using-directive and the nominated namespace. [Note: in this context, “contains” means “contains directly or indirectly”. — end note]

3. A using-directive does not add any members to the declarative region in which it appears. [Example:

```plaintext
namespace A {
    int i;
    namespace B {
        namespace C {
            int i;
        }
        using namespace A::B::C;
        void f1() {
            i = 5;       // OK, C::i visible in B and hides A::i
        }
    }
}
namespace D {
    using namespace B;
    using namespace C;
    void f2() {
        i = 5;       // ambiguous, B::C::i or A::i?
    }
    void f3() {
        i = 5;       // uses A::i
    }
    void f4() {
        i = 5;       // ill-formed; neither i is visible
    }
}

— end example]
```

4. For unqualified lookup (3.4.1), the using-directive is transitive: if a scope contains a using-directive that nominates a second namespace that itself contains using-directives, the effect is as if the using-directives from the second namespace also appeared in the first. [Note: For qualified lookup, see 3.4.3.2. — end note] [Example:

```plaintext
namespace M {
    int i;
}
```

§ 7.3.4
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

5 If a namespace is extended by an extension-namespace-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-namespace-definition can be used after the extension-namespace-definition.

6 If name lookup finds a declaration for a name in two different namespaces, and the declarations do not declare the same entity and do not declare functions, the use of the name is ill-formed. [Note: in particular, the name of a variable, function or enumerator does not hide the name of a class or enumeration declared in a different namespace. For example,

namespace A {
    class X { }
    extern "C" int g();
    extern "C++" int h();
}
namespace B {
    void X(int);
    extern "C" int g();
    extern "C++" int h(int);
}
using namespace A;
using namespace B;

§ 7.3.4
void f() {
    X(1); // error: name X found in two namespaces
    g(); // okay: name g refers to the same entity
    h(); // okay: overload resolution selects A::h
}

— end note]

7 During overload resolution, all functions from the transitive search are considered for argument matching. The set of declarations found by the transitive search is unordered. [Note: in particular, the order in which namespaces were considered and the relationships among the namespaces implied by the using-directives do not cause preference to be given to any of the declarations found by the search. — end note] An ambiguity exists if the best match finds two functions with the same signature, even if one is in a namespace reachable through using-directives in the namespace of the other. 95 [Example:

    namespace D {
        int d1;
        void f(char);
    }
    using namespace D;

    int d1; // OK: no conflict with D::d1

    namespace E {
        int e;
        void f(int);
    }

    namespace D { // namespace extension
        int d2;
        using namespace E;
        void f(int);
    }

    void f() {
        d1++; // error: ambiguous ::d1 or D::d1?
        ::d1++; // OK
        D::d1++; // OK
        d2++; // OK: D::d2
        e++; // OK: E::e
        f(1); // error: ambiguous: D::f(int) or E::f(int)?
        f('a'); // OK: D::f(char)
    }

— end example]

7.4 The asm declaration

1 An asm declaration has the form

    asm-definition:
    asm ( string-literal ) ;

95) During name lookup in a class hierarchy, some ambiguities may be resolved by considering whether one member hides the other along some paths (10.2). There is no such disambiguation when considering the set of names found as a result of following using-directives.
The \texttt{asm} declaration is conditionally-supported; its meaning is implementation-defined. \[\textit{Note:} Typically it is used to pass information through the implementation to an assembler. \textit{— end note}\]

7.5 Linkage specifications

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

2 Linkage (3.5) between C++ and non-C++ code fragments can be achieved using a \textit{linkage-specification}:

\begin{verbatim}
linkage-specification:
  extern string-literal \{ declaration-seq_opt \}
  extern string-literal declaration
\end{verbatim}

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. \[\textit{Note:} Therefore, a linkage-specification with a \textit{string-literal} that is unknown to the implementation requires a diagnostic. \textit{— end note}\] \[\textit{Note:} It is recommended that the spelling of the \textit{string-literal} be taken from the document defining that language. For example, \texttt{Ada} (not \texttt{ADA}) and \texttt{Fortran} or \texttt{FORTRAN}, depending on the vintage. \textit{— end note}\]

3 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); \hspace{1cm} // C++ linkage by default
extern "C" {
  double sqrt(double); \hspace{1cm} // C linkage
}
\end{verbatim}

\textit{— end example}\]

4 Linkage specifications nest. When linkage specifications nest, the innermost one determines the language linkage. A linkage specification does not establish a scope. A \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)); \hspace{1cm} // the name \texttt{f1} and its function type have C language
  // linkage; \texttt{pf} is a pointer to a C function
extern "C" typedef void FUNC();

FUNC f2; \hspace{1cm} // the name \texttt{f2} has C++ language linkage and the
  // function's type has C language linkage
extern "C" FUNC f3; \hspace{1cm} // the name of function \texttt{f3} and the function's type
  // have C language linkage
void (*pf2)(FUNC*); \hspace{1cm} // the name of the variable \texttt{pf2} has C++ linkage and
  // the type of \texttt{pf2} is pointer to C++ function that
  // takes one parameter of type pointer to C function
extern "C" {
  static void f4(); \hspace{1cm} // the name of the function \texttt{f4} has
    // internal linkage (not C language
    // linkage) and the function's type
\end{verbatim}
// has C language linkage.

extern "C" void f5() {  // OK: Name linkage (internal)
    extern void f4();   // and function type linkage (C
    // language linkage) gotten from
    // previous declaration.
}

extern void f4();        // OK: Name linkage (internal)
// and function type linkage (C
// language linkage) gotten from
// previous declaration.

void f6() {  // OK: Name linkage (internal)
    extern void f4();   // 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
5 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]
A function can be declared without a linkage specification after an explicit linkage specification has been seen; the linkage explicitly specified in the earlier declaration is not affected by such a function declaration.

6 At most one function with a particular name can have C language linkage. Two declarations for a function with C language linkage with the same function name (ignoring the namespace names that qualify it) that appear in different namespace scopes refer to the same function. Two declarations for an object with C language linkage with the same name (ignoring the namespace names that qualify it) that appear in different namespace scopes refer to the same object. [ Note: because of the one definition rule (3.2), only one definition for a function or object with C linkage may appear in the program; that is, such a function or object must not be defined in more than one namespace scope. For example,

```c
namespace A {
    extern "C" int f();
    extern "C" int g() { return 1; }
    extern "C" int h();
}

namespace B {
    extern "C" int f(); // A::f and B::f refer
    extern "C" int g() { return 1; } // to the same function
                              // ill-formed, the function g
    extern "C" int h(); // with C language linkage
                        // has two definitions
}

int A::f() { return 98; } //definition for the function f
                        // with C language linkage
extern "C" int h() { return 97; } // definition for the function h
                        // with C language linkage
                        // A::h and ::h refer to the same function

— end note]

7 A declaration directly contained in a linkage-specification is treated as if it contains the extern specifier (7.1.1) for the purpose of determining the linkage of the declared name and whether it is a definition. Such a declaration shall not specify a storage class. [ Example:

```c
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]

8 Linkage from C++ to objects defined in other languages and to objects defined in C++ from other languages is implementation-defined and language-dependent. Only where the object layout strategies of two language
implementations are similar enough can such linkage be achieved.

7.6 Attributes

7.6.1 Attribute syntax and semantics

1 Attributes specify additional information for various source constructs such as types, variables, names, blocks, or translation units.

    attribute-specifier:
        [ [ attribute-list ] ]
    attribute-list:
        attribute_opt
        attribute-list , attribute_opt
        attribute ...
        attribute-list , attribute ...
    attribute:
        attribute-token attribute-argument-clause_opt
    attribute-token:
        identifier
        attribute-scoped-token
    attribute-scoped-token:
        attribute-namespace :: identifier
    attribute-namespace:
        identifier
    attribute-argument-clause:
        ( balanced-token-seq )
    balanced-token-seq:
        balanced-token
        balanced-token-seq balanced-token
    balanced-token:
        ( balanced-token-seq )
        [ balanced-token-seq ]
        { balanced-token-seq }
        any token other than a parenthesis, a bracket, or a brace

2 [ Note: For each individual attribute, the form of the balanced-token-seq will be specified. — end note ]

3 In an attribute-list, an ellipsis may appear only if that attribute’s specification permits it. An attribute followed by an ellipsis is a pack expansion (14.5.3). An attribute-specifier that contains no attributes has no effect. The order in which the attribute-tokens appear in an attribute-list is not significant. If a keyword (2.12) or an alternative token (2.6) that satisfies the syntactic requirements of an identifier (2.11) is contained in an attribute-token, it is considered an identifier. No name lookup (3.4) is performed on any of the identifiers contained in an attribute-token. The attribute-token determines additional requirements on the attribute-argument-clause (if any). The use of an attribute-scoped-token is conditionally-supported, with implementation-defined behavior. [ Note: Each implementation should choose a distinctive name for the attribute-namespace in an attribute-scoped-token. — end note ]

4 Each attribute-specifier is said to appertain to some entity or statement, identified by the syntactic context where it appears (clause 7, clause 8). If an attribute-specifier that appertains to some entity or statement contains an attribute that is not allowed to apply to that entity or statement, the program is ill-formed. If an attribute-specifier appertains to a friend declaration (11.4), that declaration shall be a definition. No attribute-specifier shall appertain to an explicit instantiation (14.7.2).
For an attribute-token not specified in this International Standard, the behavior is implementation-defined.

Two consecutive left square bracket tokens shall appear only when introducing an attribute-specifier. [Note: If two consecutive left square brackets appear where an attribute-specifier is not allowed, the program is ill-formed even if the brackets match an alternative grammar production. — end note] [Example:

```c
int p[10];
void f() {
    int x = 42, y[5];
    int(p[x] { return x; }()); // error: malformed attribute on a nested
    // declarator-id and not a function-style cast of
    // an element of p.
    y[[] { return 2; }()] = 2; // error even though attributes are not allowed
    // in this context.
}
```
— end example]

7.6.2 Alignment attribute [dcl.align]

The attribute-token align specifies alignment (3.11). The attribute shall have one of the following forms:

```c
align ( type-id )
align ( assignment-expression )
```

The attribute may be followed by an ellipsis. The attribute may be applied to a variable that is neither a function parameter nor declared with the register storage class specifier and to a class data member that is not a bit-field. The attribute may also be applied to the declaration of a class or enumeration type.

When the alignment attribute is of the form align( assignment-expression ),

— the assignment-expression shall be an integral constant expression
— if the constant expression evaluates to a fundamental alignment, the alignment requirement of the declared entity 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 entity shall be that alignment
— if the constant expression evaluates to an extended alignment and the implementation does not support that alignment in the context of the declaration, the program is ill-formed
— if the constant expression evaluates to zero, the alignment specifier shall have no effect
— otherwise, the program is ill-formed.

When the alignment attribute is of the form align( type-id ), it shall have the same effect as align(alignmentof(type-id )) (5.3.6).

When multiple alignment attributes are specified for an entity, the alignment requirement shall be set to the strictest specified alignment.

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

If the defining declaration of an entity has an alignment attribute, any non-defining declaration of that entity shall either specify equivalent alignment or have no alignment attribute. Conversely, if any declaration of an entity has an alignment attribute, every defining declaration of that entity shall specify an equivalent alignment. No diagnostic is required if declarations of an entity have different alignment attributes in different translation units.

§ 7.6.2
// Translation unit #1:
struct S { int x; } s, p = &s;

// Translation unit #2:
struct [[align(16)]] S; // error: definition of S lacks alignment; no extern S* p; // diagnostic required
— end example]

Example: An aligned buffer with an alignment requirement of \( A \) and holding \( N \) elements of type \( T \) other than \text{char}, \text{signed char}, \text{or unsigned char} \ can be declared as:

\[
T \ \text{buffer} \ [[\ \text{align(}T\), \text{align(}A\))] \ [N];
\]

Specifying \text{align(}T\) in the attribute-list ensures that the final requested alignment will not be weaker than \text{alignof(}T\), and therefore the program will not be ill-formed. — end example]

Example:

\[
\text{void } f \ [[\ \text{align(double)}]] \ () \ { // error: alignment applied to function
\text{unsigned char } c\ [[\ \text{align(double)}]] \ [\text{sizeof(double)}]; // array of characters, suitably aligned for a double
\text{extern unsigned char } c[\text{sizeof(double)}]; // no align necessary
\text{extern unsigned char } c\ [[\ \text{align(float)}]] \ [\text{sizeof(double)}]; // error: different alignment in declaration
\}
— end example]

7.6.3 Noreturn attribute [dcl.attr.noreturn]

1 The \text{attribute-token \ noreturn} specifies that a function does not return. It shall appear at most once in each attribute-list and no attribute-argument-clause shall be present. The attribute may be applied to the \text{declarator-id} in a function declaration. The first declaration of a function shall specify the \text{noreturn} attribute if any declaration of that function specifies the \text{noreturn} attribute. If a function is declared with the \text{noreturn} attribute in one translation unit and the same function is declared without the \text{noreturn} attribute in another translation unit, the program is ill-formed; no diagnostic required.

2 If a function \( f \) is called where \( f \) was previously declared with the \text{noreturn} attribute and \( f \) eventually returns, the behavior is undefined. [Note: The function may terminate by throwing an exception. — end note] [Note: Implementations are encouraged to issue a warning if a function marked [[\text{noreturn}]] might return. — end note]

3 [Example:

\[
\text{void } f \ [[\ \text{noreturn}]] \ () \ { \\
\text{throw } \text{"error"}; // OK
\}
\]

\[
\text{void } q \ [[\ \text{noreturn}]] \ (\text{int } i) \ { // behavior is undefined if called with an argument } \leq 0
\text{if } (i > 0)
\text{throw } \text{"positive"};
\}
\]
7.6.4 Final attribute  

The attribute-token final specifies derivation semantics for a class and overriding semantics for a virtual function. It shall appear at most once in each attribute-list and no attribute-argument-clause shall be present. The attribute may be applied to class definitions and to virtual member functions being declared in a class definition.

1 If a class B is marked final and a class D is derived from B the program is ill formed.

2 If a virtual member function f in some class B is marked final and in a class D derived from B a function D::f overrides B::f, the program is ill-formed.

[Example:  

```c
struct B1 {
  virtual void f [ [ final ] ] ();
};

struct D1 : B1 {
  void f(); // ill formed
};

struct [ [ final ] ] B2 {
};

struct D2 : B2 { // ill formed
};
```
— end example]

7.6.5 Class member name checking attributes

The attribute-token override asserts that a virtual member function overrides a function in a base class. It shall appear at most once in each attribute-list and no attribute-argument-clause shall be present. The attribute may be applied to virtual member functions being declared in a class definition.

1 If a virtual member function f is marked override and does not override (10.3) a member function of a base class the program is ill-formed.

2 The attribute-token hiding asserts that a class member name hides a name in a base class. It shall appear at most once in each attribute-list and no attribute-argument-clause shall be present. The attribute may be applied to class members being declared in a class definition.

3 If a class member is marked hiding and its name does not hide (3.3.10, 10.2) a class member name in a base class the program is ill-formed.

4 The attribute-token base_check specifies that overriding and hiding of base members is strictly checked within a class. It shall appear at most once in each attribute-list and no attribute-argument-clause shall be present. The attribute may be applied to a class definition.

5 In a class definition marked base_check, if a virtual member function that is neither implicitly-declared nor a destructor overrides (10.3) a member function of a base class and it is not marked override, the program is ill-formed. Similarly, in such a class definition, if a class member name other than that of an implicitly-declared special member function hides (3.3.10, 10.2) a class member name in a base class and it is not marked hiding, the program is ill-formed. [Note: a using-declaration makes the potentially hidden name visible, avoiding the need for the hiding attribute. —end note]
[Example:

class B {
    virtual void some_func();
    virtual void f(int);
    virtual void h(int);
    void j(int);
    void k();
    typedef B self;
};

class D [[base_check]] : public B {
    void sone_func [[override]](); // error: misspelled name
    void f [[override]](int); // OK: f implicitly virtual, overrides B::f
    virtual void f [[override]](long); // error: non-matching argument type
    virtual void f [[override]](int) const; // error: non-matching cv-qualification
    virtual void f [[override]](int); // error: non-matching return type
    virtual void g(long); // OK: new virtual function introduced
    void h(int); // error: h implicitly virtual, but overriding without marker
    virtual void h(double); // error: hides B::h without marker
    virtual void h [[hiding]] (char *); // OK
    using B::j;
    int j(double); // OK: not hiding due to “using”
    void j(int); // OK, despite ‘obscuring’ B::j(int)
    virtual int j [[hiding]] (void); // error: not hiding due to “using”
    int k; // error: hides B::k without marker
    int m [[hiding]] (int); // error: no hiding despite marker
    typedef D self; // error: hides B::self without marker
};

—end example]

7.6.6 Carries dependency attribute

1 The attribute-token carries_dependency specifies dependency propagation into and out of functions. It shall appear at most once in each attribute-list and no attribute-argument-clause shall be present. The attribute may be applied to the declarator-id of a parameter-declaration in a function declaration or lambda, in which case it specifies that the initialization of the parameter carries a dependency to (1.10) each lvalue-to-rvalue conversion (4.1) of that object. The attribute may also be applied to the declarator-id of a function declaration, in which case it specifies that the return value, if any, carries a dependency to the evaluation of the function call expression.

2 The first declaration of a function shall specify the carries_dependency attribute for its declarator-id if any declaration of the function specifies the carries_dependency attribute. Furthermore, the first declaration of a function shall specify the carries_dependency attribute for a parameter if any declaration of that function specifies the carries_dependency attribute for that parameter. If a function or one of its parameters is declared with the carries_dependency attribute in its first declaration in one translation unit and the
same function or one of its parameters is declared without the `carries_dependency` attribute in its first declaration in another translation unit, the program is ill-formed; no diagnostic required.

3 [Note: the `carries_dependency` attribute does not change the meaning of the program, but may result in generation of more efficient code. — end note]

4 [Example:

```c
/* Translation unit A. */

struct foo { int* a; int* b; };  
std::atomic<struct foo*> foo_head[10];
int foo_array[10][10];

struct foo* f [[carries_dependency]] (int i) {
    return foo_head[i].load(memory_order_consume);
}

int g(int* x, int* y [[carries_dependency]]) {
    return kill_dependency(foo_array[*x][*y]);
}

/* Translation unit B. */

struct foo* f [[carries_dependency]] (int i);
int* g(int* x, int* y [[carries_dependency]]);

int c = 3;

void h(int i) {
    struct foo* p;

    p = f(i);
    do_something_with(g(&c, p->a));
    do_something_with(g(p->a, &c));
}
```

5 The `carries_dependency` attribute on function `f` means that the return value carries a dependency out of `f`, so that the implementation need not constrain ordering upon return from `f`. Implementations of `f` and its caller may choose to preserve dependencies instead of emitting hardware memory ordering instructions (a.k.a. fences).

6 Function `g`'s second argument has a `carries_dependency` attribute, but its first argument does not. Therefore, function `h`'s first call to `g` carries a dependency into `g`, but its second call does not. The implementation might need to insert a fence prior to the second call to `g`. — end example]
8 Declarators

A declarator declares a single variable, function, or type, within a declaration. The *init-declarator-list* appearing in a declaration is a comma-separated sequence of declarators, each of which can have an initializer.

\[
\text{init-declarator-list:}
\begin{align*}
\text{init-declarator} \\
\text{init-declarator-list , init-declarator}
\end{align*}
\]

The three components of a *simple-declaration* are the attributes (7.6), the specifiers (*decl-specifier-seq; 7.1*) and the declarators (*init-declarator-list*). The specifiers indicate the type, storage class or other properties of the entities being declared. The declarators specify the names of these entities and (optionally) modify the type of the specifiers with operators such as * (pointer to) and () (function returning). Initial values can also be specified in a declarator; initializers are discussed in 8.5 and 12.6.

Each *init-declarator* in a declaration is analyzed separately as if it was in a declaration by itself.\(^{96}\)

Declarators have the syntax

\[
declarator:
\begin{align*}
\text{ptr-declarator} \\
\text{noptr-declarator parameters-and-qualifiers trailing-return-type}
\end{align*}
\]

\[
\text{ptr-declarator:}
\begin{align*}
\text{noptr-declarator} \\
\text{ptr-operator ptr-declarator}
\end{align*}
\]

\[
\text{noptr-declarator:}
\begin{align*}
\text{declarator-id attribute-specifier_{opt}} \\
\text{noptr-declarator parameters-and-qualifiers} \\
\text{noptr-declarator [ constant-expression_{opt} ] attribute-specifier_{opt}} \\
\text{ ( ptr-declarator )}
\end{align*}
\]

\[
\text{parameters-and-qualifiers:}
\begin{align*}
\text{ ( parameter-declaration-clause ) attribute-specifier_{opt} cv-qualifier-seq_{opt}} \\
\text{ref-qualifier exception-specification_{opt}}
\end{align*}
\]

\[
\text{trailing-return-type:}
\begin{align*}
\rightarrow \text{trailing-type-specifier-seq abstract-declarator_{opt}}
\end{align*}
\]

\(^{96}\) A declaration with several declarators is usually equivalent to the corresponding sequence of declarations each with a single declarator. That is

\[
T \ D_1, D_2, \ldots, D_n;
\]

is usually equivalent to

\[
T \ D_1; T \ D_2, \ldots, T \ D_n;
\]

where \(T\) is a *decl-specifier-seq* and each \(D_i\) 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

\[
\text{struct } S \ldots ;
\]

\[
S \ S, T; // \text{declare two instances of struct } S
\]

which is not equivalent to

\[
\text{struct } S \ldots ;
\]

\[
S \ S;
\]

\[
S \ T; // \text{error}
\]
ptr-operator:
  * attribute-specifier<sup>opt</sup> cv-qualifier-seq<sup>opt</sup>
  & attribute-specifier<sup>opt</sup>
  && attribute-specifier<sup>opt</sup>
  ::<sup>opt</sup> nested-name-specifier * attribute-specifier<sup>opt</sup> cv-qualifier-seq<sup>opt</sup>

cv-qualifier-seq:
  cv-qualifier cv-qualifier-seq<sup>opt</sup>

cv-qualifier:
  const
  volatile

ref-qualifier:
  &
  &&

declarator-id:
  ...<sup>opt</sup> id-expression
  ::<sup>opt</sup> nested-name-specifier<sup>opt</sup> class-name

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).

5 The optional attribute-specifier in a trailing-return-type appertains to the indicated return type. The type-id in a trailing-return-type includes the longest possible sequence of abstract-declarators. [Note: This resolves the ambiguous binding of array and function declarators. [Example:]

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

— end example] — end note]

8.1 Type names [dcl.name]

1 To specify type conversions explicitly, and as an argument of sizeof, alignof, new, or typeid, the name of a type shall be specified. This can be done with a type-id, which is syntactically a declaration for a variable or function of that type that omits the name of the entity.

type-id:
  type-specifier-seq abstract-declarator<sup>opt</sup>

abstract-declarator:
  ptr-abstract-declarator
  noptr-abstract-declarator<sup>opt</sup> parameters-and-qualifiers trailing-return-type
  ...

ptr-abstract-declarator:
  noptr-abstract-declarator
  ptr-operator ptr-abstract-declarator<sup>opt</sup>

noptr-abstract-declarator:
  noptr-abstract-declarator<sup>opt</sup> parameters-and-qualifiers
  noptr-abstract-declarator<sup>opt</sup> [ constant-expression ] attribute-specifier<sup>opt</sup>
  ( ptr-abstract-declarator )

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

```c
int // int i
int * // int *pi
int *[3] // int *p[3]
```
name respectively the types “int,” “pointer to int,” “array of 3 pointers to int,” “pointer to array of 3 int,” “function of (no parameters) returning pointer to int,” and “pointer to a function of (double) returning int.” — end example]  

2 A type can also be named (often more easily) by using a typedef (7.1.3).

### 8.2 Ambiguity resolution

1 The ambiguity arising from the similarity between a function-style cast and a declaration mentioned in 6.8 can also occur in the context of a declaration. In that context, the choice is between a function declaration with a redundant set of parentheses around a parameter name and an object declaration with a function-style cast as the initializer. Just as for the ambiguities mentioned in 6.8, the resolution is to consider any construct that could possibly be a declaration a declaration. [Note: a declaration can be explicitly disambiguated by a nonfunction-style cast, by an = to indicate initialization or by removing the redundant parentheses around the parameter name. — end note] [Example:

```c
struct S {
    S(int);
};

void foo(double a) {
    S w(int(a));   // function declaration
    S x(int());    // function declaration
    S y((int)a);   // object declaration
    S z = int(a);  // object declaration
}
```

— end example]

2 The ambiguity arising from the similarity between a function-style cast and a type-id can occur in different contexts. The ambiguity appears as a choice between a function-style cast expression and a declaration of a type. The resolution is that any construct that could possibly be a type-id in its syntactic context shall be considered a type-id.

3 [Example:

```c
#include <cstddef>
char *p;
void *operator new(std::size_t, int);
void foo() {
    const int x = 63;
    new (int(*p)) int;   // new-placement expression
    new (int[*x]);      // new type-id
}
```

4 For another example,

```c
#include <class T>
struct S {
    T *p;
};
S<int()> x;   // type-id
S<int(1)> y;  // expression (ill-formed)
```
For another example,

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

For another example,

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

— end example]

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

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

int g(C);

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

For another example,

```c
class C { };  
void h(int *(C[10])); // void h(int *(fp)(C parm[10]));  
                        // not: void h(int *[10]);
```

— end example]

### 8.3 Meaning of declarators [dcl.meaning]

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 an 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, in the case of a namespace, of an element of the inline namespace set of that namespace (7.3.1)) or to a specialization thereof; the member shall not merely have been introduced by a using-declaration in the scope of the class or namespace nominated by the nested-name-specifier of the declarator-id. The nested-name-specifier of a qualified declarator-id shall not begin with a decltype-specifier. [Note: if the
qualifier is the global :: scope resolution operator, the declarator-id refers to a name declared in the global namespace scope. — end note] The optional attribute-specifier following a declarator-id appertains to the entity that is declared.

2 A static, thread_local, extern, register, mutable, friend, inline, virtual, or typedef specifier applies directly to each declarator-id in an init-declarator-list; the type specified for each declarator-id depends on both the decl-specifier-seq and its declarator.

3 Thus, a declaration of a particular identifier has the form

T D

where T is of the form attribute-specifier_opt decl-specifier-seq and D is a declarator. Following is a recursive procedure for determining the type specified for the contained declarator-id by such a declaration.

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

T D

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

int unsigned i;

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

5 In a declaration attribute-specifier_opt T D where D is an unadorned identifier the type of this identifier is “T”.

6 In a declaration T D where D has the form

(D1)

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

T D1

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

8.3.1 Pointers [decl.ptr]

1 In a declaration T D where D has the form

* attribute-specifier_opt cv-qualifier-seq_opt D1

and the type of the identifier in the declaration T D1 is “derived-declarator-type-list T,” then the type of the identifier of D is “derived-declarator-type-list cv-qualifier-seq pointer to T.” The cv-qualifiers apply to the pointer and not to the object pointed to. Similarly, the optional attribute-specifier (7.6.1) appertains 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
Examples of ill-formed operations are

- \( i = ci; \) // error
- \( *cp = ci; \) // error
- \( pc++; \) // error
- \( pc = cpc; \) // error
- \( pc = p; \) // error
- \( ppc = &pc; \) // error

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

- \( *ppc = &ci; \) // OK, but would make \( p \) point to \( ci \) ...
- \( *p = 5; \) // clobber \( ci \)

— end example ]

See also 5.17 and 8.5.

[ 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

- `& attribute-specifieropt D1`
- `&& attribute-specifieropt D1`

and the type of the identifier in the declaration `T D1` is “`derived-declarator-type-list T`,” then the type of the identifier of `D` is “`derived-declarator-type-list reference to T`.” The optional `attribute-specifier` appertains to the reference type. Cv-qualified references are ill-formed except when the cv-qualifiers are introduced through the use of a typedef (7.1.3) or of a template type argument (14.3), in which case the cv-qualifiers are ignored.

[ Example:

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

The type of `aref` is “lvalue reference to `int`”, not “lvalue reference to `const int`”. — end example ] [ Note: a reference can be thought of as a name of an object. — end note ] A declarator that specifies the type “reference to `cv void`” is ill-formed.

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

[ 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;
    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]

6 If a typedef (7.1.3), a type template-parameter (14.3.1), or a decltype-specifier (7.1.6.2) denotes a type TR that is a reference to a type T, an attempt to create the type “lvalue reference to cv TR” creates the type “lvalue reference to T”, while an attempt to create the type “rvalue reference to cv TR” creates the type TR. [Example:

    int i;
    typedef int& LRI;
    typedef int&& RRI;

§ 8.3.2
LRI& r1 = i;            // r1 has the type int&
const LRI& r2 = i;      // r2 has the type int&
const LRI& r3 = i;      // r3 has the type int&

RRI& r4 = i;            // r4 has the type int&
RRI& r5 = i;            // r5 has the type int&&

decltype(r2)& r6 = i;   // r6 has the type int&
decltype(r2)& r7 = i;   // r7 has the type int&

— end example ]

8.3.3 Pointers to members [dcl.mptr]

1 In a declaration T D where D has the form

    ::opt nested-name-specifier * attribute-specifieropt cv-qualifier-seqopt D1

and the nested-name-specifier denotes a class, and the type of the identifier in the declaration T D1 is “derived-declarator-type-list T”, then the type of the identifier of D is “derived-declarator-type-list cv-qualifier-sequence pointer to member of class nested-name-specifier of type T”. The optional attribute-specifier (7.6.1) appertains to the pointer-to-member.

2 [Example:

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

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

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

    X obj;
    // ...
    obj.*pmi = 7;     // assign 7 to an integer
                        // member of obj
    (obj.*pmf)(7);    // call a function member of obj
                        // with the argument 7

— end example ]

3 A pointer to member shall not point to a static member of a class (9.4), a member with reference type, or “cv void.”
8.3.4 Arrays

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

\[
D_1 \ [ \ constant-expression_{opt} \ ] \ attribute-specifier_{opt}
\]

and the type of the identifier in the declaration \( T \ D_1 \) is “derived-declarator-type-list \( T \)”, then the type of the identifier of \( D \) is an array type; if the type of the identifier of \( D \) contains the \( \text{auto} \) type-specifier, the program is ill-formed. \( T \) is called the array element type; this type shall not be a reference type, the (possibly cv-qualified) type \( \text{void} \), a function type or an abstract class type. If the \( constant-expression \) (5.19) is present, it shall be an integral constant expression and its value shall be greater than zero. The constant expression specifies the bound of (number of elements in) the array. If the value of the constant expression is \( N \), the array has \( N \) elements numbered 0 to \( N-1 \), and the type of the identifier of \( D \) is “derived-declarator-type-list array of \( N \ T \)”. 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 “derived-declarator-type-list array of unknown bound of \( T \)”, an incomplete object type. The type “derived-declarator-type-list array of \( N \ T \)” is a different type from the type “derived-declarator-type-list array of unknown bound of \( T \)”, see 3.9. Any type of the form “cv-qualifier-seq array of \( N \ T \)” is adjusted to “array of \( N \ cv-qualifier-seq \ T \)”, and similarly for “array of unknown bound of \( T \)”. The optional \( attribute-specifier \) appertains to the array.

\[ \text{Example:} \]

```c
typedef int A[5], AA[2][3];
typedef const A CA;               // type is “array of 5 const int”
typedef const AA CAA;             // type is “array of 2 array of 3 const int”
```

— end example [ Note: an “array of \( N \ cv-qualifier-seq \ T \)” has cv-qualified type; see 3.9.3. — end note ]

2 An array can be constructed from one of the fundamental types (except \( \text{void} \)), 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 \( \text{initializer} \) (8.5). In this case the bound is calculated from the number of initial elements (say, \( N \)) supplied (8.5.1), and the type of the identifier of \( D \) is “array of \( N \ T \)”.

\[ \text{Example:} \]

```c
float fa[17], *afp[17];
```

declares an array of \( \text{float} \) numbers and an array of pointers to \( \text{float} \) numbers. For another example,

```c
static int x3d[3][5][7];
```

declares a static three-dimensional array of integers, with rank \( 3 \times 5 \times 7 \). In complete detail, \( x3d \) is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions \( x3d, x3d[1], x3d[1][j], x3d[1][j][k] \) can reasonably appear in an expression. — end example

4 [ Note: conversions affecting expressions of array type are described in 4.2. Objects of array types cannot be modified, see 3.10. — end note ]
6 [ Note: Except where it has been declared for a class (13.5.5), the subscript operator \[\] is interpreted in such a way that E1[E2] is identical to *(E1)+(E2)). Because of the conversion rules that apply to *, if E1 is an array and E2 an integer, then E1[E2] refers to the E2-th member of E1. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

7 A consistent rule is followed for multidimensional arrays. If E is an n-dimensional array of rank \(i \times j \times \ldots \times k\), then E appearing in an expression that is subject to the array-to-pointer conversion (4.2) is converted to a pointer to an \((n-1)\)-dimensional array with rank \(j \times \ldots \times k\). 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[1] which is equivalent to *(x+i), x is first converted to a pointer as described; then x+i is converted to the type of x, which involves multiplying i by the length of the object to which the pointer points, namely five integer objects. The results are added and indirection applied to yield an array (of five integers), which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer. — end example] — end note

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

8.3.5 Functions

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 attribute-specifier_opt
```

and the type of the contained declarator-id in the declaration T D1 is “derived-declarator-type-list T”, the type of the declarator-id in D is “derived-declarator-type-list function of (parameter-declaration-clause ) cv-qualifier-seq_opt ref-qualifier_opt returning T”. The optional attribute-specifier appertains to the 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 attribute-specifier_opt trailing-return-type
```

and the type of the contained declarator-id in the declaration T D1 is “derived-declarator-type-list T”, T shall be the single type-specifier auto. The type of the declarator-id in D is “function of (parameter-declaration-clause) cv-qualifier-seq_ref returning type-id”. The optional attribute-specifier appertains to the function type.

3 A type of either form is a function type.97

```c
parameter-declaration-clause:
    parameter-declaration-list_opt ..._opt
    parameter-declaration-list , ...
```

```c
parameter-declaration-list:
    parameter-declaration
    parameter-declaration-list , parameter-declaration
```

97) As indicated by syntax, cv-qualifiers are a significant component in function return types.
parameter-declaration:
  attribute-specifieropt decl-specifier-seq declarator
  attribute-specifieropt decl-specifier-seq declarator = assignment-expression
  attribute-specifieropt decl-specifier-seq abstract-declaratoropt
  attribute-specifieropt decl-specifier-seq abstract-declaratoropt = assignment-expression

The optional attribute-specifier in a parameter-declaration appertains to the parameter.

4 The parameter-declaration-clause determines the arguments that can be specified, and their processing, when the function is called. [Note: the parameter-declaration-clause is used to convert the arguments specified on the function call; see 5.2.2. — end note] If the parameter-declaration-clause is empty, the function takes no arguments. 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

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

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

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

However, the first argument must be of a type that can be converted to a const char* — end example] [Note: the standard header <cstdarg> contains a mechanism for accessing arguments passed using the ellipsis (see 5.2.2 and 18.10). — end note]

5 A single name can be used for several different functions in a single scope; this is function overloading (Clause 13). All declarations for a function shall agree exactly in both the return type and the parameter-type-list. The type of a function is determined using the following rules. The type of each parameter (including function parameter packs) is determined from its own decl-specifier-seq and declarator. After determining the type of each parameter, any parameter of type “array of T” or “function returning T” is adjusted to be “pointer to T” or “pointer to function returning T,” respectively. After producing the list of parameter types, 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.

6 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. [Note: a function type that has a cv-qualifier-seq is not a cv-qualified type; there are no cv-qualified function types. — end note] 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:

    typedef void F();
    struct S {
        const F f; // OK: equivalent to: void f();
    };

§ 8.3.5
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]

7 [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]

8 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.98 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).

9 A typedef of function type may be used to declare a function but shall not be used to define a function (8.4).

[Example:]
```c
typedef void F();
F fv; // OK: equivalent to void fv();
F fv {} // ill-formed
void fv {} // OK: definition of fv
```

[end example] A typedef of a function type whose declarator includes a cv-qualifier-seq shall be used only to declare the function type for a non-static member function, to declare the function type to which a pointer to member refers, or to declare the top-level function type of another function typedef declaration.

[Example:]
```c
typedef int FIC(int) const;
FIC f; // ill-formed: does not declare a member function
struct S {
    FIC f; // OK
};
FIC S::*pm = &S::f; // OK
```

— end example] 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 its function declarator because that is the extent of its potential scope (3.3.4). — end note]

10 [Example: the declaration]

98) 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.
int i,
    *pi,
    f(),
    *fpi(int),
    (*pif)(const char*, const char*),
    (*fpif(int))(int);

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

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

or

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

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

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

rather than

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

— end note]

12 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] 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 \texttt{parameter-declaration-clause} without a preceding comma. In this case, the ellipsis is parsed as part of the \texttt{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 \texttt{parameter-declaration-clause}.\(^99\)

\section*{8.3.6 Default arguments}[dcl.fct.default]  

If an expression is specified in a parameter declaration this expression is used as a default argument. Default arguments will be used in calls where trailing arguments are missing.

[Example: the declaration

\begin{verbatim}
void point(int = 3, int = 4);
\end{verbatim}

declares a function that can be called with zero, one, or two arguments of type \texttt{int}. It can be called in any of these ways:

\begin{verbatim}
point(1,2); point(1); point();
\end{verbatim}

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

A default argument expression shall be specified only in the \texttt{parameter-declaration-clause} of a function declaration or in a \texttt{template-parameter} (14.1). It shall not be specified for a parameter pack. If it is specified in a \texttt{parameter-declaration-clause}, it shall not occur within a \texttt{declarator} or \texttt{abstract-declarator} of a \texttt{parameter-declaration}.\(^{100}\)

For non-template functions, default arguments can be added in later declarations of a function in the same scope. Declarations in different scopes have completely distinct sets of default arguments. That is, declarations in inner scopes do not acquire default arguments from declarations in outer scopes, and vice versa. In a given function declaration, each parameter subsequent to a parameter with a default argument shall have a default argument supplied in this or a previous declaration or shall be a function parameter pack. A default argument shall not be redefined by a later declaration (not even to the same value). [Example:

\begin{verbatim}
void g(int = 0, ...); // OK, ellipsis is not a parameter so it can follow
void f(int, int);    // OK, calls f(3, 7)
void f(int, int = 7); // OK
void h() {           // from surrounding scope
  void f(int = 1, int); // error: does not use default
  f(3);                // OK
  void f(int, int);    // error: wrong number of arguments
  f(4);                // OK
  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 m() {            // has no defaults
  void f(int, int);   // error: wrong number of arguments
  f(4);               // OK
  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() {             //
\end{verbatim}

99\) One can explicitly disambiguate the parse either by introducing a comma (so the ellipsis will be parsed as part of the \texttt{parameter-declaration-clause}) or by introducing a name for the parameter (so the ellipsis will be parsed as part of the \texttt{declarator-idd}).

100\) This means that default arguments cannot appear, for example, in declarations of pointers to functions, references to functions, or \texttt{typedef} declarations.
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.

5 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 mem-
ber 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 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:

```c
class C {
    void f(int i = 3);
    void g(int i, int j = 99);
};

void C::f(int i = 3) {        // error: default argument already
}                        // specified in class scope
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:

```c
void f() {
    int i;
    extern void g(int x = i);   //error
    //...
}
```

§ 8.3.6
The keyword **this** shall not be used in a default argument of a member function. **[Example:**

```c
class A {
    void f(A* p = this) { }  // error
};
```

**— end example**]

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:**

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

**— end example**] Similarly, a non-static member shall not be used in a default argument expression, even if it is not evaluated, unless it appears as the id-expression of a class member access expression (5.2.5) or unless it is used to form a pointer to member (5.3.1). **[Example: the declaration of X::mem1() in the following example is ill-formed because no object is supplied for the non-static member X::a used as an initializer.**

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

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

```c
int f(int = 0);

void h() {
    int j = f(1);
    int k = f();         // OK, means f(0)
}

int (*p1)(int) = &f;
int (*p2)() = &f;      // error: type mismatch
```

**— end example**] When a declaration of a function is introduced by way of a using-declaration (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()
}
```

— end example]

8.4 Function definitions

8.4.1 In general

Function definitions have the form

```
function-definition:
    attribute-specifier_opt declspec-seq_opt declarator function-body
    attribute-specifier_opt declspec-seq_opt declarator = default ;
    attribute-specifier_opt declspec-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 optional `attribute-specifier` in a `function-definition` appertains to the function.

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

```
D1 ( parameter-declaration-clause ) cv-qualifier-seq_opt
    ref-qualifier_opt exception-specification_opt attribute-specifier_opt trailing-return-type_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

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

Here `int` is the `declspec-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,
void print(int a, int) {
    std::printf("a = %d\n", a);
}

— end note]

7 In the function-body, a function-local predefined variable denotes a block-scope object of static storage duration that is implicitly defined (see 3.3.3).

8 The function-local predefined variable __func__ is defined as if a definition of the form

```cpp
static const char __func__[]{"function-name "};
```

had been provided, where function-name is an implementation-defined string. It is unspecified whether such a variable has an address distinct from that of any other object in the program.\(^{101}\)

[Example:
```cpp
struct S {
    S() : s(__func__) { } // OK
    const char *s;
};
void f(const char * s = __func__); // error: __func__ is undeclared
```
— end example]

### 8.4.2 Explicitly-defaulted functions \([\text{dcl.fct.def.default}]\)

1 A function definition of the form:

```cpp
attribute-specifier_opt decl-specifier-seqopt declarator = default ;
```

is called an explicitly-defaulted definition. A function that is explicitly defaulted shall

— be a special member function,
— have the same declared function type (except for possibly differing ref-qualifiers and except that in the case of a copy constructor or copy assignment operator, the parameter type may be “reference to non-const T”, where T is the name of the member function’s class) as if it had been implicitly declared,
— not have default arguments, and
— not have an exception-specification.

2 [Note: This implies that parameter types, return type, and cv-qualifiers must match the hypothetical implicit declaration. — end note] An explicitly-defaulted function may be declared constexpr only if it would have been implicitly declared as constexpr. If it is explicitly defaulted on its first declaration,

— it shall be public,
— it shall not be explicit,
— it shall not be virtual,
— it is implicitly considered to have the same exception-specification as if it had been implicitly declared (15.4), and

\(^{101}\) Implementations are permitted to provide additional predefined variables with names that are reserved to the implementation (17.6.3.3.2). If a predefined variable is not used (3.2), its string value need not be present in the program image.
in the case of a copy constructor, move constructor, copy assignment operator, or move assignment 
operator, it shall have the same parameter type as if it had been implicitly declared.

3 [Note: Such a special member function may be trivial, and thus its accessibility and explicitness should 
match the hypothetical implicit definition; see below. — end note] [Example:

```cpp
struct S {
    S(int a = 0) = default; // ill-formed: default argument
    void operator=(const S&) = default; // ill-formed: non-matching return type
    ~S() throw() = default; // ill-formed: exception specification
    private:
    S(S&); // OK: private copy constructor
};
S::S(S&) = default; // OK: defines copy constructor
```

— end example]

Explicitly-defaulted functions and implicitly-declared functions are collectively called *defaulted* functions, 
and the implementation shall provide implicit definitions for them (12.1 12.4, 12.8), which might mean 
defining them as deleted. A special member function is *user-provided* if it is user-declared and not explicitly 
defaulted on its first declaration. A user-provided explicitly-defaulted function is defined at the point where 
it is explicitly defaulted; if such a function is implicitly defined as deleted, the program is ill-formed. [Note: 
while an implicitly-declared special member function is inline (Clause 12), an explicitly-defaulted definition 
may be non-inline. Non-inline definitions are user-provided, and hence non-trivial (12.1, 12.4, 12.8). This 
rule enables efficient execution and concise definition while enabling a stable binary interface to an evolving 
code base. — end note]

5 [Example:

```cpp
struct trivial {
    trivial() = default;
    trivial(const trivial&) = default;
    trivial(trivial&&) = default;
    trivial& operator=(const trivial&) = default;
    trivial& operator=(trivial&&) = default;
    ~trivial() = default;
};
```

```cpp
struct nontrivial1 {
    nontrivial1();
};
nontrivial1::nontrivial1() = default; // not inline
```

```cpp
struct nontrivial2 {
    nontrivial2();
};
inline nontrivial2::nontrivial2() = default; // not first declaration
```

```cpp
struct nontrivial3 {
    virtual ~nontrivial3() = 0; // virtual
};
inline nontrivial3::~nontrivial3() = default; // not first declaration
```

— end example]
8.4.3 Deleted definitions

A function definition of the form:

\[ \text{attribute-specifier}_{\text{opt}} \text{ decl-specifier-seq}_{\text{opt}} \text{ declarator} = \text{delete} ; \]

is called a deleted definition. A function with a deleted definition is also called a deleted function.

A program that refers to a deleted function implicitly or explicitly, other than to declare it, is ill-formed. [Note: This includes calling the function implicitly or explicitly and forming a pointer or pointer-to-member to the function. It applies even for references in expressions that are not potentially-evaluated. If a function is overloaded, it is referenced only if the function is selected by overload resolution. — end note]

[Example: One can enforce non-default initialization and non-integral initialization with

```c
struct sometype {
    sometype() = delete; // OK, but redundant
    some_type(std::intmax_t) = delete;
    some_type(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;
};
```

```c
sometype *p = new sometype; // error, deleted class operator new
sometype *q = new sometype[3]; // error, deleted class operator new[]
```

— end example]

[Example: One can make a class uncopyable, i.e. move-only, by using deleted definitions of the copy constructor and copy assignment operator, and then providing defaulted definitions of the move constructor and move assignment operator.

```c
struct moveonly {
    moveonly() = default;
    moveonly(const moveonly&) = delete;
    moveonly(moveonly&&) = default;
    moveonly& operator=(const moveonly&) = delete;
    moveonly& operator=(moveonly&&) = default;
    ~moveonly() = default;
};
```

```c
moveonly *p;
mmoveonly q(*p); // error, deleted copy constructor
```

— end example]

A deleted function is implicitly inline. [Note: The one-definition rule (3.2) applies to deleted definitions. — end note] A deleted definition of a function shall be the first declaration of the function or, for an explicit specialization of a function template, the first declaration of that specialization. [Example:

```c
struct sometype {
    sometype();
};
```
sometype::sometype() = delete; // ill-formed; not first declaration

— end example]

8.5 Initializers [dcl.init]

1 A declarator can specify an initial value for the identifier being declared. The identifier designates a variable being initialized. The process of initialization described in the remainder of 8.5 applies also to initializations specified by other syntactic contexts, such as the initialization of function parameters with argument expressions (5.2.2) or the initialization of return values (6.6.3).

\[
\text{initializer:} \\
\text{brace-or-equal-initializer} \\
\{ \text{expression-list} \}
\]

\[
\text{brace-or-equal-initializer:} \\
\text{=} \text{initializer-clause} \\
\text{braced-init-list}
\]

\[
\text{initializer-clause:} \\
\text{assignment-expression} \\
\text{braced-init-list}
\]

\[
\text{initializer-list:} \\
\text{initializer-clause} \ldots \text{opt} \\
\text{initializer-list} , \text{initializer-clause} \ldots \text{opt}
\]

\[
\text{braced-init-list:} \\
\{ \text{initializer-list} \text{, opt} \} \\
\{ \}
\]

2 Automatic, register, thread_local, static, and namespace-scoped external variables can be initialized by arbitrary expressions involving literals and previously declared variables and functions. [Example:

\[
\text{int f(int);} \\
\text{int a = 2;} \\
\text{int b = f(a);} \\
\text{int c(b);} \\
\]

— end example]

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

4 The order of initialization of variables with static storage duration is described in 3.6 and 6.7. — end note]

5 To **zero-initialize** an object or reference of type \( T \) means:

- if \( T \) is a scalar type (3.9), the object is set to the value 0 (zero), taken as an integral constant expression, converted to \( T \);\(^{102}\)
- if \( T \) is a (possibly cv-qualified) non-union class type, each non-static data member and each base-class subobject is zero-initialized;
- if \( T \) is a (possibly cv-qualified) union type, the object’s first non-static 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.

102) As specified in 4.10, converting an integral constant expression whose value is 0 to a pointer type results in a null pointer value.

§ 8.5
To **default-initialize** an object of type `T` means:

— if `T` is a (possibly cv-qualified) 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, no initialization is performed.

If a program calls for the default initialization of an object of a const-qualified type `T`, `T` shall be a class type with a user-provided default constructor.

To **value-initialize** an object of type `T` means:

— if `T` is a (possibly cv-qualified) 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 (possibly cv-qualified) non-union class type without a user-provided constructor, then the object is zero-initialized and, if `T`’s implicitly-declared default constructor is non-trivial, that constructor is called.
— 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.

[Note: Every object of static storage duration is zero-initialized at program startup before any other initialization takes place. In some cases, additional initialization is done later. — end note]

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*,

```cpp
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, the object is default-initialized; if no initialization is performed, an object with automatic or dynamic storage duration has indeterminate value. [Note: objects with static or thread storage duration are zero-initialized, see 3.6.2. — end note]

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

```cpp
int a;

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

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

— end example]

§ 8.5
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 the form

\[
T \ x = \ a;
\]

as well as in argument passing, function return, throwing an exception (15.1), handling an exception (15.3), and aggregate member initialization (8.5.1) is called \textit{copy-initialization}. \[ \text{Note: } \text{Copy-initialization may invoke a move (12.8). } \text{- end note} \]

The initialization that occurs in the forms

\[
T \ x(a);
\]
\[
T \ x\{a\};
\]

as well as in \texttt{new} expressions (5.3.4), \texttt{static\_cast} expressions (5.2.9), functional notation type conversions (5.2.3), and base and member initializers (12.6.2) is called \textit{direct-initialization}.

The semantics of initializers are as follows. The \textit{destination type} is the type of the object or reference being initialized and the \textit{source type} is the type of the initializer expression. The source type is not defined when the initializer is a \texttt{braced-init-list} or when it is a parenthesized list of expressions.

- If the initializer is a \texttt{braced-init-list}, the object is list-initialized (8.5.4).
- If the destination type is a reference type, see 8.5.3.
- If the destination type is an array of characters, an array of \texttt{char16\_t}, an array of \texttt{char32\_t}, or an array of \texttt{wchar\_t}, and the initializer is a string literal, see 8.5.2.
- If the initializer is \( () \), the object is value-initialized.
- Otherwise, if the destination type is an array, the program is ill-formed.
- If the destination type is a (possibly \texttt{cv}-qualified) class type:
  - If the initialization is direct-initialization, or if it is copy-initialization where the \texttt{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 \texttt{cv}-unqualified version of the destination type. The temporary is a prvalue. The result of the call (which is the temporary for the constructor case) is then used to direct-initialize, according to the rules above, the object that is the destination of the copy-initialization. In certain cases, an implementation is permitted to eliminate the copying inherent in this direct-initialization by constructing the intermediate result directly into the object being initialized; see 12.2, 12.8.
  - Otherwise, if the source type is a (possibly \texttt{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 \( \text{cv1 T} \) can initialize an object of type \( \text{cv2 T} \) independently of the cv-qualifiers \( \text{cv1} \) and \( \text{cv2} \). \]

\[
\begin{aligned}
\text{int } &a; \\
\text{const int } &b = a; \\
\text{int } &c = b;
\end{aligned}
\]

— end note]

17 An initializer-clause followed by an ellipsis is a pack expansion (14.5.3).

8.5.1 Aggregates [dcl.init.aggr]

1 An aggregate is an array or a class (Clause 9) with no user-provided constructors (12.1), no brace-or-equal-initializers for non-static data members (9.2), no private or protected non-static data members (Clause 11), no base classes (Clause 10), and no virtual functions (10.3).

2 When an aggregate is initialized by an initializer list, as specified in 8.5.4, the elements of the initializer list are taken as initializers for the members of the aggregate, in increasing subscript or member order. Each member is copy-initialized from the corresponding initializer-clause. If the initializer-clause is an expression and a narrowing conversion (8.5.4) is required to convert the expression, the program is ill-formed. \[ Note: If an initializer-clause is itself an initializer list, the member is list-initialized, which will result in a recursive application of the rules in this section if the member is an aggregate. — end note] \[ Example:

\[
\begin{aligned}
\text{struct A } &{} \{ \\
\text{int } &x; \\
\text{struct B } &{} \{ \\
\text{int } &i; \\
\text{int } &j; \\
&{} \} \ b; \\
&{} \} \ a = \{ 1, \{ 2, 3 \} \};
\end{aligned}
\]

initializes \( \text{a.x} \) with 1, \( \text{a.b.i} \) with 2, \( \text{a.b.j} \) with 3. — end example]

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

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

\[
\text{int } x[] = \{ 1, 3, 5 \};
\]

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.\(^\text{103}\)

5 Static data members and anonymous bit fields are not considered members of the class for purposes of aggregate initialization. \[ Example:

\[
\text{struct A } \{ \\
\}
\]

\(^{103}\) The syntax provides for empty initializer-lists, but nonetheless C++ does not have zero length arrays.
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

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

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

is ill-formed. — end example]

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 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 uninitialzied, the program is ill-formed.

10 When initializing a multi-dimensional array, the *initializer-clauses* initialize the elements with the last (right-most) index of the array varying the fastest (8.3.4). [Example:

```c
int x[2][2] = { 3, 1, 4, 2 };
```

initializes x[0][0] to 3, x[0][1] to 1, x[1][0] to 4, and x[1][1] to 2. On the other hand,
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};

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

In a declaration of the form

T x = { a };

braces can be elided in an initializer-list as follows.\(^\text{104}\) If the initializer-list begins with a left brace, then the succeeding comma-separated list of initializer-clauses initializes the members of a subaggregate; it is erroneous for there to be more initializer-clauses than members. If, however, the initializer-list for a subaggregate does not begin with a left brace, then only enough initializer-clauses from the list are taken to initialize the members of the subaggregate; any remaining initializer-clauses are left to initialize the next member of the aggregate of which the current subaggregate is a member. [Example:

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

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

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

The initializer for y begins with a left brace, but the one for y[0] does not, therefore three elements from the list are used. Likewise the next three are taken successively for y[1] and y[2]. — end example]

12 All implicit type conversions (Clause 4) are considered when initializing the aggregate member with an assignment-expression. If the assignment-expression can initialize a member, the member is initialized. Otherwise, if the member is itself a subaggregate, brace elision is assumed and the assignment-expression is considered for the initialization of the first member of the subaggregate. [Note: As specified above, brace elision cannot apply to subaggregates with no members for purposes of aggregate initialization; an initializer-clause for the entire subobject is required. — end note]

[Example:

```c
struct A {
    int i;
    operator int();
};
struct B {
    A a1, a2;
    int z;
};
```
A a;
B b = { 4, a, a };

Braces are elided around the initializer-clause for b.a1.1. b.a1.i is initialized with 4, b.a2 is initialized with a, b.z is initialized with whatever a.operator int() returns. — end example]

[ Note: An aggregate array or an aggregate class may contain members of a class type with a user-provided constructor (12.1). Initialization of these aggregate objects is described in 12.6.1. — end note]

[ Note: Whether the initialization of aggregates with static storage duration is static or dynamic is specified in 3.6.2 and 6.7. — end note]

When a union is initialized with a brace-enclosed initializer, the braces shall only contain an initializer-clause for the first non-static data member of the union. [Example:

union u { int a; char* b; };
    u a = { 1 };
    u b = a;
    u c = 1;    // error
    u d = { 0, "asdf" };    // error
    u e = { "asdf" };    // error

— end example]

[ Note: As described above, the braces around the initializer-clause for a union member can be omitted if the union is a member of another aggregate. — end note]

The full-expressions in an initializer-clause are evaluated in the order in which they appear.

8.5.2 Character arrays [dcl.init.string]

1 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 narrow character literal, char16_t string literal, char32_t string literal, or wide string literal, respectively, or by an appropriately-typed string literal enclosed in braces. Successive characters of the value of the string literal initialize the elements of the array. [Example:

    char msg[] = "Syntax error on line %s
";

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

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

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

is ill-formed since there is no space for the implied trailing '\0'. — end example]

3 If there are fewer initializers than there are array elements, each element not explicitly initialized shall be zero-initialized (8.5).

8.5.3 References [dcl.init.ref]

1 A variable declared to be a T& or T&&, that is, “reference to type T” (8.3.2), shall be initialized by an object, or function, of type T or by an object that can be converted into a T. [Example:

    int g(int);
    void f() {
        int i;

§ 8.5.3
int& r = i;    // r refers to i
r = 1;        // the value of i becomes 1
int* p = &r;  // p points to i
int& rr = r;  // rr refers to what r refers to, that is, to i
int (&rg)(int) = g; // rg refers to the function g
g(i);          // calls function g
int a[3];      // 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.

A reference to type “cv1 T1” is initialized by an expression of type “cv2 T2” as follows:

— If the reference is an lvalue reference and 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), where T1 is not reference-related to T2, and can be implicitly converted to an lvalue of type “cv3 T3,” where “cv1 T1” is reference-compatible with “cv3 T3”105 (this conversion is selected by enumerating the applicable conversion functions (13.3.1.6) and choosing the best one through overload resolution (13.3)),

then the reference is bound to the initializer expression lvalue in the first case and to the lvalue result of the conversion in the second case (or, in either case, to the appropriate base class subobject of the object). [Note: the usual lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are not needed, and therefore are suppressed, when such direct bindings to lvalues are done. —end note]

[Example:

§ 8.5.3 203

105) This requires a conversion function (12.3.2) returning a reference type.
double d = 2.0;
double& rd = d; // rd refers to d
const double& rcd = d; // rcd refers to d

struct A { }
struct B : A { } b;
A& ra = b; // ra refers to A subobject in b
const A& rca = b; // rca refers to A subobject in b

— end example ]

— Otherwise, the reference shall be an lvalue reference to a non-volatile const type (i.e., cv1 shall be const), or the reference shall be an rvalue reference and the initializer expression shall be an rvalue or have a function type. [ 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; // error: rvalue reference cannot bind to lvalue

— end example ]

— If T1 is a function type, then

— if T2 is the same type as T1, the reference is bound to the initializer expression lvalue;

— if T2 is a class type and the initializer expression can be implicitly converted to an lvalue of type T1 (this conversion is selected by enumerating the applicable conversion functions (13.3.1.6) and choosing the best one through overload resolution (13.3)), the reference is bound to the function lvalue that is the result of the conversion;

— otherwise, the program is ill-formed.

— Otherwise, if T2 is a class type and

— the initializer expression is an rvalue and “cv1 T1” is reference-compatible with “cv2 T2”, or

— T1 is not reference-related to T2 and the initializer expression can be implicitly converted to an rvalue of type “cv3 T3” (this conversion is selected by enumerating the applicable conversion functions (13.3.1.6) and choosing the best one through overload resolution (13.3)),

then the reference is bound to the initializer expression rvalue in the first case and to the object that is the result of the conversion in the second case (or, in either case, to the appropriate base class subobject of the object).

[ Example:

struct A { }
struct B : A { } b;
extern B f();
const A& rca = f(); // bound to the A subobject of the B rvalue.
A& rcb = f(); // same as above
struct X {
    operator B();
} x;
const A& r = x; // bound to the A subobject of the result of the conversion

— end example ]
If the initializer expression is an rvalue, with \( T_2 \) an array type, and "\( \text{cv1} \ T_1 \)" is reference-compatible with "\( \text{cv2} \ T_2 \)," the reference is bound to the object represented by the rvalue (see 3.10).

Otherwise, a temporary of type "\( \text{cv1} \ T_1 \)" 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 \( T_1 \) is reference-related to \( T_2 \), \( \text{cv1} \) must be the same cv-qualification as, or greater cv-qualification than, \( \text{cv2} \); otherwise, the program is ill-formed. [Example:

\[
\begin{align*}
\text{const double}& \ rcd2 = 2; & \quad \text{\( rcd2 \) refers to temporary with value 2.0} \\
\text{double}& \& \ rcd3 = 2; & \quad \text{\( rcd3 \) refers to temporary with value 2.0} \\
\text{const volatile int} & \ cvi = 1; \\
\text{const int}& \& \ r = \ cvi; & \quad \text{error: type qualifiers dropped}
\end{align*}
\]

— end example]

In all cases except the last (i.e., creating and initializing a temporary from the initializer expression), the reference is said to bind directly to the initializer expression.

6 [Note: 12.2 describes the lifetime of temporaries bound to references. — end note]

### 8.5.4 List-initialization

List-initialization is initialization of an object or reference from a braced-init-list. Such an initializer is called an initializer list, and the comma-separated initializer-clauses of the list are called the elements of the initializer list. An initializer list may be empty. List-initialization can occur in direct-initialization or copy-initialization contexts; list-initialization in a direct-initialization context is called direct-list-initialization and list-initialization in a copy-initialization context is called copy-list-initialization. [Note: List-initialization can be used

— as the initializer in a variable definition (8.5)
— as the initializer in a new expression (5.3.4)
— in a return statement (6.6.3)
— as a function argument (5.2.2)
— as a subscript (5.2.1)
— as an argument to a constructor invocation (8.5, 5.2.3)
— as an initializer for a non-static data member (9.2)
— as a base-or-member initializer (12.6.2)
— on the right-hand side of an assignment (5.17)

[Example:

\[
\begin{align*}
\text{int} & \ a = \{1\}; \\
\text{std::complex<} & \text{double}> \ z\{1,2\}; \\
\text{new } \text{std::vector<} & \text{std::string}\{\text{"once", } \text{"upon", } \text{"a", } \text{"time"}\}; & \quad \text{\( 4 \) string elements} \\
f( \{\text{"Nicholas", } \text{"Annemarie"}\} ); & \quad \text{pass list of two elements} \\
\text{return } & \{\text{"Norah"}\}; & \quad \text{return list of one element} \\
\text{int}* & \ e \{}; & \quad \text{initialization to zero / null pointer} \\
x & = \text{double}\{1\}; & \quad \text{explicitly construct a double} \\
\text{std::map<} & \text{std::string,}\text{int}> \ anim = \{\{\text{"bear", }4\}, \{\text{"cassowary", }2\}, \{\text{"tiger", }7\}\};
\end{align*}
\]

§ 8.5.4
A constructor is an *initializer-list constructor* if its first parameter is of type `std::initializer_list<E>` or reference to possibly cv-qualified `std::initializer_list<E>` for some type `E`, and either there are no other parameters or else all other parameters have default arguments (8.3.6). [Note: Initializer-list constructors are favored over other constructors in list-initialization (13.3.1.7). — end note] The template `std::initializer_list` is not predefined; if the header `<initializer_list>` is not included prior to a use of `std::initializer_list` — even an implicit use in which the type is not named (7.1.6.4) — the program is ill-formed.

List-initialization of an object or reference of type `T` is defined as follows:

— If the initializer list has no elements and `T` is a class type with a default constructor, the object is value-initialized.

— Otherwise, if the initializer list has no elements and `T` is an aggregate, each of the members of `T` is initialized from an empty initializer list. [Example:

```cpp
struct A {
    A(std::initializer_list<int>);  // #1
};
struct B {
    A a;
};
B b { };  // OK, uses #1
B b{ 1 };  // error
```

— end example]

— Otherwise, if `T` is an aggregate, aggregate initialization is performed (8.5.1).

[Example:

```cpp
double ad[] = { 1, 2.0 };  // OK
int ai[] = { 1, 2.0 };     // error: narrowing

struct S2 {
    int m1;
    double m2, m3;
};
S2 s21 = { 1, 2, 3.0 };    // OK
S2 s22 { 1.0, 2, 3 };     // error: narrowing
S2 s23 { };               // OK: default to 0,0,0
```

— end example]

— Otherwise, if `T` is a specialization of `std::initializer_list<E>`, an `initializer_list` object is constructed as described below and used to initialize the object according to the rules for initialization of an object from a class of the same type (8.5).

— Otherwise, if `T` is a class type, constructors are considered. If `T` has an initializer-list constructor, the argument list consists of the initializer list as a single argument; otherwise, the argument list consists of the elements of the initializer list. The applicable constructors are enumerated (13.3.1.7) and the best one is chosen through overload resolution (13.3). If a narrowing conversion (see below) is required to convert any of the arguments, the program is ill-formed.

[Example:
struct S {
    S(std::initializer_list<double>);    // #1
    S(std::initializer_list<int>);       // #2
    S();                                 // #3
    // ...
};
S s1 = { 1.0, 2.0, 3.0 };              // invoke #1
S s2 = { 1, 2, 3 };                    // invoke #2
S s3 = { };                           // invoke #3

— end example

Example:

struct Map {
    Map(std::initializer_list<std::pair<std::string, int>>);
};
Map ship = {{"Sophie",14}, {"Surprise",28}};

— end example

Example:

struct S {
    // no initializer-list constructors
    S(int, double, double);                // #1
    S();                                  // #2
    // ...
};
S s1 = { 1, 2, 3.0 };                  // OK: invoke #1
S s2 { 1.0, 2, 3 };                    // error: narrowing
S s3 { };                             // OK: invoke #2

— end example

— Otherwise, if \( T \) is a reference to class type or if \( T \) is any reference type and the initializer list has no elements, a prvalue temporary of the type referenced by \( T \) is list-initialized, and the reference is bound to that temporary. [Note: As usual, the binding will fail and the program is ill-formed if the reference type is an lvalue reference to a non-const type. — end note]

Example:

struct S {
    S(std::initializer_list<double>);    // #1
    S(const std::string&);              // #2
    // ...
};
const S& r1 = { 1, 2, 3.0 };           // OK: invoke #1
const S& r2 { "Spinach" };            // OK: invoke #2
S& r3 = { 1, 2, 3 };                  // error: initializer is not an lvalue

— end example

— Otherwise, if the initializer list has a single element, the object is initialized from that element; if a narrowing conversion (see below) is required to convert the element to \( T \), the program is ill-formed.

[Example:}
int x1 {2}; // OK
int x2 {2.0}; // error: narrowing

— end example ]
— Otherwise, if the initializer list has no elements, the object is value-initialized.

[ Example:
  int** pp {};
  // initialized to null pointer
— end example ]
— Otherwise, the program is ill-formed.

[ Example:
  struct A { int i; int j; }
  A a1 { 1, 2 }; // aggregate initialization
  A a2 { 1.2 }; // error: narrowing
  struct B {
    B(std::initializer_list<int>);
  };
  B b1 { 1, 2 }; // creates initializer_list<int> and calls constructor
  B b2 { 1, 2.0 }; // error: narrowing
  struct C {
    C(int i, double j);
  };
  C c1 = { 1, 2.2 }; // calls constructor with arguments (1, 2.2)
  C c2 = { 1.1, 2 }; // error: narrowing
  int j { 1 }; // initialize to 1
  int k { }; // initialize to 0
— end example ]

4 An object of type std::initializer_list<E> is constructed from an initializer list as if the implementation allocated an array of N elements of type E, where N is the number of elements in the initializer list. Each element of that array is copy-initialized with the corresponding element of the initializer list, and the std::initializer_list<E> object is constructed to refer to that array. If a narrowing conversion is required to initialize any of the elements, the program is ill-formed.[ Example:

  struct X {
    X(std::initializer_list<double> v);
  };
  X x{ 1,2,3 };

  The initialization will be implemented in a way roughly equivalent to this:

  double __a[3] = {double{1}, double{2}, double{3}};
  X x(std::initializer_list<double>(__a, __a+3));

  assuming that the implementation can construct an initializer_list object with a pair of pointers. — end example ]

5 The lifetime of the array is the same as that of the initializer_list object. [ Example:
typedef std::complex<double> cmplx;
std::vector<cmplx> v1 = { 1, 2, 3 };
void f() {
    std::vector<cmplx> v2{ 1, 2, 3 };
    std::initializer_list<int> i3 = { 1, 2, 3 };
}

For v1 and v2, the initializer_list object and array created for { 1, 2, 3 } have full-expression lifetime. For i3, the initializer_list object and array have automatic lifetime. — end example] [Note: The implementation is free to allocate the array in read-only memory if an explicit array with the same initializer could be so allocated. — end note]

6 A narrowing conversion is an implicit conversion

— from a floating-point type to an integer type, or

— from long double to double or float, or from double to float, except where the source is a constant expression and the actual value after conversion is within the range of values that can be represented (even if it cannot be represented exactly), or

— from an integer type or unscoped enumeration type to a floating-point type, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted back to the original type, or

— from an integer type or unscoped enumeration type to an integer type that cannot represent all the values of the original type, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted back to the original type.

[Note: As indicated above, such conversions are not allowed at the top level in list-initializations. — end note] [Example:

int x = 999; // x is not a constant expression
const int y = 999;
const int z = 99;
char c1 = x; // OK, though it might narrow (in this case, it does narrow)
char c2{x}; // error: might narrow
char c3(y); // error: narrows (assuming char is 8 bits)
char c4(z); // OK: no narrowing needed
unsigned char uc1 = {5}; // OK: no narrowing needed
unsigned char uc2 = {-1}; // error: narrows
unsigned int ui1 = {-1}; // error: narrows
signed int si1 = { (unsigned int)-1 }; // error: narrows
int ii = {2.0}; // error: narrows
float f1 { x }; // error: might narrow
float f2 { 7 }; // OK: 7 can be exactly represented as a float
int f(int);
int a[] =
    { 2, f(2), f(2.0) }; // OK: the double-to-int conversion is not at the top level

— end example]
9 Classes

1 A class is a type. Its name becomes a class-name (9.1) within its scope.

   class-name:
   identifier
   simple-template-id

Class-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-specification_opt }

   class-head:
   class-key attribute-specifier_opt identifier_opt base-clause_opt
   class-key attribute-specifier_opt nested-name-specifier identifier base-clause_opt
   class-key attribute-specifier_opt nested-name-specifier_opt simple-template-id base-clause_opt

   class-key:
   class
   struct
   union

A class-specifier where the class-head omits the optional identifier defines an unnamed class.

2 A class-name is inserted into the scope in which it is declared immediately after the class-name is seen. The class-name is also inserted into the scope of the class itself; this is known as the injected-class-name. For purposes of access checking, the injected-class-name is treated as if it were a public member name. A class-specifier is commonly referred to as a class definition. A class is considered defined after the closing brace of its class-specifier has been seen even though its member functions are in general not yet defined. The optional attribute-specifier appertains to the class; the attributes in the attribute-specifier are thereafter considered attributes of the class whenever it is named.

3 Complete objects and member subobjects of class type shall have nonzero size.\[106\] [Note: Class objects can be assigned, passed as arguments to functions, and returned by functions (except objects of classes for which copying or moving has been restricted; see 12.8). Other plausible operators, such as equality comparison, can be defined by the user; see 13.5. — end note]

4 A union is a class defined with the class-key union; it holds only one data member at a time (9.5). [Note: aggregates of class type are described in 8.5.1. — end note]

5 A trivially copyable class is a class that:
   — has no non-trivial copy constructors (12.8),
   — has no non-trivial move constructors (12.8),
   — has no non-trivial copy assignment operators (13.5.3, 12.8),
   — has no non-trivial move assignment operators (13.5.3, 12.8), and
   — has a trivial destructor (12.4).

\[106\] Base class subobjects are not so constrained.

[210]
A **trivial class** is a class that has a trivial default constructor \((12.1)\) and is trivially copyable.

[Note: in particular, a trivially copyable or 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.\(^{107}\)

7 A **standard-layout struct** is a standard-layout class defined with the class-key \texttt{struct} or the class-key \texttt{class}.

A **standard-layout union** is a standard-layout class defined with the class-key \texttt{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:

```cpp
struct N { // neither trivial nor standard-layout
  int i;
  int j;
  virtual ~N();
};

struct T { // trivial but not standard-layout
  int i;
  private:
    int j;
};

struct SL { // standard-layout but not trivial
  int i;
  int j;
  ~SL();
};

struct POD { // both trivial and standard-layout
  int i;
  int j;
};
```

\(^{107}\) 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 \((5.10)\).
If a class-head contains a nested-name-specifier, the class-specifier shall refer to a class that was previously declared directly in the class or namespace to which the nested-name-specifier refers, or in an element of the inline namespace set (7.3.1) of that namespace (i.e., not merely inherited or introduced by a using-declaration), and the class-specifier shall appear in a namespace enclosing the previous declaration. In such cases, the nested-name-specifier of the class-head of the definition shall not begin with a decltype-specifier.

9.1 Class names

A class definition introduces a new type. [Example:

```c
struct X { int a; };
struct Y { int a; };
X a1;
Y a2;
int a3;
```
declares three variables of three different types. This implies that

```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]

A class declaration introduces the class name into the scope where it is declared and hides any class, variable, function, or other declaration of that name in an enclosing scope (3.3). If a class name is declared in a scope where a variable, function, or enumerator of the same name is also declared, then when both declarations are in scope, the class can be referred to only using an elaborated-type-specifier (3.4.4). [Example:

```c
struct stat {
   // ...
};

stat gstat; // use plain stat to
           // define variable

int stat(struct stat*); // redeclare stat as function

void f() {
   struct stat* ps; // struct prefix needed
                    // to name struct stat
   stat(ps); // call stat()
}
```
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 block-scope declaration
    s* p;
    // refer to local struct s
    struct s { char* p; }; // define local struct s
    struct s; // redeclaration, has no effect
}
```
— end example]

[Note: Such declarations allow definition of classes that refer to each other. [Example:

```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] [Example:

```c
struct s { int a; };

void g(int s) {
    struct s* p = new struct s; // global s
    p->a = s; // parameter s
}
```
— end example]

4 [Note: The declaration of a class name takes effect immediately after the identifier is seen in the class definition or elaborated-type-specifier. For example,

```c
class A * A;
```

first specifies A to be the name of a class and then redefines it as the name of a pointer to an object of that class. This means that the elaborated form class A must be used to refer to the class. Such artistry with names can be confusing and is best avoided. — end note]

5 A typedef-name (7.1.3) that names a class type, or a cv-qualified version thereof, is also a class-name. If a typedef-name that names a cv-qualified class type is used where a class-name is required, the cv-qualifiers
are ignored. A typedef-name shall not be used as the identifier in a class-head.

9.2 Class members

\[\text{class.mem}\]

\begin{verbatim}
member-specification:
  member-declaration member-specification_opt
  access-specifier : member-specification_opt

member-declaration:
  attribute-specifier_opt decl-specifier-seq_opt
  member-declarator-list_opt ;
  function-definition ; opt
  :: opt nested-name-specifier template_opt unqualified-id ;
  using-declaration
  static_assert-declaration
  template-declaration

member-declarator-list:
  member-declarator
  member-declarator-list , member-declarator

member-declarator:
  declarator pure-specifier_opt
  declarator brace-or-equal-initializer_opt
  identifier_opt attribute-specifier_opt : constant-expression

pure-specifier:
  = 0
\end{verbatim}

1 The member-specification in a class definition declares the full set of members of the class; no member can be added elsewhere. Members of a class are data members, member functions (9.3), nested types, and enumerators. Data members and member functions are static or non-static; see 9.4. Nested types are classes (9.1, 9.7) and enumerations (7.2) defined in the class, and arbitrary types declared as members by use of a typedef declaration (7.1.3). The enumerators of an unscoped enumeration (7.2) defined in the class are members of the class. Except when used to declare friends (11.4) 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, and except that an enumeration can be introduced with an opaque-enum-declaration and later redeclared with an enum-specifier.

A class is considered a completely-defined object type (3.9) (or complete type) at the closing } of the class-specifier. Within the class member-specification, the class is regarded as complete within function bodies, default arguments, exception-specifications, and brace-or-equal-initializers for non-static data members (including such things in nested classes). Otherwise it is regarded as incomplete within its own class member-specification.

2 [Note: a single name can denote several function members provided their types are sufficiently different (Clause 13). — end note]

3 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]

4 A member can be initialized using a brace-or-equal-initializer. (For static data members, see 9.4.2; for non-static data members, see 12.6.2).

5 A member shall not be declared with the extern or register storage-class-specifier. Within a class definition, a member shall not be declared with the thread_local storage-class-specifier unless also declared static.

§ 9.2
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).

The optional `attribute-specifier` in a `member-declaration` appertains to each of the entities declared by the `member-declarators`; it shall not appear if the optional `member-declarator-list` is omitted.

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

```c
struct tnode {
  char tword[20];
  int count;
  tnode *left;
  tnode *right;
};
```

which contains an array of twenty characters, an integer, and two pointers to objects of the same type. Once this definition has been given, the declaration

```c
tnode s, *sp;
```

declares s to be a `tnode` and sp to be a pointer to a `tnode`. With these declarations, `sp->count` refers to the `count` member of the object to which sp points; `s.left` refers to the `left` subtree pointer of the object s; and `s.right->tword[0]` refers to the initial character of the `tword` member of the `right` subtree of s. — end example]

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.

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 either neither member is a bit-field or both are bit-fields with the same width 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. \textit{[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 \hfill [class.mfct]

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 \textit{static} member function of its class (9.4); otherwise it is a \textit{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 \textit{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 \textit{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 \textit{inline}. \textit{[Note: member functions of a class in namespace scope have external linkage. Member functions of a local class (9.8) have no linkage. See 3.5. — end note]}

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 \textit{inline} member function definition in a program. See 3.2 and 7.1.2.

5 If the definition of a member function is lexically outside its class definition, the member function name shall be qualified by its class name using the :: operator. \textit{[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:

\begin{verbatim}
struct X {
    typedef int T;
    static T count;
    void f(T);
};
void X::f(T t = count) {} 
\end{verbatim}

The member function f of class X is defined in global scope; the notation X::f specifies that the function f is a member of class X and in the scope of class X. In the function definition, the parameter type T refers to
the typedef member \( T \) declared in class \( X \) and the default argument \( \text{count} \) refers to the static data member \( \text{count} \) declared in class \( X \). — end example]

6 A **static** local variable in a member function always refers to the same object, whether or not the member function is **inline**.

7 Member functions may be mentioned in **friend** declarations after their class has been defined.

8 Member functions of a local class shall be defined inline in their class definition, if they are defined at all.

9 [ **Note:** a member function can be declared (but not defined) using a typedef for a function type. The resulting member function has exactly the same type as it would have if the function declarator were provided explicitly, see 8.3.5. For example,

\[
\begin{align*}
\text{typedef void \(fv\)(void);} \\
\text{typedef void \(fvc\)(void) const;} \\
\text{struct \(S\) \{} \\
\text{\hspace{1em} \(fv\ \text{memfunc1}; \quad \text{// equivalent to: void memfunc1(void);} \)}} \\
\text{\hspace{1em} \text{void memfunc2();} } \\
\text{\hspace{1em} \(fv\ \text{memfunc3}; \quad \text{// equivalent to: void memfunc3(void) const;} \})} \\
\text{\}}; \\
\text{fv \(S::*\ pmfv1 = \&S::\text{memfunc1;}\)} \\
\text{fv \(S::*\ pmfv2 = \&S::\text{memfunc2;}\)} \\
\text{fvc \(S::*\ pmfv3 = \&S::\text{memfunc3;}\)} \\
\end{align*}
\]

Also see 14.3. — end note]

9.3.1 Nonstatic member functions [class.mfct.non-static]

1 A **non-static** member function may be called for an object of its class type, or for an object of a class derived (Clause 10) from its class type, using the class member access syntax (5.2.5, 13.3.1.1). A non-static member function may also be called directly using the function call syntax (5.2.2, 13.3.1.1) from within the body of a member function of its class or of a class derived from its class.

2 If a non-static member function of a class \( X \) is called for an object that is not of type \( X \), or of a type derived from \( X \), the behavior is undefined.

3 When an id-expression (5.1) that is not part of a class member access syntax (5.2.5) and not used to form a pointer to member (5.3.1) is used in the body of a non-static member function of class \( X \), if name lookup (3.4) resolves the name in the id-expression to a non-static non-type member of some class \( C \), the id-expression is transformed into a class member access expression (5.2.5) using (**this**) (9.3.2) as the postfix-expression to the left of the . operator. [ **Note:** if \( C \) is not \( X \) or a base class of \( X \), the class member access expression is ill-formed. — end note] Similarly during name lookup, when an unqualified-id (5.1) used in the definition of a member function for class \( X \) resolves to a **static** member, an enumerator or a nested type of class \( X \) or of a base class of \( X \), the unqualified-id is transformed into a qualified-id (5.1) in which the nested-name-specifier names the class of the member function. [ **Example:**

\[
\begin{align*}
\text{struct tnod\{} \\
\text{\hspace{1em} char tword[20];} \\
\text{\hspace{1em} int count;} \\
\text{\hspace{1em} tnod \*\left;} \\
\text{\hspace{1em} \\left \*\right;} \\
\text{\hspace{1em} \\left \*\right \\left \*\right;} \\
\text{\hspace{1em} \\left \*\right \text{set(char*, tnod\* \l, tnod\* \r);} \\
\text{\}}; \\
\text{\}}; \\
\text{\text{void tnod::set(char* w, tnod\* \l, tnod\* \r) \{} \\
\text{\hspace{1em} count = strlen(w)+1;} \\
\text{\}}} \\
\end{align*}
\]

§ 9.3.1 217
if (sizeof(tword) <= count)
    perror("tnode string too long");
strcpy(tword, w);
left = l;
right = r;
}

void f(tnode n1, tnode n2) {
    n1.set("abc", &n2, 0);
    n2.set("def", 0, 0);
}

In the body of the member function `tnode::set`, the member names `tword`, `count`, `left`, and `right` refer to members of the object for which the function is called. Thus, in the call `n1.set("abc", &n2, 0)`, `tword` refers to `n1.tword`, and in the call `n2.set("def", 0, 0)`, it refers to `n2.tword`. The functions `strlen`, `perror`, and `strcpy` are not members of the class `tnode` and should be declared elsewhere. — end example

4 A non-static member function may be declared `const`, `volatile`, or `const volatile`. These `cv-qualifiers` affect the type of the `this` pointer (9.3.2). They also affect the function type (8.3.5) of the member function; a member function declared `const` is a `const` member function, a member function declared `volatile` is a `volatile` member function and a member function declared `const volatile` is a `const volatile` member function. [Example:

    struct X {
        void g() const;
        void h() const volatile;
    };

    X::g is a `const` member function and X::h is a `const volatile` member function. — end example]

5 A non-static member function may be declared with a `ref-qualifier` (8.3.5); see 13.3.1.

6 A non-static member function may be declared `virtual` (10.3) or `pure virtual` (10.4).

9.3.2 The this pointer

1 In the body of a non-static (9.3) member function, the keyword `this` is a prvalue expression whose value is the address of the object for which the function is called. The type of `this` in a member function of a class `X` is `X*`. If the member function is declared `const`, the type of `this` is `const X*`, if the member function is declared `volatile`, the type of `this` is `volatile X*`, and if the member function is declared `const volatile`, the type of `this` is `const volatile X*`.

2 In a `const` member function, the object for which the function is called is accessed through a `const` access path; therefore, a `const` member function shall not modify the object and its non-static data members. [Example:

    struct s {
        int a;
        int f() const;
        int g() { return a++; } // error
        int h() const { return a++; }
    };

    int s::f() const { return a; }

108) See, for example, `<cstring>` (21.7).
The `a++` in the body of `s::h` is ill-formed because it tries to modify (a part of) the object for which `s::h()` is called. This is not allowed in a `const` member function because `this` is a pointer to `const`; that is, `*this` has `const` type. — end example

3 Similarly, `volatile` semantics (7.1.6.1) apply in `volatile` member functions when accessing the object and its non-static data members.

4 A `cv-qualified` member function can be called on an object-expression (5.2.5) only if the object-expression is as `cv-qualified` or less `cv-qualified` than the member function. [Example:

```c
void k(s& x, const s& y) {
  x.f();
  x.g();
  y.f();
  y.g();  // error
}
```

The call `y.g()` is ill-formed because `y` is `const` and `s::g()` is a non-`const` member function, that is, `s::g()` is less-qualified than the object-expression `y`. — end example]

5 Constructors (12.1) and destructors (12.4) shall not be declared `const`, `volatile` or `const volatile`. [Note: However, these functions can be invoked to create and destroy objects with `cv-qualified` types, see (12.1) and (12.4). — end note]

### 9.4 Static members

1 A data or function member of a class may be declared `static` in a class definition, in which case it is a `static member` of the class.

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();
```
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]

Static members obey the usual class member access rules (Clause 11). When used in the declaration of a class member, the static specifier shall only be used in the member declarations that appear within the member-specification of the class definition. [Note: it cannot be specified in member declarations that appear in namespace scope. — end note]

### 9.4.1 Static member functions

[Note: the rules described in 9.3 apply to static member functions. — end note]

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

A static data member is not part of the subobjects of a class. If a static data member is declared thread_local there is one copy of the member per thread. If a static data member is not declared thread_local there is one copy of the data member that is shared by all the objects of the class.

The declaration of a static data member in its class definition is not a definition and may be of an incomplete type other than cv-qualified void. The definition for a static data member shall appear in a namespace scope enclosing the member’s class definition. In the definition at namespace scope, the name of the static data member shall be qualified by its class name using the :: operator. The initializer expression in the definition of a static data member is in the scope of its class (3.3.7). [Example:

```cpp
class process {
    static process* run_chain;
    static process* running;
};
```

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 brace-or-equal-initializer in which every initializer-clause that is an assignment-expression is a constant expression. A static data member of literal type can be declared in the class definition with the constexpr specifier; if so, its declaration shall specify a brace-or-equal-initializer in which every initializer-clause that is an
assignment-expression is a constant expression. [Note: In both these cases, the member may appear in constant expressions. — end note] The member shall still be defined in a namespace scope if it is 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 variables (3.6.2, 3.6.3).

A static data member shall not be mutable (7.1.1).

9.5 Unions

In a union, at most one of the non-static data members can be active at any time, that is, the value of at most one of the non-static data members can be stored in a union at any time. [Note: one special guarantee is made in order to simplify the use of unions: If a standard-layout union contains several standard-layout structs that share a common initial sequence (9.2), and if an object of this standard-layout union type contains one of the standard-layout structs, it is permitted to inspect the common initial sequence of any of standard-layout struct members; see 9.2. — end note] The size of a union is sufficient to contain the largest of its non-static data members. Each non-static data member is allocated as if it were the sole member of a struct.

A union can have member functions (including constructors and destructors), but not virtual (10.3) functions. A union shall not have base classes. A union shall not be used as a base class. If a union contains a non-static data member of reference type the program is ill-formed. At most one non-static data member of a union may have a brace-or-equal-initializer. [Note: if any non-static data member of a union has a non-trivial default constructor (12.1), copy constructor (12.8), move constructor (12.8), copy assignment operator (12.8), move assignment operator (12.8), or destructor (12.4), the corresponding member function of the union must be user-provided or it will be implicitly deleted (8.4.3) for the union. — end note]

Example: Consider the following union:

```cpp
union U {
    int i;
    float f;
    std::string s;
};
```

Since std::string (21.3) declares non-trivial versions of all of the special member functions, U will have an implicitly deleted default constructor, copy/move constructor, copy/move assignment operator, and destructor. To use U, some or all of these member functions must be user-provided. — end example]

Note: In general, one must use explicit destructor calls and placement new operators to change the active member of a union. — end note] [Example: Consider an object u of a union type U having non-static data members m of type M and n of type N. If M has a non-trivial destructor and N has a non-trivial constructor (for instance, if they declare or inherit virtual functions), the active member of u can be safely switched from m to n using the destructor and placement new operator as follows:

```cpp
u.m~M();
new (&u.n) N;
```

— end example]

§ 9.5
A union of the form

```c
union { member-specification } ;
```

is called an anonymous union; it defines an unnamed object of unnamed type. The `member-specification` of an anonymous union shall only define non-static data members. [Note: nested types and functions cannot be declared within an anonymous union. — end note] The names of the members of an anonymous union shall be distinct from the names of any other entity in the scope in which the anonymous union is declared. For the purpose of name lookup, after the anonymous union definition, the members of the anonymous union are considered to have been defined in the scope in which the anonymous union is declared. [Example:

```c
void f() {
    union { int a; char* p; ;
    a = 1;
    p = "Jennifer";
}
```

Here `a` and `p` are used like ordinary (nonmember) variables, but since they are union members they have the same address. — end example]

Anonymous unions declared in a named namespace or in the global namespace shall be declared `static`. Anonymous unions declared at block scope shall be declared with any storage class allowed for a block-scope variable, or with no storage class. A storage class is not allowed in a declaration of an anonymous union in a class scope. An anonymous union shall not have `private` or `protected` members (Clause 11). An anonymous union shall not have function members.

A union for which objects or pointers are declared is not an anonymous union. [Example:

```c
union { int aa; char* p; } obj, *ptr = &obj;
aa = 1; // error
ptr->aa = 1; // OK
```

The assignment to plain `aa` is ill-formed since the member name is not visible outside the union, and even if it were visible, it is not associated with any particular object. — end example] [Note: Initialization of unions with no user-declared constructors is described in (8.5.1). — end note]

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

A `member-declarator` of the form

```c
identifier_opt attribute-specifier_opt : constant-expression
```

specifies a bit-field; its length is set off from the bit-field name by a colon. The optional `attribute-specifier` appertains to the entity being declared. The bit-field attribute is not part of the type of the class member. The `constant-expression` shall be an integral constant expression with a value greater than or equal to zero. The value of the integral constant expression may be larger than the number of bits in the object representation (3.9) of the bit-field’s type; in such cases the extra bits are used as padding bits and do not participate in the value representation (3.9) of the bit-field. Allocation of bit-fields within a class object is implementation-defined. Alignment of bit-fields is implementation-defined. Bit-fields are packed into some addressable allocation unit. [Note: bit-fields straddle allocation units on some machines and not on others. Bit-fields are assigned right-to-left on some machines, left-to-right on others. — end note]
A declaration for a bit-field that omits the identifier declares an unnamed bit-field. Unnamed bit-fields are not members and cannot be initialized. [Note: an unnamed bit-field is useful for padding to conform to externally-imposed layouts. — end note] As a special case, an unnamed bit-field with a width of zero specifies alignment of the next bit-field at an allocation unit boundary. Only when declaring an unnamed bit-field may the value of the constant-expression be equal to zero.

A bit-field shall not be a static member. A bit-field shall have integral or enumeration type (3.9.1). It is implementation-defined whether a plain (neither explicitly signed nor unsigned) char, short, int or long bit-field is signed or unsigned. A bool value can successfully be stored in a bit-field of any nonzero size. The address-of operator & shall not be applied to a bit-field, so there are no pointers to bit-fields. A non-const reference shall not be bound to a bit-field (8.5.3). [Note: if the initializer for a reference of type const T& is an lvalue that refers to a bit-field, the reference is bound to a temporary initialized to hold the value of the bit-field; the reference is not bound to the bit-field directly. See 8.5.3. — end note]

If the value true or false is stored into a bit-field of type bool of any size (including a one bit bit-field), the original bool value and the value of the bit-field shall compare equal. If the value of an enumerator is stored into a bit-field of the same enumeration type and the number of bits in the bit-field is large enough to hold all the values of that enumeration type (7.2), the original enumerator value and the value of the bit-field shall compare equal. [Example:

```c
enum BOOL { FALSE=0, TRUE=1 };  
struct A {  
    BOOL b:1;  
};  
A a;  
void f() {
    a.b = TRUE;  
    if (a.b == TRUE) // yields true
        {/* ... */}
}
```

— end example]

9.7 Nested class declarations

A class can be declared within another class. A class declared within another is called a nested class. The name of a nested class is local to its enclosing class. The nested class is in the scope of its enclosing class. [Note: see 5.1 for restrictions on the use of non-static data members and non-static member functions. — end note]

[Example:

```c
int x;  
int y;

struct enclose {
    int x;
    static int s;

    struct inner {
        void f(int i) {
            int a = sizeof(x); // OK: operand of sizeof is an unevaluated operand
            x = i; // error: assign to enclose::x
            s = i; // OK: assign to enclose::s
            ::x = i; // OK: assign to global x
            y = i; // OK: assign to global y
        }
    }
};
```

§ 9.7
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:

```cpp
class enclose {
    class inner {
        static int x;
        void f(int i);
    };
    int enclose::inner::x = 1;
}
void enclose::inner::f(int i) {
    // ... }
```

— end example]

3 If class X is defined in a namespace scope, a nested class Y may be declared in class X and later defined in the definition of class X or be later defined in a namespace scope enclosing the definition of class X. [Example:

```cpp
class E {
    class I1; // forward declaration of nested class
    class I2;
    class I1 { };
    class E::I2 { };
}
```

— end example]

4 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.

9.8 Local class declarations [class.local]

1 A class can be declared within a function definition; such a class is called a local class. The name of a local class is local to its enclosing scope. The local class is in the scope of the enclosing scope, and has the same access to names outside the function as does the enclosing function. Declarations in a local class 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 has automatic storage duration
    int h() { return s; }  // OK
    int k() { return ::x; } // OK
    int l() { return g(); } // OK
};

local* p = 0;                  // error: local not in scope

— end example]

2 An enclosing function has no special access to members of the local class; it obeys the usual access rules (Clause 11). Member functions of a local class shall be defined within their class definition, if they are defined at all.

3 If class X is a local class a nested class Y may be declared in class X and later defined in the definition of class X or be later defined in the same scope as the definition of class X. A class nested within a local class is a local class.

4 A local class shall not have static data members.

9.9 Nested type names [class.nested.type]

1 Type names obey exactly the same scope rules as other names. In particular, type names defined within a class definition cannot be used outside their class without qualification. [Example:

```c
struct X {
    typedef int I;
    class Y { /* ... */ }; // error: local not in scope
    I a;
};

I b;                      // error
Y c;                      // error
X::Y d;                   // OK
X::I e;                   // OK

— end example]
```
10 Derived classes

A list of base classes can be specified in a class definition using the notation:

```
base-clause:
  : base-specifier-list
base-specifier-list:
  base-specifier ... opt
  base-specifier-list , base-specifier ... opt
base-specifier:
  attribute-specifier_opt base-type-specifier
  attribute-specifier_opt virtual access-specifier_opt base-type-specifier
  attribute-specifier_opt access-specifier virtual_opt base-type-specifier
class-or-decltype:
  :: opt nested-name-specifier_opt class-name
  decltype-specifier
base-type-specifier:
  class-or-decltype
access-specifier:
  private
  protected
  public
```

The optional attribute-specifier appertains to the base-specifier.

The type denoted by a base-type-specifier shall be a class type that is not an incompletely defined class (Clause 9); this class is called a direct base class for the class being defined. During the lookup for a base class name, non-type names are ignored (3.3.10). If the name found is not a class-name, the program is ill-formed. A class B is a base class of a class D if it is a direct base class of D or a direct base class of one of D’s base classes. A class is an indirect base class of another if it is a base class but not a direct base class. A class is said to be (directly or indirectly) derived from its (direct or indirect) base classes. [Note: see Clause 11 for the meaning of access-specifier. — end note] Unless redeclared in the derived class, members of a base class are also considered to be members of the derived class. The base class members are said to be inherited by the derived class. Inherited members can be referred to in expressions in the same manner as other members of the derived class, unless their names are hidden or ambiguous (10.2). [Note: the scope resolution operator :: (5.1) can be used to refer to a direct or indirect base member explicitly. This allows access to a name that has been redeclared in the derived class. A derived class can itself serve as a base class subject to access control; see 11.2. A pointer to a derived class can be implicitly converted to a pointer to an accessible unambiguous base class (4.10). An lvalue of a derived class type can be bound to a reference to an accessible unambiguous base class (8.5.3). — end note]

The base-specifier-list specifies the type of the base class subobjects contained in an object of the derived class type. [Example:

```c
struct Base {
  int a, b, c;
};

struct Derived : Base {
  int b;
};
```
Here, an object of class Derived2 will have a subobject of class Derived which in turn will have a subobject of class Base. — end example]

4 A base-specifier followed by an ellipsis is a pack expansion (14.5.3).

5 The order in which the base class subobjects are allocated in the most derived object (1.8) is unspecified. [Note: a derived class and its base class subobjects can be represented by a directed acyclic graph (DAG) where an arrow means "directly derived from." A DAG of subobjects is often referred to as a "subobject lattice."

\[
\begin{align*}
&\text{Base} \\
&\quad \downarrow \\
&\text{Derived1} \\
&\quad \downarrow \\
&\text{Derived2}
\end{align*}
\]

Figure 2 — Directed acyclic graph

6 The arrows need not have a physical representation in memory. — end note]

7 [Note: initialization of objects representing base classes can be specified in constructors; see 12.6.2. — end note]

8 [Note: A base class subobject might have a layout (3.7) different from the layout of a most derived object of the same type. A base class subobject might have a polymorphic behavior (12.7) different from the polymorphic behavior of a most derived object of the same type. A base class subobject may be of zero size (Clause 9); however, two subobjects that have the same class type and that belong to the same most derived object must not be allocated at the same address (5.10). — end note]

### 10.1 Multiple base classes

1 A class can be derived from any number of base classes. [Note: the use of more than one direct base class is often called multiple inheritance. — end note] [Example:

```cpp

class A {
  /* ... */;
};
class B {
  /* ... */;
};
class C {
  /* ... */;
};
class D : public A, public B, public C {
  /* ... */;
}

— end example]

2 [Note: the order of derivation is not significant except as specified by the semantics of initialization by constructor (12.6.2), cleanup (12.4), and storage layout (9.2, 11.1). — end note]

3 A class shall not be specified as a direct base class of a derived class more than once. [Note: a class can be an indirect base class more than once and can be a direct and an indirect base class. There are limited
things that can be done with such a class. The non-static data members and member functions of the direct base class cannot be referred to in the scope of the derived class. However, the static members, enumerations and types can be unambiguously referred to. — end note] [Example:

```cpp
class X { /* ... */ };  // well-formed
class Y : public X, public X { /* ... */ };  // ill-formed

class L { public: int next; /* ... */ };  // ill-formed
class A : public L { /* ... */ };  // ill-formed
class B : public L { /* ... */ };  // well-formed
class C : public A, public B { void f(); /* ... */ };  // well-formed
```

— end example]

4 A base class specifier that does not contain the keyword `virtual`, specifies a non-virtual base class. A base class specifier that contains the keyword `virtual`, specifies a virtual base class. For each distinct occurrence of a non-virtual base class in the class lattice of the most derived class, the most derived object (1.8) shall contain a corresponding distinct base class subobject of that type. For each distinct base class that is specified virtual, the most derived object shall contain a single base class subobject of that type. [Example: for an object of class type `C`, each distinct occurrence of a (non-virtual) base class `L` in the class lattice of `C` corresponds one-to-one with a distinct `L` subobject within the object of type `C`. Given the class `C` defined above, an object of class `C` will have two subobjects of class `L` as shown below.

![Diagram](image-url)

Figure 3 — Non-virtual base

5 In such lattices, explicit qualification can be used to specify which subobject is meant. The body of function `C::f` could refer to the member `next` of each `L` subobject:

```cpp
void C::f() { A::next = B::next; }  // well-formed
```

Without the `A::` or `B::` qualifiers, the definition of `C::f` above would be ill-formed because of ambiguity (10.2).

6 For another example,

```cpp
class V { /* ... */ };  // ill-formed
class A : virtual public V { /* ... */ };  // ill-formed
class B : virtual public V { /* ... */ };  // ill-formed
class C : public A, public B { /* ... */ };  // ill-formed
```

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.
Figure 4 — Virtual base

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.

Figure 5 — Virtual and non-virtual base

— 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 (3.3.7). Name lookup can result in an ambiguity, in which case the program is ill-formed. For an id-expression, name lookup begins in the class scope of this; for a qualified-id, name lookup begins in the scope of the nested-name-specifier. Name lookup takes place before access control (3.4, Clause 11).

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, variable, and enumerator declarations. As another example, looking up a name in a using-declaration (7.3.3) includes the declaration of a class or enumeration that would ordinarily be hidden by another declaration of that name in the same scope. — end note] If the resulting declaration set is not empty, the subobject set contains C itself, and calculation is complete.

5 Otherwise (i.e., C does not contain a declaration of f or the resulting declaration set is empty), S(f, C) is initially empty. If C has base classes, calculate the lookup set for f in each direct base class subobject B_i, and merge each such lookup set S(f, B_i) in turn into S(f, C).

6 The following steps define the result of merging lookup set S(f, B_i) into the intermediate S(f, C):

— 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.

7 The result of name lookup for f in C is the declaration set of S(f, C). If it is an invalid set, the program is ill-formed. [Example:

```c
struct A { int x; }; // S(x,A) = { { A::x }, { A } }
struct B { float x; }; // S(x,B) = { { B::x }, { B } }
struct C: public A, public B { }; // S(x,C) = { invalid, { A in C, B in C } }
struct D: public virtual C { }; // S(x,D) = S(x,C)
struct E: public virtual C { char x; }; // S(x,E) = { { E::x }, { E } }
struct F: public D, public E { }; // S(x,F) = S(x,E)
int main() {
    F f;
    f.x = 0; // OK, lookup finds E::x
    return 0;
}
```

S(x, F) is unambiguous because the A and B base subobjects of D are also base subobjects of E, so S(x, D) is discarded in the first merge step. — end example]

8 If the name of an overloaded function is unambiguously found, overloading resolution (13.3) also takes place before access control. Ambiguities can often be resolved by qualifying a name with its class name. [Example:

```c
struct A {
    int f();
};

struct B {
    int f();
};

struct C : A, B {
    int f() { return A::f() + B::f(); }
};
```

§ 10.2 230
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 as 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(); }
```

![Figure 6 — Name lookup](image)

— end example]

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]
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 expression designating an object of a derived class
to a pointer or reference to one of its base classes shall unambiguously refer to a unique object representing
the base class. [Example:

```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;
    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, 5.3.1, 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)
        f(0.0);           // Unambiguous (only one B2)
        int B1::* mpB1 = &D::i;  // Unambiguous
        int D::* mpD = &D::i;   // Ambiguous conversion
    }
}

— end example
```

§ 10.2
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 \texttt{vf} is declared in a class \texttt{Base} and in a class \texttt{Derived}, derived directly or indirectly from \texttt{Base}, a member function \texttt{vf} with the same name, parameter-type-list (8.3.5), cv-qualification, and ref-qualifier (or absence of same) as \texttt{Base::vf} is declared, then \texttt{Derived::vf} is also virtual (whether or not it is so declared) and it overrides \texttt{Base::vf}. For convenience we say that any virtual function overrides itself. A virtual member function \texttt{C::vf} of a class object \texttt{S} is a final overrider unless the most derived class (1.8) of which \texttt{S} is a base class subobject (if any) declares or inherits another member function that overrides \texttt{vf}. In a derived class, if a virtual member function of a base class subobject has more than one final overrider the program is ill-formed. [Example:

```cpp
struct A {
  virtual void f();
};
struct B : virtual A {
  virtual void f();
};
struct C : B, virtual A {
  using A::f;
};

void foo() {
  C c;
  c.f(); // calls B::f, the final overrider
  c.C::f(); // calls A::f because of the using-declaration
}
```

— end example]

[Example:

```cpp
struct A { virtual void f(); };
struct B : A { };  
struct C : A { void f(); };
struct D : B, C { }; // OK: A::f and C::f are the final overriders  // for the B and C subobjects, respectively
```

— end 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);
};
struct D2 : D {
  void f();
};
```

\(109\) A function with the same name but a different parameter list (Clause 13) as a virtual function is not necessarily virtual and does not override. The use of the \texttt{virtual} specifier in the declaration of an overriding function is legal but redundant (has empty semantics). Access control (Clause 11) is not considered in determining overriding.
the function \( f(int) \) in class \( D \) hides the virtual function \( f() \) in its base class \( B; D::f(int) \) is not a virtual function. However, \( f() \) declared in class \( D2 \) has the same name and the same parameter list as \( B::f() \), and therefore is a virtual function that overrides the function \( B::f() \) even though \( B::f() \) is not visible in class \( D2 \). — end note]

4 Even though destructors are not inherited, a destructor in a derived class overrides a base class destructor declared virtual; see 12.4 and 12.5.

5 The return type of an overriding function shall be either identical to the return type of the overridden function or covariant with the classes of the functions. If a function \( D::f \) overrides a function \( B::f \), the return types of the functions are covariant if they satisfy the following criteria:

- both are pointers to classes, both are lvalue references to classes, or both are rvalue references to classes\(^{110}\)
- the class in the return type of \( B::f \) is the same class as the class in the return type of \( D::f \), or is an unambiguous and accessible direct or indirect base class of the class in the return type of \( D::f \)
- both pointers or references have the same cv-qualification and the class type in the return type of \( D::f \) has the same cv-qualification as or less cv-qualification than the class type in the return type of \( B::f \).

6 If the return type of \( D::f \) differs from the return type of \( B::f \), the class type in the return type of \( D::f \) shall be complete at the point of declaration of \( D::f \) or shall be the class type \( D \). When the overriding function is called as the final overrider of the overridden function, its result is converted to the type returned by the (statically chosen) overridden function (5.2.2). [Example:

```c
class B {
};
class D : private B { friend class Derived; };
struct Base {
    virtual void vf1();
    virtual void vf2();
    virtual void vf3();
    virtual B* vf4();
    virtual B* vf5();
    void f();
};

struct No_good : public Base {
    D* vf4(); // error: B (base class of D) inaccessible
};

class A;
struct Derived : public Base {
    void vf1(); // virtual and overrides Base::vf1()
    void vf2(int); // not virtual, hides Base::vf2()
    char vf3(); // error: invalid difference in return type only
    D* vf4(); // OK: returns pointer to derived class
    A* vf5(); // error: returns pointer to incomplete class
    void f();
};

void g() {
    Derived d;
    Base* bp = &d; // standard conversion:
    // Derived* to Base*
```
bp->vf1(); // calls Derived::vf1()
bp->vf2(); // calls Base::vf2()
bp->f(); // calls Base::f() (not virtual)
B* p = bp->vf4(); // calls Derived::pf() and converts the
            // result to B*
Derived* dp = &d;
D* q = dp->vf4(); // calls Derived::pf() and does not
            // convert the result to B*
dp->vf2(); // ill-formed: argument mismatch
}

— end example]

7 [Note: the interpretation of the call of a virtual function depends on the type of the object for which it is called (the dynamic type), whereas the interpretation of a call of a non-virtual member function depends only on the type of the pointer or reference denoting that object (the static type) (5.2.2). — end note]

8 [Note: the virtual specifier implies membership, so a virtual function cannot be a nonmember (7.1.2) function. Nor can a virtual function be a static member, since a virtual function call relies on a specific object for determining which function to invoke. A virtual function declared in one class can be declared a friend in another class. — end note]

9 A virtual function declared in a class shall be defined, or declared pure (10.4) in that class, or both; but no diagnostic is required (3.2).

10 [Example: here are some uses of virtual functions with multiple base classes:

struct A {
    virtual void f();
};

struct B1 : A { // note non-virtual derivation
    void f();
};

struct B2 : A {
    void f();
};

struct D : B1, B2 { // D has two separate A subobjects
};

void foo() {
    D d;
    // A* ap = &d; // would be ill-formed: ambiguous
    B1* blp = &d;
    A* ap = blp;
    D* dp = &d;
    ap->f(); // calls D::B1::f
    dp->f(); // ill-formed: ambiguous
}

In class D above there are two occurrences of class A and hence two occurrences of the virtual member function A::f. The final overrider of B1::A::f is B1::f and the final overrider of B2::A::f is B2::f.

11 The following example shows a function that does not have a unique final overrider:

§ 10.3
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.

12 The following example uses the well-formed classes from above.

struct VB1a : virtual A {    // does not declare f
};

struct Da : VB1a, VB2 {
};

void foe() {
    VB1a* vb1ap = new Da;
    vb1ap->f();        // calls VB2::f
}

— end example]

13 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]

14 A function with a deleted definition (8.4) shall not override a function that does not have a deleted definition. Likewise, a function that does not have a deleted definition shall not override a function with a deleted definition.

10.4 Abstract classes  [class.abstract]

1 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 {
   // abstract class
   point center;
   public:
      point where() { return center; }
      void move(point p) { center = p; draw(); }
      virtual void rotate(int) = 0; // pure virtual
      virtual void draw() = 0; // pure virtual
};
```

— end example] [Note: a function declaration cannot provide both a pure-specifier and a definition — end note] [Example:

```cpp
struct C {
   virtual void f() = 0;
};
```

— 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);
shape& h(shape&); // error
```

— end example]

4 A class is abstract if it contains or inherits at least one pure virtual function for which the final overrider is pure virtual. [Example:

```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]

5  [Note: an abstract class can be derived from a class that is not abstract, and a pure virtual function may override a virtual function which is not pure. — end note]

6  Member functions can be called from a constructor (or destructor) of an abstract class; the effect of making a virtual call (10.3) to a pure virtual function directly or indirectly for the object being created (or destroyed) from such a constructor (or destructor) is undefined.
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.\(^{111}\)

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 { };       
      public:
      typedef B BB;
   };
   
   void f() {
      A::BB x;          // OK, typedef name A::BB is public
      A::B y;           // access error, A::B is private
   }
   ```
   — end note]

5 It should be noted that it is **access** to members and base classes that is controlled, not their **visibility**. Names of members are still visible, and implicit conversions to base classes are still considered, when those members and base classes are inaccessible. The interpretation of a given construct is established without regard to

\(^{111}\) Access permissions are thus transitive and cumulative to nested and local classes.
access control. If the interpretation established makes use of inaccessible member names or base classes, the
construct is ill-formed.

6 All access controls in Clause 11 affect the ability to access a class member name from 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:

```c
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]

7 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.

8 The names in a default template-argument (14.1) have their access checked in the context in which they
appear rather than at any points of use of the default template-argument. [Example:

```c
class B { };
template <class T> class C {
  protected:
    typedef T TT;
};

template <class U, class V = typename U::TT>
class D : public U { };

D <C<B> >* d;      // access error, C::TT is protected

— end example]

240
11.1 Access specifiers

1 Member declarations can be labeled by an access-specifier (Clause 10):

\[\text{access-specifier : member-specification}_{\text{opt}}\]

An access-specifier specifies the access rules for members following it until the end of the class or until another access-specifier is encountered. [Example:

```cpp
class X {
  int a;  // X::a is private by default: class used
public:
  int b;  // X::b is public
  int c;  // X::c is public
};
```

— end example]

2 Any number of access specifiers is allowed and no particular order is required. [Example:

```cpp
struct S {
  int a;  // S::a is public by default: struct used
protected:
  int b;  // S::b is protected
private:
  int c;  // S::c is private
public:
  int d;  // S::d is public
};
```

— end example]

3 [Note: the effect of access control on the order of allocation of data members is described in 9.2. — end note]

4 When a member is redeclared within its class definition, the access specified at its redeclaration shall be the same as at its initial declaration. [Example:

```cpp
struct S {
  class A;
  enum E : int;
private:
  class A { };  // error: cannot change access
  enum E: int { e0 };  // error: cannot change access
};
```

— end example]

5 [Note: In a derived class, the lookup of a base class name will find the injected-class-name instead of the name of the base class in the scope in which it was declared. The injected-class-name might be less accessible than the name of the base class in the scope in which it was declared. — end note]

[Example:

```cpp
class A { };  
class B : private A { };  
class C : public B {
  A *p;  // error: injected-class-name A is inaccessible
         ::A *q;  // OK
};
```
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.\(^\text{112}\).

In the absence of an access-specifier for a base class, public is assumed when the derived class is defined with the class-key struct and private is assumed when the class is defined with the class-key class. [Example:

```cpp
class B { /* ... */; 
class D1 : private B { /* ... */; 
class D2 : public B { /* ... */; 
class D3 : B { /* ... */; // B private by default 
struct D4 : public B { /* ... */; 
struct D5 : private B { /* ... */; 
struct D6 : B { /* ... */; // B public by default 
class D7 : protected B { /* ... */; 
struct D8 : protected B { /* ... */; 

Here B is a public base of D2, D4, and D6, a private base of D1, D3, and D5, and a protected base of D7 and D8. — end example]
```

[Note: A member of a private base class might be inaccessible as an inherited member name, but accessible directly. Because of the rules on pointer conversions (4.10) and explicit casts (5.4), a conversion from a pointer to a derived class to a pointer to an inaccessible base class might be ill-formed if an implicit conversion is used, but well-formed if an explicit cast is used. For example,

```cpp
class B {
    public:
        int mi; // non-static member
        static int si; // static member
    }

class D : private B {
}

class DD : public D {
    void f();
}

void DD::f() {
    mi = 3; // error: mi is private in D
    si = 3; // error: si is private in D
    ::B b;
    b.mi = 3; // OK (b.mi is different from this->mi)
    b.si = 3; // OK (b.si is different from this->si)
    ::B::si = 3; // OK
    ::B* bp1 = this; // error: B is a private base class
    ::B* bp2 = (::B*)this; // OK with cast
```

\(^{112}\) As specified previously in Clause 11, private members of a base class remain inaccessible even to derived classes unless friend declarations within the base class definition are used to grant access explicitly.

§ 11.2
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:

```cpp
class B {
  public:
    int m;
};

class S: private B {
  friend class N;
};

class N: private S {
  void f() {
    B* p = this;  // OK because class S satisfies the fourth condition
    // above: B is a base class of N accessible in f() because
    // B is an accessible base class of S and S is an accessible
    // base class of N.
  }
};

— end example ]

5 If a base class is accessible, one can implicitly convert a pointer to a derived class to a pointer to that base class (4.10, 4.11). [Note: it follows that members and friends of a class X can implicitly convert an X* to a pointer to a private or protected immediate base class of X. — end note] The access to a member is affected by the class in which the member was looked up and found. [Note: this class can be explicit, e.g., when a qualified-id is used, or implicit, e.g., when a class member access operator (5.2.5) is used (including cases where an implicit “this->” is added). If both a class member access operator and a qualified-id are used to name the member (as in p->T::m), the class naming the member is the class denoted by the nested-name-specifier of the qualified-id (that is, T). — end note] A member m is accessible at the point R when named in class N if
   — m as a member of N is public, or
   — m as a member of N is private, and R occurs in a member or friend of class N, or
   — m as a member of N is protected, and R occurs in a member or friend of class N, or in a member or friend of a class P derived from N, where m as a member of P is public, private, or protected, or

§ 11.2
there exists a base class \( B \) of \( N \) that is accessible at \( R \), and \( m \) is accessible at \( R \) when named in class \( B \).

**Example:**

```cpp
class B;
class A {
private:
    int i;
    friend void f(B*);
};
class B : public A {
    void f(B* p) {
        p->i = 1; // OK: B* can be implicitly converted to A*,
        // and f has access to i in A
    }
}
```

**end example**

If a class member access operator, including an implicit “\this->,” is used to access a non-static data member or non-static member function, the reference is ill-formed if the left operand (considered as a pointer in the “.” operator case) cannot be implicitly converted to a pointer to the naming class of the right operand.

**Note:** this requirement is in addition to the requirement that the member be accessible as named. — end note

### 11.3 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 ;`.\(^{113}\)

**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
```

\(^{113}\) Access declarations are deprecated; member *using-declarations* (7.3.3) provide a better means of doing the same things. In earlier versions of the C++ language, access declarations were more limited; they were generalized and made equivalent to *using-declarations* in the interest of simplicity. Programmers are encouraged to use *using-declarations*, rather than the new capabilities of access declarations, in new code.
The external function \texttt{ef} can use only the names \texttt{c}, \texttt{z}, \texttt{z1}, \texttt{e}, and \texttt{df}. Being a member of \texttt{D}, the function \texttt{df} can use the names \texttt{b}, \texttt{c}, \texttt{z}, \texttt{z1}, \texttt{bf}, \texttt{x}, \texttt{y}, \texttt{d}, \texttt{e}, \texttt{df}, and \texttt{g}, but not \texttt{a}. Being a member of \texttt{B}, the function \texttt{bf} can use the members \texttt{a}, \texttt{b}, \texttt{c}, \texttt{z}, \texttt{z1}, \texttt{bf}, \texttt{x}, and \texttt{y}. The function \texttt{xf} can use the public and protected names from \texttt{D}, that is, \texttt{c}, \texttt{z}, \texttt{z1}, \texttt{e}, and \texttt{df} (public), and \texttt{x}, and \texttt{g} (protected). Thus the external function \texttt{ff} has access only to \texttt{c}, \texttt{z}, \texttt{z1}, \texttt{e}, and \texttt{df}. If \texttt{D} were a protected or private base class of \texttt{X}, \texttt{xf} would have the same privileges as before, but \texttt{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 the class. A class specifies its friends, if any, by way of friend declarations. Such declarations give special access rights to the friends, but they do not make the nominated friends members of the befriending class. [Example: the following example illustrates the differences between members and friends:

```cpp
class X {
  int a;
  friend void friend_set(X*, int);
public:
  void member_set(int);
};

void friend_set(X* p, int i) { p->a = i; }
void X::member_set(int i) { a = i; }

void f() {
  X obj;
  friend_set(&obj,10);
  obj.member_set(10);
}

— end example]

Declaring a class to be a friend implies that the names of private and protected members from the class granting friendship can be accessed in the \textit{base-specifiers} and member declarations of the befriended class. [Example:

```cpp
class A {
  class B { }; // adjust access to A::z
};
```
struct X : A::B {  // OK: A::B accessible to friend
    A::B mx;  // OK: A::B accessible to member of friend
    class Y {  // OK: A::B accessible to nested member of friend
        A::B my;
    };
};

— end example] A class shall not be defined in a friend declaration. [Example:

    class X {
        enum { a=100 };  
        friend class Y;
    };

class Y {
    int v[X::a];  // OK, Y is a friend of X
};

class Z {
    int v[X::a];  // error: X::a is private
};

— end example]

§ 11.4 246

3 A friend declaration that does not declare a function shall have one of the following forms:

    friend elaborated-type-specifier ;
    friend simple-type-specifier ;
    friend typename-specifier ;

[Note: a friend declaration may be the declaration in a template-declaration (Clause 14, 14.5.4). — end note] If the type specifier in a friend declaration designates a (possibly cv-qualified) class type, that class is declared as a friend; otherwise, the friend declaration is ignored. [Example:

    class C;
    typedef C Ct;

class X1 {
    friend C;  // OK: class C is a friend
};

class X2 {
    friend Ct;  // OK: class C is a friend
    friend D;  // error: no type-name D in scope
    friend class D;  // OK: elaborated-type-specifier declares new class
};

template <typename T> class R {
    friend T;
};

    R<C> rc;  // class C is a friend of R<C>
    R<int> Ri;  // OK: "friend int;" is ignored

— 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:

```c
class Y {
    friend char* X::foo(int);
    friend X::X(char);  // constructors can be friends
    friend X::~X();     // destructors can be friends
};
```
— end example]

A function can be defined in a friend declaration of a class if and only if the class is a non-local class (9.8), the function name is unqualified, and the function has namespace scope. [Example:

```c
class M {
    friend void f() { }  // definition of global f, a friend of M,
                        // not the definition of a member function
};
```
— end example]

Such a function is implicitly inline. A friend function defined in a class is in the (lexical) scope of the class in which it is defined. A friend function defined outside the class is not (3.4.1).

No storage-class-specifier shall appear in the decl-specifier-seq of a friend declaration.

A name nominated by a friend declaration shall be accessible in the scope of the class containing the friend declaration. The meaning of the friend declaration is the same whether the friend declaration appears in the private, protected or public (9.2) portion of the class member-specification.

Friendship is neither inherited nor transitive. [Example:

```c
class A {
    friend class B;
    int a;
};

class B {
    friend class C;
};

class C {
    void f(A* p) {
        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
    }
};
```
11  If a friend declaration appears in a local class (9.8) and the name specified is an unqualified name, a prior declaration is looked up without considering scopes that are outside the innermost enclosing non-class scope. For a friend function declaration, if there is no prior declaration, the program is ill-formed. For a friend class declaration, if there is no prior declaration, the class that is specified belongs to the innermost enclosing non-class scope, but if it is subsequently referenced, its name is not found by name lookup until a matching declaration is provided in the innermost enclosing non-class scope. [Example:

```c
class X;
void a();
void f() {
  class Y;
  extern void b();
  class A {
    friend class X; // OK, but X is a local class, not ::X
    friend class Y; // OK
    friend class Z; // OK, introduces local class Z
    friend void a(); // error, ::a is not considered
    friend void b(); // OK
    friend void c(); // error
  };
  X *px; // OK, but ::X is found
  Z *pz; // error, no Z is found
}
```

— end example]

11.5 Protected member access [class.protected]

An additional access check beyond those described earlier in Clause 11 is applied when a non-static data member or non-static member function is a protected member of its naming class (11.2)\(^{14}\) As described earlier, access to a protected member is granted because the reference occurs in a friend or member of some class C. If the access is to form a pointer to member (5.3.1), the nested-name-specifier shall denote C or a class derived from C. All other accesses involve a (possibly implicit) object expression (5.2.5). In this case, the class of the object expression shall be C or a class derived from C. [Example:

```c
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
}
```

\(^{14}\) This additional check does not apply to other members, e.g., static data members or enumerator member constants.
p2->i = 3;       // OK (access through a D2)
p2->B::i = 4;    // OK (access through a D2, even though
                 // naming class is B)
int B::* pmi_B = &B::i;
int B::* pmi_B2 = &D2::i; // OK (type of &D2::i is int B::*
B::j = 5;        // OK (because refers to static member)
D2::j = 6;       // OK (because refers to static member)
}

void D2::mem(B* pb, D1* p1) {
    pb->i = 1;       // ill-formed
    p1->i = 2;       // ill-formed
    i = 3;          // OK (access through this)
    B::i = 4;       // OK (access through this, qualification ignored)
    int B::* pmi_B = &B::i;
    int B::* pmi_B2 = &D2::i; // ill-formed
    j = 5;          // OK (because j refers to static member)
    B::j = 6;       // OK (because B::j refers to static member)
}

void g(B* pb, D1* p1, D2* p2) {
    pb->i = 1;       // ill-formed
    p1->i = 2;       // ill-formed
    p2->i = 3;       // ill-formed
}

— end example]

11.6 Access to virtual functions  [class.access.virt]

The access rules (Clause 11) for a virtual function are determined by its declaration and are not affected by
the rules for a function that later overrides it. [Example:

    class B {
        public:
            virtual int f();
    };

    class D : public B {
        private:
            int f();
    };

    void f() {
        D d;
        B* pb = &d;
        D* pd = &d;

        pb->f();       // OK: B::f() is public,
                        // D::f() is invoked
        pd->f();       // error: D::f() is private
    } — end example]

§ 11.6 249
Access is checked at the call point using the type of the expression used to denote the object for which the member function is called (B* in the example above). The access of the member function in the class in which it was defined (D in the example above) is in general not known.

11.7 Multiple access

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 {         // OK: E::I can access E::B 
  B b;            
  int y;          
  void f(E* p, int i) {  
    p->x = i;        // OK: E::I can access E::x  
  }  
};

int g(I* p) {    
  return p->y;    // error: I::y is private      
}
};
```

— end example]
12 Special member functions [special]

1 The default constructor (12.1), copy constructor and copy assignment operator (12.8), move constructor and move assignment operator (12.8), and destructor (12.4) are special member functions. [Note: The implementation will implicitly declare these member functions for some class types when the program does not explicitly declare them. The implementation will implicitly define them if they are used. See 12.1, 12.4 and 12.8. — end note] Programs shall not define implicitly-declared special member functions.

2 Programs may explicitly refer to implicitly-declared special member functions. [Example: a program may explicitly call, take the address of or form a pointer to member to an implicitly-declared special member function.]

struct A { };          // implicitly-declared A::operator=
struct B : A {
    B& operator=(const B&);
};
B& B::operator=(const B& s) {
    this->A::operator=(s);      // well-formed
    return *this;
}

— end example]

3 [Note: The special member functions affect the way objects of class type are created, copied, moved, and destroyed, and how values can be converted to values of other types. Often such special member functions are called implicitly. — end note]

4 Special member functions obey the usual access rules (Clause 11). [Example: declaring a constructor protected ensures that only derived classes and friends can create objects using it. — end example]

12.1 Constructors [class.ctor]

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:

    struct S {
        S();                  // declares the constructor
    };

    S::S() { }           // defines the constructor

    — end example]

2 A constructor is used to initialize objects of its class type. Because constructors do not have names, they are never found during name lookup; however an explicit type conversion using the functional notation (5.2.3) will cause a constructor to be called to initialize an object. [Note: for initialization of objects of class type see 12.6. — end note]

3 A 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 constructor having no parameters is implicitly declared as defaulted (8.4). An implicitly-declared default constructor is an inline public member of its class. A defaulted default constructor for class $X$ is defined as deleted if:

- $X$ is a union-like class that has a variant member with a non-trivial default constructor,
- any non-variant data member with no brace-or-equal-initializer is of reference type,
- any non-variant non-static data member of const-qualified type (or array thereof) with no brace-or-equal-initializer does not have a user-provided default constructor,
- $X$ is a union and all of its variant members are of const-qualified type (or array thereof),
- $X$ is a non-union class and all members of any anonymous union member are of const-qualified type (or array thereof), or
- any direct or virtual base class, or non-static data member with no brace-or-equal-initializer, has class type $M$ (or array thereof) and either $M$ has no default constructor or overload resolution (13.3) as applied to $M$'s default constructor results in an ambiguity or in a function that is deleted or inaccessible from the defaulted default constructor.

A default constructor is trivial if it is neither user-provided nor deleted and if:

- its class has no virtual functions (10.3) and no virtual base classes (10.1), and
- no non-static data member of its class has a brace-or-equal-initializer, 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 default constructor that is defaulted and not defined as deleted is implicitly defined when it is used (3.2) to create an object of its class type (1.8) or when it is explicitly defaulted after its first declaration. The implicitly-defined default constructor performs the set of initializations of the class that would be performed by a user-written default constructor for that class with no 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 defaulted default constructor for a class is implicitly defined, all the non-user-provided default constructors for its base classes and its non-static data members shall have been implicitly defined. [Note: an implicitly-declared default constructor has an exception-specification (15.4). An explicitly-defaulted definition might have an implicit exception-specification, see 8.4. — end note]

Default constructors are called implicitly to create class objects of static, thread, or automatic storage duration (3.7.1, 3.7.2, 3.7.3) defined without an initializer (8.5), are called to create class objects of dynamic storage duration (3.7.4) created by a 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).
A copy constructor (12.8) is used to copy objects of class type. A move constructor (12.8) is used to move the contents of objects of class type.

No return type (not even \texttt{void}) shall be specified for a constructor. A \texttt{return} statement in the body of a constructor shall not specify a return value. The address of a constructor shall not be taken.

A functional notation type conversion (5.2.3) can be used to create new objects of its type. [\textit{Note}: The syntax looks like an explicit call of the constructor. — \textit{end note}] [\texttt{Example}:

\begin{verbatim}
complex zz = complex(1,2.3);
cprint( complex(7.8,1.2) );
\end{verbatim}

— \textit{end example}]

An object created in this way is unnamed. [\textit{Note}: 12.2 describes the lifetime of temporary objects. — \textit{end note}][\textit{Note}: explicit constructor calls do not yield lvalues, see 3.10. — \textit{end note}]

[\textit{Note}: some language constructs have special semantics when used during construction; see 12.6.2 and 12.7. — \textit{end note}] During the construction of a \texttt{const} object, if the value of the object or any of its subobjects is accessed through a glvalue that is not obtained, directly or indirectly, from the constructor’s \texttt{this} pointer, the value of the object or subobject thus obtained is unspecified. [\textit{Example}:

\begin{verbatim}
struct C;
void no_opt(C*);

struct C {
  int c;
  C() : c(0) { no_opt(this); }
};

const C cobj;

void no_opt(C* cpotr) {
  int i = cobj.c * 100; // value of \texttt{cobj.c} is unspecified
  cpotr->c = 1;
  cout << cobj.c * 100 // value of \texttt{cobj.c} is unspecified
       << '\n';
}
\end{verbatim}

— \textit{end example}]

\section*{12.2 Temporary objects}[\texttt{class.temporary}]

Temporary objects of class type are created in various contexts: binding a reference to a prvalue (8.5.3), returning a prvalue (6.6.3), a conversion that creates a prvalue (4.1, 5.2.9, 5.2.11, 5.4), throwing an exception (15.1), entering a \texttt{handler} (15.3), and in some initializations (8.5). [\textit{Note}: the lifetime of exception objects is described in 15.1. — \textit{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. [\textit{Note}: Even if the copy/move constructor is not called, all the semantic restrictions, such as accessibility (Clause 11), shall be satisfied. — \textit{end note}]

[\textit{Example}: Consider the following code:
class X {
public:
    X(int);
    X(const X&);
    ~X();
};

class Y {
public:
    Y(int);
    Y(Y&&);
    ~Y();
};

X f(X);
Y g(Y);

void h() {
    X a(1);
    X b = f(X(2));
    Y c = g(Y(3));
    a = f(a);
}

An implementation might use a temporary in which to construct \texttt{X(2)} before passing it to \texttt{f()} using \texttt{X}'s copy constructor; alternatively, \texttt{X(2)} might be constructed in the space used to hold the argument. Likewise, an implementation might use a temporary in which to construct \texttt{Y(3)} before passing it to \texttt{g()} using \texttt{Y}'s move constructor; alternatively, \texttt{Y(3)} might be constructed in the space used to hold the argument. Also, a temporary might be used to hold the result of \texttt{f(X(2))} before copying it to \texttt{b} using \texttt{X}'s copy constructor; alternatively, \texttt{f()}’s result might be constructed in \texttt{b}. Likewise, a temporary might be used to hold the result of \texttt{g(Y(3))} before moving it to \texttt{c} using \texttt{Y}'s move constructor; alternatively, \texttt{g()}’s result might be constructed in \texttt{c}. On the other hand, the expression \texttt{a=f(a)} requires a temporary for the result of \texttt{f(a)}, which is then assigned to \texttt{a}. — end example

When an implementation introduces a temporary object of a class that has a non-trivial constructor (12.1, 12.8), it shall ensure that a constructor is called for the temporary object. Similarly, the destructor shall be called for a temporary with a non-trivial destructor (12.4). Temporary objects are destroyed as the last step in evaluating the full-expression (1.9) that (lexically) contains the point where they were created. This is true even if that evaluation ends in throwing an exception. The value computations and side effects of destroying a temporary object are associated only with the full-expression, not with any specific subexpression.

There are two contexts in which temporaries are destroyed at a different point than the end of the full-expression. The first context is when a default constructor is called to initialize an element of an array. If the constructor has one or more default arguments, the destruction of every temporary created in a default argument expression is sequenced before the construction of the next array element, if any.

The second context is when a reference is bound to a temporary. The temporary to which the reference is bound or the temporary that is the complete object of a subobject to which the reference is bound persists for the lifetime of the reference except:

— 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.

§ 12.2
The lifetime of a temporary bound to the returned value in a function return statement (6.6.3) is not extended; the temporary is destroyed at the end of the full-expression in the return statement.

A temporary bound to a reference in a new-initializer (5.3.4) persists until the completion of the full-expression containing the new-initializer. [Example:

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

— end example] [Note: This may introduce a dangling reference, and implementations are encouraged to issue a warning in such a case. — end note]

The destruction of a temporary whose lifetime is not extended by being bound to a reference is sequenced before the destruction of every temporary which is constructed earlier in the same full-expression. If the lifetime of two or more temporaries to which references are bound ends at the same point, these temporaries are destroyed at that point in the reverse order of the completion of their construction. In addition, the destruction of temporaries bound to references shall take into account the ordering of destruction of objects with static, thread, or automatic storage duration (3.7.1, 3.7.2, 3.7.3); that is, if `obj1` is an object with the same storage duration as the temporary and created before the temporary is created the temporary shall be destroyed before `obj1` is destroyed; if `obj2` is an object with the same storage duration as the temporary and created after the temporary is created the temporary shall be destroyed after `obj2` is destroyed. [Example:

```cpp
struct S {
    S();
    S(int);
    friend S operator+(const S&, const S&);
    ~S();
};
S obj1;
const S& cr = S(16)+S(23);
S obj2;
```

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

12.3 Conversions

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).
3 [Note: See 13.3 for a discussion of the use of conversions in function calls as well as examples below. — end note]

4 At most one user-defined conversion (constructor or conversion function) is implicitly applied to a single value.

   [Example:
   struct X {
      operator int();
   };

   struct Y {
      operator X();
   };

   Y a;
   int b = a; // error
   // a.operator X().operator int() not tried
   int c = X(a); // OK: a.operator X().operator int()
   — end example]

5 User-defined conversions are used implicitly only if they are unambiguous. A conversion function in a derived class does not hide a conversion function in a base class unless the two functions convert to the same type. Function overload resolution (13.3.3) selects the best conversion function to perform the conversion.

   [Example:
   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 [class.convctor]

1 A constructor declared without the function-specifier explicit specifies a conversion from the types of its parameters to the type of its class. Such a constructor is called a converting constructor. [Example:

   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 ]

2 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
Z a5 = static_cast<Z>(1); // OK: explicit cast used

— end example ]

3 A non-explicit copy/move constructor (12.8) is a converting constructor. An implicitly-declared copy/move
constructor is not an explicit constructor; it may be called for implicit type conversions.

12.3.2 Conversion functions [class.conv.fct]

1 A member function of a class X having no parameters with a name of the form

```
classification-function-id::
    operator conversion-type-id
```

conversion-type-id:

```
type-specifier-seq conversion-declarator_opt
```

conversion-declarator:

```
ptr-operator conversion-declarator_opt
```

specifies a conversion from X to the type specified by the conversion-type-id. Such functions are called
conversion functions. No return type can be specified. If a conversion function is a member function, the
type of the conversion function (8.3.5) is “function taking no parameter returning conversion-type-id”. A
conversion function is never used to convert a (possibly cv-qualified) object to the (possibly cv-qualified)
same object type (or a reference to it), to a (possibly cv-qualified) base class of that type (or a reference to
it), or to (possibly cv-qualified) void.¹¹⁵

[ Example:

```cpp
struct X {
    operator int();
};
```

¹¹⁵ These conversions are considered as standard conversions for the purposes of overload resolution (13.3.3.1, 13.3.3.1.4) and
therefore initialization (8.5) and explicit casts (5.2.9). A conversion to void does not invoke any conversion function (5.2.9).
Even though never directly called to perform a conversion, such conversion functions can be declared and can potentially be
reached through a call to a virtual conversion function in a base class.
void f(X a) {
    int i = int(a);
    i = (int)a;
    i = a;
}

In all three cases the value assigned will be converted by X::operator int(). — end example]

2 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:

    class Y { }
    struct Z {
        explicit operator Y() const;
    };

    void h(Z z) {
        Y y1(z); // OK: direct-initialization
        Y y2 = z; // ill-formed: copy-initialization
        Y y3 = (Y)z; // OK: cast notation
    }

    void g(X a, X b) {
        int i = (a) ? 1+a : 0;
        int j = (a&&b) ? a+b : i;
        if (a) {
            }
    }

    — end example]

3 The conversion-type-id shall not represent a function type nor an array type. The conversion-type-id in a conversion-function-id is the longest 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.
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`.

If a class has no user-declared destructor, a destructor is implicitly declared as defaulted (8.4). An implicitly-declared destructor is an `inline public` member of its class.

A defaulted destructor for a class `X` is defined as deleted if:

- `X` is a union-like class that has a variant member with a non-trivial destructor,
- any of the non-static data members has class type `M` (or array thereof) and `M` has a deleted destructor or a destructor that is inaccessible from the defaulted destructor, or
- any direct or virtual base class has a deleted destructor or a destructor that is inaccessible from the defaulted destructor.

A destructor is trivial if it is neither user-provided nor deleted and if:

- the destructor is not `virtual`,
- all of the direct base classes of its class have trivial destructors, and
- for all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor.

Otherwise, the destructor is `non-trivial`.

A destructor that is defaulted and not defined as deleted is `implicitly defined` when it is used to destroy an object of its class type (3.7) or when it is explicitly defaulted after its first declaration.

Before the defaulted destructor for a class is implicitly defined, all the non-user-provided destructors for its base classes and its non-static data members shall have been implicitly defined. [Note: an implicitly-declared destructor has an `exception-specification` (15.4). An explicitly defaulted definition has no implicit `exception-specification`. — end note]

After executing the body of the destructor and destroying any automatic objects allocated within the body, a destructor for class `X` calls the destructors for `X`'s direct 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).

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]

Destructors are invoked implicitly

- for constructed objects with static storage duration (3.7.1) at program termination (3.6.3),

§ 12.4
— for constructed objects with thread storage duration (3.7.2) at thread exit,
— for constructed objects with automatic storage duration (3.7.3) when the block in which an object is
created exits (6.7),
— for constructed temporary objects when the lifetime of a temporary object ends (12.2),
— for constructed objects allocated by a new-expression (5.3.4), through use of a delete-expression (5.3.5),
— in several situations due to the handling of exceptions (15.3).

A program is ill-formed if an object of class type or array thereof is declared and the destructor for the class
is not accessible at the point of the declaration. Destructors can also be invoked explicitly.

At the point of definition of a virtual destructor (including an implicit definition (12.8)), the non-array
deallocation function is looked up in the scope of the destructor’s class (10.2), and, if no declaration is
found, the function is looked up in the global scope. If the result of this lookup is ambiguous or inaccessible,
or if the lookup selects a placement deallocation function or a function with a deleted definition (8.4), the
program is ill-formed. [Note: this assures that a deallocation function corresponding to the dynamic type
of an object is available for the delete-expression (12.5). — end note]

In an explicit destructor call, the destructor name appears as a ~ followed by a type-name or decltype-
specifier that denotes the destructor’s class type. The invocation of a destructor is subject to the usual
rules for member functions (9.3), that is, if the object is not of the destructor’s class type and not of a class
derived from the destructor’s class type, the program has undefined behavior (except that invoking delete
on a null pointer has no effect). [Example:

```c
struct B {
  virtual ~B() { }
};
struct D : B {
  ~D() { }
};

D D_object;
typedef B B_alias;
B* B_ptr = &D_object;

void f() {
  D_object.B::~B(); // calls B's destructor
  B_ptr->~B(); // calls D's destructor
  B_ptr->B_alias::~B(); // calls 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 opera-
tor (5.2.5) or a qualified-id (5.1); in particular, the unary-expression ~X() in a member function is not an
explicit destructor call (5.3.1). — end note]

[Note: explicit calls of destructors are rarely needed. One use of such calls is for objects placed at specific
addresses using a new-expression with the placement option. Such use of explicit placement and destruction
of objects can be necessary to cope with dedicated hardware resources and for writing memory management
facilities. For example,

```c
void* operator new(std::size_t, void* p) { return p; }
struct X {
```
X(int);
~X();
};
void f(X* p);

void g() {   // rare, specialized use:
  char* buf = new char[sizeof(X)];
  X* p = new(buf) X(222);  // use buf[] and initialize
  f(p);
  p->X::~X();  // cleanup
}

— end note]

12 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]

13  [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

Any allocation function for a class T is a static member (even if not explicitly declared static).

[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]

When an object is deleted with a delete-expression (5.3.5), a deallocation function (operator delete() for
non-array objects or operator delete[] () for arrays) is (implicitly) called to reclaim the storage occupied
by the object (3.7.4.2).

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)\textsuperscript{116}. Otherwise, if the \textit{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.

When a \textit{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.\textsuperscript{117}

Any deallocation function for a class \( X \) is a static member (even if not explicitly declared \texttt{static}). [Example:

```cpp
class X {
    void operator delete(void*);
    void operator delete[](void*, std::size_t);
};

class Y {
    void operator delete(void*, std::size_t);
    void operator delete[](void*);
};

-- end example]
```

Since member allocation and deallocation functions are \texttt{static} they cannot be virtual. [Note: however, when the \textit{cast-expression} of a \textit{delete-expression} refers to an object of class type, because the deallocation function actually called is looked up in the scope of the class that is the dynamic type of the object, if the destructor is virtual, the effect is the same. For example,

```cpp
struct B {
    virtual ~B();
    void operator delete(void*, std::size_t);
};

struct D : B {
    void operator delete(void*);
};

void f() {
    B* bp = new D;
    delete bp;       // uses D::operator delete(void*)
}
```

Here, storage for the non-array object of class \( D \) is deallocated by \( D::\texttt{operator delete()} \), due to the virtual destructor. -- end note] [Note: virtual destructors have no effect on the deallocation function actually called when the \textit{cast-expression} of a \textit{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);
};
```

\textsuperscript{116} A similar provision is not needed for the array version of \texttt{operator delete} because \texttt{5.3.5} requires that in this situation, the static type of the object to be deleted be the same as its dynamic type.

\textsuperscript{117} If the static type of the object to be deleted is different from the dynamic type and the destructor is not virtual the size might be incorrect, but that case is already undefined; see \texttt{5.3.5}.
struct D : B {
    void operator delete[](void*, std::size_t);
};

void f(int i) {
    D* dp = new D[i];
    delete [] dp; // uses D::operator delete[](void*, std::size_t)
    B* bp = new D[i];
    delete[] bp; // undefined behavior
}

— end note]

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 [class.init]

1 When no initializer is specified for an object of (possibly cv-qualified) class type (or array thereof), or the initializer has the form (), the object is initialized as specified in 8.5.

2 An object of class type (or array thereof) can be explicitly initialized; see 12.6.1 and 12.6.2.

3 When an array of class objects is initialized (either explicitly or implicitly) and the elements are initialized by constructor, the constructor shall be called for each element of the array, following the subscript order; see 8.3.4. [Note: destructors for the array elements are called in reverse order of their construction. — end note]

12.6.1 Explicit initialization [class.expl.init]

1 An object of class type can be initialized with a parenthesized expression-list, where the expression-list is construed as an argument list for a constructor that is called to initialize the object. Alternatively, a single assignment-expression can be specified as an initializer using the = form of initialization. Either direct-initialization semantics or copy-initialization semantics apply; see 8.5. [Example:

struct complex {
    complex();
    complex(double);
    complex(double,double);
};

complex sqrt(complex,complex);

complex a(1); // initialize by a call of
    // complex(double)
complex b = a; // initialize by a copy of a
complex c = complex(1,2); // construct complex(1,2)
    // using complex(double,double)
    // copy/move it into c
complex d = sqrt(b,c); // call sqrt(complex,complex)
    // and copy/move the result into d
complex e; // initialize by a call of
    // complex()
complex f = 3; // construct complex(3) using
   // complex(double)
   // copy/move it into f
complex g = { 1, 2 }; // construct complex(1, 2)
   // using complex(double, double)
   // and copy/move it into g

— end example] [Note: overloading of the assignment operator (13.5.3) has no effect on initialization. — end note]

2 An object of class type can also be initialized by a braced-init-list. List-initialization semantics apply; see 8.5 and 8.5.4. [Example:

complex v[6] = { 1, complex(1,2), complex(), 2 };

Here, complex::complex(double) is called for the initialization of v[0] and v[3], complex::complex(double, double) is called for the initialization of v[1], complex::complex() is called for the initialization v[2], v[4], and v[5]. For another example,

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 or thread 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]

1 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-list:
         mem-initializer ...opt
         mem-initializer , mem-initializer-list ...opt
      mem-initializer:
         mem-initializer-id ( expression-listopt )
         mem-initializer-id braced-init-list
      mem-initializer-id:
         class-or-decltype
         identifier

2 In a mem-initializer-id an initial unqualified identifier is looked up in the scope of the constructor’s class and, if not found in that scope, it is looked up in the scope containing the constructor’s definition. [Note: if the constructor’s class contains a member with the same name as a direct or virtual base class of the class, a mem-initializer-id naming the member or base class and composed of a single identifier refers to
the class member. A mem-initializer-id for the hidden base class may be specified using a qualified name.
— end note ] Unless the mem-initializer-id names the constructor’s class, a non-static data member of the
constructor’s class, or a direct or virtual base of that class, the mem-initializer is ill-formed.

3 A mem-initializer-list can initialize a base class using any class-or-decltype that denotes that base class type.
[ Example:

```c
struct A { A(); }; 
typedef A global_A;
struct B { }; 
struct C: public A, public B { C(); }; 
C::C(): global_A() { } // mem-initializer for base A
```

— end example ]

4 If a mem-initializer-id is ambiguous because it designates both a direct non-virtual base class and an inherited
virtual base class, the mem-initializer is ill-formed. [ Example:

```c
struct A { A(); }; 
struct B: public virtual A { }; 
struct C: public A, public B { C(); }; 
C::C(): A() { } // ill-formed: which A?
```

— end example ]

5 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 or for the same base
class, the ctor-initializer is ill-formed.

6 A mem-initializer-list can delegate to another constructor of the constructor’s class using any class-or-decltype that denotes the constructor’s class itself. If a mem-initializer-id designates the constructor’s class,
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
deleagating constructor is executed. If a constructor delegates to itself directly or indirectly, the program is
ill-formed; no diagnostic is required. [ Example:

```c
struct C {
    C( int ) { } // #1: non-delegating constructor
    C(): C(42) { } // #2: delegates to #1
    C( char c ): C(42.0) { } // #3: ill-formed due to recursion with #4
    C( double d ): C('a') { } // #4: ill-formed due to recursion with #3
};
```

— end example ]

7 The expression-list or braced-init-list in a mem-initializer is used to initialize the base class or non-static
data member subobject denoted by the mem-initializer-id according to the initialization rules of 8.5 for
direct-initialization.

[ Example:

```c
struct B1 { B1(int); /* ... */ }; 
struct B2 { B2(int); /* ... */ }; 
struct D : B1, B2 {
    D(int);
```

§ 12.6.2
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. A mem-initializer where the mem-initializer-id denotes a virtual base class is ignored during execution of a constructor of any class that is not the most derived class.

If a given non-static data member or base class is not designated by a mem-initializer-id (including the case where there is no mem-initializer-list because the constructor has no ctor-initializer) and the entity is not a virtual base class of an abstract class (10.4), then

- if the entity is a non-static data member that has a brace-or-equal-initializer, the entity is initialized as specified in 8.5;
- otherwise, if the entity is a variant member (9.5), no initialization is performed;
- otherwise, the entity is default-initialized (8.5).

Example:

```cpp
class A {
    A();
};
class B {
    B(int);   // initializes members as follows:
    A a;     // OK: calls A::A()
    const B b;   // error: B has no default constructor
    int i;     // OK: i has indeterminate value
    int j = 5;   // OK: j has the value 5
};
```

If a given non-static data member has both a brace-or-equal-initializer and a mem-initializer, the initialization specified by the mem-initializer is performed, and the non-static data member's brace-or-equal-initializer is ignored. Example: Given

```cpp
struct A {
    int i = /* some integer expression with side effects */;
    A(int arg) : i(arg) { }
    // ...
};
```
the A(int) constructor will simply initialize i to the value of arg, and the side effects in i’s brace-or-equal-initializer will not take place. — end example]

Initialization proceeds in the following order:

— First, and only for the constructor of the most derived class (1.8), virtual base classes are initialized in the order they appear on a depth-first left-to-right traversal of the directed acyclic graph of base classes, where “left-to-right” is the order of appearance of the base classes in the derived class base-specifier-list.

— Then, direct base classes are initialized in declaration order as they appear in the base-specifier-list (regardless of the order of the member-initializers).

— Then, non-static data members are initialized in the order they were declared in the class definition (again regardless of the order of the member-initializers).

— 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]

11 [Example:

```c
struct V {
    V();
    V(int);
};

struct A : virtual V {
    A();
    A(int);
};

struct B : virtual V {
    B();
    B(int);
};

struct C : A, B, virtual V {
    C();
    C(int);
};

A::A(int i) : V(i) { /* ... */ }
B::B(int i) { /* ... */ }
C::C(int i) { /* ... */ }
```

V v(1); // use V(int)
A a(2); // use V(int)
B b(3); // use V()
C c(4); // use V()

— end example]

12 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]

13 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()), j(f()) { }  // well-defined: bases are all initialized  
    };  

    class C {
      public:
        C(int);  
    };  

    class D : public B, C {
      int i;
    public:
      D() : C(f()), i(f()) { }  // well-defined: bases are all initialized  
    };  

    — end example]

14 [Note: 12.7 describes the result of virtual function calls, typeid and dynamic_casts during construction for the well-defined cases; that is, describes the polymorphic behavior of an object under construction. — end note]
A mem-initializer followed by an ellipsis is a pack expansion (14.5.3) that initializes the base classes specified by a pack expansion in the base-specifier-list for the class. [Example:

```cpp
template<class... Mixins>
class X : public Mixins... {
    public:
        X(const Mixins&... mixins) : Mixins(mixins)... { }
};
```

— end example]

### 12.7 Construction and destruction

1. For an object with a non-trivial constructor, referring to any non-static member or base class of the object before the constructor begins execution results in undefined behavior. For an object with a non-trivial destructor, referring to any non-static member or base class of the object after the destructor finishes execution results in undefined behavior. [Example:

```cpp
struct X { int i; }
struct Y : X { Y(); }; // non-trivial
struct A { int a; }
struct B : public A { int j; Y y; }; // non-trivial

extern B bobj;
B* pb = &bobj; // OK
int* p1 = &bobj.a; // undefined, refers to base class member
int* p2 = &bobj.y.i; // undefined, refers to member's member

A* pa = &bobj; // undefined, upcast to a base class type
B bobj; // definition of bobj

extern X xobj;
int* p3 = &xobj.i; // OK, X is a trivial class
X xobj;
```
struct A { }
struct B : virtual A { }
struct C : B { }
struct D : virtual A { D(A*); }
struct X { X(A*); }
struct E : C, D, X {
    E() : D(this), // undefined: upcast from E* to A*
        // might use path E* → D* → A*
        // but D is not constructed
        // D(C*this), // defined:
        // E* → C* defined because E() has started
        // and C* → A* defined because
        // C fully constructed
    X(this) { // defined: upon construction of X,
        // C/B/D/A sublattice is fully constructed
    }
};

— end example

4 Member functions, including virtual functions (10.3), can be called during construction or destruction (12.6.2). When a virtual function is called directly or indirectly from a constructor (including the mem-initializer or brace-or-equal-initializer for a non-static 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:

```c
struct V {
    virtual void f();
    virtual void g();
};

struct A : virtual V {
    virtual void f();
};

struct B : virtual V {
    virtual void g();
    B(V*, A*);
};

struct D : A, B {
    virtual void f();
    virtual void g();
    D() : B((A*)this, this) { }
};

B::B(V* v, A* a) {
    f(); // calls V::f, not A::f
    g(); // calls B::g, not D::g
    v->g(); // v is base of B, the call is well-defined, calls B::g
}
```
The `typeid` operator (5.2.8) can be used during construction or destruction (12.6.2). When `typeid` is used in a constructor (including the *mem-initializer* or *brace-or-equal-initializer* for a non-static data member) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the operand of `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.

### Example:
```cpp
struct V {
  virtual void f();
};

struct A : virtual V {
};

struct B : virtual V {
  B(V*, A*);
};

struct D : A, B {
  D() : B((A*)this, this) { }
};

B::B(V* v, A* a) {
  typeid(*this); // type_info for B
  typeid(v);    // well-defined: v has type V, a base of B
  typeid(a);   // undefined behavior: type A not a base of B
  dynamic_cast<B*>(v); // well-defined: v of type V*, V base of B
  dynamic_cast<B*>(a); // undefined behavior,
                       // a has type A*, A not a base of B
}
```

### Copying and moving class objects

A class object can be copied or moved in two ways: by initialization (12.1, 8.5), including for function argument passing (5.2.2) and for function value return (6.6.3); and by assignment (5.17). Conceptually, these two operations are implemented by a copy/move constructor (12.1) and copy/move assignment operator (13.5.3).
A non-template constructor for class \( X \) is a copy constructor if its first parameter is of type \( X&, \text{ const } X&, \text{ volatile } X&, \text{ or const volatile } X& \), and either there are no other parameters or else all other parameters have default arguments (8.3.6). [\textit{Example:} \( X::X(\text{const } X&) \) and \( X::X(X&,\text{int}=1) \) are copy constructors.]

```cpp
struct X {
    X(int); // calls X(int);
    X(const X&, int = 1); // calls X(const X&, int);
};
X a(1); // calls X(int);
X b(a, 0); // calls X(const X&, int);
X c = b; // calls X(const X&, int);
```

— \textit{end example}\]

A non-template constructor for class \( X \) is a move constructor if its first parameter is of type \( X&&, \text{ const } X&&, \text{ volatile } X&&, \text{ or const volatile } X&& \), and either there are no other parameters or else all other parameters have default arguments (8.3.6). [\textit{Example:} \( Y::Y(Y&&) \) is a move constructor.]

```cpp
struct Y {
    Y(const Y&); // calls Y(const Y&);
    Y(Y&&); // calls Y(Y&&)
};
extern Y f(int);
Y d(f(1)); // calls Y(Y&&)
Y e = d; // calls Y(const Y&)
```

— \textit{end example}\]

[\textit{Note:} All forms of copy/move constructor may be declared for a class. [\textit{Example:}]

```cpp
struct X {
    X(const X&); // OK
    X(X&); // OK, but possibly not sensible
};
```

— \textit{end example} — \textit{end note}\]

[\textit{Note:} if a class \( X \) only has a copy constructor with a parameter of type \( X& \), an initializer of type \text{ const } X \text{ or volatile } X \text{ cannot initialize an object of type (possibly cv-qualified) } X. [\textit{Example:}]

```cpp
struct X {
    X(); // default constructor
    X(X&); // copy constructor with a nonconst parameter
};
const X cx;
X x = cx; // error: \( X::X(X&) \) cannot copy \( cx \) into \( x \)
```

— \textit{end example} — \textit{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. [\textit{Example:}]

\[ \textsection 12.8 \]
```c++
struct S {
    template<typename T> S(T);
    template<typename T> S(T&&);
    S();
};

S f();
const S g;

void h() {
    S a( f() ); // does not instantiate member template;
    // uses the implicitly generated move constructor
    S a(g);
    // does not instantiate the member template;
    // uses the implicitly generated copy constructor
}

— end example —
```

8 If the class definition does not explicitly declare a copy constructor and there is no user-declared move constructor, a copy constructor is implicitly declared as defaulted (8.4). Thus, for the class definition

```c++
struct X {
    X(const X&, int);
};
```

a copy constructor is implicitly-declared. If the user-declared constructor is later defined as

```c++
X::X(const X& x, int i =0) { /* ... */ }
```

then any use of X’s copy constructor is ill-formed because of the ambiguity; no diagnostic is required.

9 The implicitly-declared copy constructor for a class X will have the form

```c++
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&`.118

Otherwise, the implicitly-declared copy constructor will have the form

```c++
X::X(X&)
```

10 If the class definition does not explicitly declare a move constructor, one will be implicitly declared as defaulted if and only if

— X does not have a user-declared copy constructor and

— the move constructor would not be implicitly defined as deleted.

118) This implies that the reference parameter of the implicitly-declared copy constructor cannot bind to a `volatile` lvalue; see C.1.8.
[Note: When the move constructor is not implicitly declared or explicitly supplied, expressions that otherwise would have invoked the move constructor may instead invoke a copy constructor. — end note]

The implicitly-declared move constructor for class \( X \) will have the form

\[ X::X(X&&) \]

An implicitly-declared copy/move constructor is an **inline public** member of its class. A defaulted copy/move constructor for a class \( X \) is defined as deleted (8.4.3) if \( X \) has:

- a variant member with a non-trivial corresponding constructor and \( X \) is a union-like class,
- a non-static data member of class type \( M \) (or array thereof) that cannot be copied/moved because overload resolution (13.3), as applied to \( M \)'s corresponding constructor, results in an ambiguity or a function that is deleted or inaccessible from the defaulted constructor, or
- a direct or virtual base class \( B \) that cannot be copied/moved because overload resolution (13.3), as applied to \( B \)'s corresponding constructor, results in an ambiguity or a function that is deleted or inaccessible from the defaulted constructor, or
- for the move constructor, a non-static data member or direct or virtual base class with a type that does not have a move constructor and is not trivially copyable.

A copy/move constructor for class \( X \) is trivial if it is neither user-provided nor deleted and if

- class \( X \) has no virtual functions (10.3) and no virtual base classes (10.1), and
- the constructor selected to copy/move each direct base class subobject is trivial, and
- for each non-static data member of \( X \) that is of class type (or array thereof), the constructor selected to copy/move that member is trivial;

otherwise the copy/move constructor is **non-trivial**.

A copy/move constructor that is defaulted and not defined as deleted is **implicitly defined** 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\(^{119}\) or when it is explicitly defaulted after its first declaration. [Note: the copy/move constructor is implicitly defined even if the implementation elided its use (12.2). — end note]

Before the defaulted copy/move constructor for a class is implicitly defined, all non-user-provided copy/move constructors for its direct and virtual base classes and its non-static data members shall have been implicitly defined. [Note: an implicitly-declared copy/move constructor has an **exception-specification** (15.4). An explicitly-defaulted definition (8.4.2) has no implicit exception-specification. — end note]

The implicitly-defined copy constructor for a non-union class \( X \) performs a memberwise copy of its subobjects. [Note: brace-or-equal-initializers of non-static data members are ignored. See also the example in 12.6.2. — end note] The order of copying is the same as the order of initialization of bases and members in a user-defined constructor (see 12.6.2). Each subobject is copied in the manner appropriate to its type:

- if the subobject is of class type, the copy constructor for the class is used;
- if the subobject is an array, each element is copied, in the manner appropriate to the element type;
- if the subobject is of scalar type, the built-in assignment operator is used.

Virtual base class subobjects shall be copied only once by the implicitly-defined copy constructor (see 12.6.2).

The implicitly-defined move constructor for a non-union class \( X \) performs a memberwise move of its subobjects. [Note: brace-or-equal-initializers of non-static data members are ignored. See also the example

\(^{119}\) See 8.5 for more details on direct and copy initialization.
in 12.6.2. — end note] The order of moving is the same as the order of initialization of bases and members in a user-defined constructor (see 12.6.2). Given a parameter named \( x \), each base or non-static data member is moved in the manner appropriate to its type:

- a named member \( m \) of reference or class type \( T \) is direct-initialized with the expression `static_cast<T&&>(x.m);`
- a base class \( B \) is direct-initialized with the expression `static_cast<B&&>(x);`
- an array is initialized by moving each element in the manner appropriate to the element type;
- a scalar type is initialized with the built-in assignment operator.

Virtual base class subobjects shall be moved only once by the implicitly-defined move constructor (see 12.6.2).

The implicitly-defined copy constructor for a union \( X \) copies the object representation (3.9) of \( X \).

A user-declared copy assignment operator \( X::operator= \) is a non-static non-template member function of class \( X \) with exactly one parameter of type \( X, X&, \text{const } X, \text{volatile } X & \) or \( \text{const volatile } X & \).\(^{120}\) [ Note: an overloaded assignment operator must be declared to have only one parameter; see 13.5.3. — end note] [ Note: more than one form of copy assignment operator may be declared for a class. — end note] [ Note: if a class \( X \) only has a copy assignment operator with a parameter of type \( X&, \text{an expression of type const } X \) cannot be assigned to an object of type \( X \).] [ Example:

```cpp
struct X {
    X();
    X& operator=(X&);  // error: X::operator=(X&) cannot assign cx into x
};

const X cx;
X x;
void f() {
    x = cx;
}
```

] — end example] — end note]

If the class definition does not explicitly declare a copy assignment operator and there is no user-declared move assignment operator, a copy assignment operator is implicitly declared as defaulted (8.4). The implicitly-declared copy assignment operator for a class \( X \) will have the form

\[ X& X::operator=(\text{const } X&) \]

if

- each direct base class \( B \) of \( X \) has a copy assignment operator whose parameter is of type \( \text{const } B&, \text{const volatile } B & \) 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 \( \text{const } M&, \text{const volatile } M & \) or \( M \).\(^{121}\)

Otherwise, the implicitly-declared copy assignment operator will have the form

\[ X& X::operator=(X&) \]

\(^{120}\) Because a template assignment operator or an assignment operator taking an rvalue reference parameter is never a copy assignment operator, the presence of such an assignment operator does not suppress the implicit declaration of a copy assignment operator. Such assignment operators participate in overload resolution with other assignment operators, including copy assignment operators, and, if selected, will be used to assign an object.

\(^{121}\) This implies that the reference parameter of the implicitly-declared copy assignment operator cannot bind to a volatile lvalue; see C.1.8.
A user-declared move assignment operator \( X::\text{operator=} \) is a non-static non-template member function of class \( X \) with exactly one parameter of type \( X&&, \) const \( X&&, \) volatile \( X&&, \) or const volatile \( X&&. \) [Note: An overloaded assignment operator must be declared to have only one parameter; see \( \text{13.5.3}. \) — end note]

Note: More than one form of move assignment operator may be declared for a class. — end note]

If the class definition does not explicitly declare a move assignment operator, one will be implicitly declared as defaulted if and only if

- the copy assignment operator is not user-declared and
- the move assignment operator would not be implicitly defined as deleted.

[Example: The class definition

```
struct S {
    int a;
    S& operator=(const S&) = default;
};
```

will not have a default move assignment operator implicitly declared because the copy assignment operator has been user-declared. The move assignment operator may be explicitly defaulted.

```
struct S {
    int a;
    S& operator=(const S&) = default;
    S& operator=(S&&) = default;
};
```

— end example]

The implicitly-declared move assignment operator for a class \( X \) will have the form

\[
& X::\text{operator}=(X&&);
\]

The implicitly-declared copy/move assignment operator for class \( X \) has the return type \( X&; \) it returns the object for which the assignment operator is invoked, that is, the object assigned to. An implicitly-declared copy/move assignment operator is an \text{inline public} member of its class.

A defaulted copy/move assignment operator for class \( X \) is defined as deleted if \( X \) has:

- a variant member with a non-trivial corresponding assignment operator and \( X \) is a union-like class, or
- a non-static data member of \text{const} non-class type (or array thereof), or
- a non-static data member of reference type, or
- a non-static data member of class type \( M \) (or array thereof) that cannot be copied/moved because overload resolution (13.3), as applied to \( M \)’s corresponding assignment operator, results in an ambiguity or a function that is deleted or inaccessible from the defaulted assignment operator, or
- a direct or virtual base class \( B \) that cannot be copied/moved because overload resolution (13.3), as applied to \( B \)’s corresponding assignment operator, results in an ambiguity or a function that is deleted or inaccessible from the defaulted assignment operator, or
- for the move assignment operator, a non-static data member or direct base class with a type that does not have a move assignment operator and is not trivially copyable, or any direct or indirect virtual base class.
Because a copy/move assignment operator is implicitly declared for a class if not declared by the user, a base class copy/move assignment operator is always hidden by the corresponding assignment operator of a derived class (13.5.3). A using-declaration (7.3.3) that brings in from a base class an assignment operator with a parameter type that could be that of a copy/move assignment operator for the derived class is not considered an explicit declaration of such an operator and does not suppress the implicit declaration of the derived class operator; the operator introduced by the using-declaration is hidden by the implicitly-declared operator in the derived class.

A copy/move assignment operator for class \(X\) is trivial if it is neither user-provided nor deleted and if

- class \(X\) has no virtual functions (10.3) and no virtual base classes (10.1), and
- the assignment operator selected to copy/move each direct base class subobject is trivial, and
- for each non-static data member of \(X\) that is of class type (or array thereof), the assignment operator selected to copy/move that member is trivial;

otherwise the copy/move assignment operator is non-trivial.

A copy/move assignment operator that is defaulted and not defined as deleted 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 or when it is explicitly defaulted after its first declaration.

Before the defaulted copy/move assignment operator for a class is implicitly defined, all non-user-provided copy/move assignment operators for its direct base classes and its non-static data members shall have been implicitly defined. [Note: An implicitly-declared copy/move assignment operator has an exception-specification (15.4). An explicitly-defaulted definition has no implicit exception-specification. — end note]

The implicitly-defined copy assignment operator for a non-union class \(X\) performs memberwise copy 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 copy assignment operator. [Example:

```c
struct V { }
struct A : virtual V { }
struct B : virtual V { }
struct C : B, A { }
```

It is unspecified whether the virtual base class subobject \(V\) is assigned twice by the implicitly-defined copy assignment operator for \(C\). — end example]

The implicitly-defined move assignment operator for a non-union 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. Given a parameter named \(x\), each subobject is assigned in the manner appropriate to its type:

- if the subobject is a named member \(c\) of class type \(C\), as if by the expression
  ```c
  this->c = static_cast<C&&>(x.c);
  ```
— if the subobject is a direct base class \( B \), as if by the expression `this->B::operator=(static_cast<B&&>(x))`;
— if the subobject is an array, each element is moved, in the manner appropriate to the element type;
— if the subobject is of scalar type, the built-in assignment operator is used.

32 The implicitly-defined copy assignment operator for a union \( X \) copies the object representation (3.9) of \( X \).

33 A program is ill-formed if the copy/move constructor or the copy/move assignment operator for an object
is implicitly used and the special member function is not accessible (Clause 11). [Note: Copying/moving
one object into another using the copy/move constructor or the copy/move assignment operator does not
change the layout or size of either object. — end note]

34 When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class
object, even if the copy/move constructor and/or destructor for the object have side effects. In such cases,
the implementation treats the source and target of the omitted copy/move operation as simply two different
ways of referring to the same object, and the destruction of that object occurs at the later of the times
when the two objects would have been destroyed without the optimization.122 This elision of copy/move
operations, called *copy elision*, is permitted in the following circumstances (which may be combined to
eliminate multiple copies):

— in a `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/move operation can be omitted by constructing the automatic object directly into the function’s
return value

— in a `throw-expression`, when the operand is the name of a non-volatile automatic object whose scope
does not extend beyond the end of the innermost enclosing `try-block` (if there is one), the copy/move
operation from the operand to the exception object (15.1) can be omitted by constructing the automatic
object directly into the exception object

— when a temporary class object that has not been bound to a reference (12.2) would be copied/moved
to a class object with the same cv-unqualified type, the copy/move operation can be omitted by
constructing the temporary object directly into the target of the omitted copy/move

— when the `exception-declaration` of an exception handler (Clause 15) declares an object of the same type
(except for cv-qualification) as the exception object (15.1), the copy/move operation can be omitted
by treating the `exception-declaration` as an alias for the exception object if the meaning of the program
will be unchanged except for the execution of constructors and destructors for the object declared by
the `exception-declaration`.

[Example:
```cpp
class Thing {
public:
    Thing();
    ~Thing();
    Thing(const Thing&);
};

Thing f() {
    Thing t;
    return t;
}
```]

122) Because only one object is destroyed instead of two, and one copy/move constructor is not executed, there is still one
object destroyed for each one constructed.
Thing t2 = f();

Here the criteria for elision can be combined to eliminate two calls to the copy constructor of class Thing: the copying of the local automatic object t into the temporary object for the return value of function f() and the copying of that temporary object into object t2. Effectively, the construction of the local object t can be viewed as directly initializing the global object t2, and that object’s destruction will occur at program exit. Adding a move constructor to Thing has the same effect, but it is the move construction from the temporary object to t2 that is elided. — end example]

35 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 t2

— end example]

12.9 Inheriting Constructors  [class.inhctor]

1 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:

— all non-template constructors of X, and

— 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

— all constructor templates of X, and
— 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.

2 The constructor characteristics of a constructor or constructor template are

— the template parameter list (14.1), if any,
— the parameter-type-list (8.3.5),
— the exception-specification (15.4),
— absence or presence of explicit (12.3.1), and
— absence or presence of constexpr (7.1.5).

3 For each non-template constructor in the candidate set of inherited constructors other than a constructor
having no parameters or a copy/move constructor having a single parameter, a constructor is implicitly
declared with the same constructor characteristics unless there is a user-declared constructor with the same
signature in the 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]

4 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).

5 [Note: Default and copy/move constructors may be implicitly declared as specified in 12.1 and 12.8. — end
note]

6 [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

— \( B1(\text{const } B1&) \)
— \( B1(B1&&) \)
— \( B1(\text{int}) \)

The set of constructors present in \( D1 \) is

— \( D1() \), implicitly-declared default constructor, ill-formed if used

§ 12.9
— D1(const D1&), implicitly-declared copy constructor, not inherited
— D1(D1&&), implicitly-declared move constructor, not inherited
— D1(int), implicitly-declared inheriting constructor

The candidate set of inherited constructors in D2 for B2 is
— B2(const B2&)
— B2(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(D2&&), implicitly-declared move constructor, not inherited
— D2(int, int), implicitly-declared inheriting constructor
— D2(int), implicitly-declared inheriting constructor
— end example]

[Note: If two using-declarations declare inheriting constructors with the same signatures, the program is ill-formed (9.2, 13.1), because an implicitly-declared constructor introduced by the first using-declaration is not a user-declared constructor and thus does not preclude another declaration of a constructor with the same signature by a subsequent using-declaration. [Example:

```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

§ 12.9
only mem-initializer has a mem-initializer-id that names the base class denoted in the nested-name-specifier of the using-declaration and an expression-list as specified below, and where the compound-statement in its function body is empty (12.6.2). If that user-written constructor would be ill-formed, the program is ill-formed. Each expression in the expression-list is of the form \texttt{static\_cast<T&&>(p)}, where \( p \) is the name of the corresponding constructor parameter and \( T \) is the declared type of \( p \).

Example:

```cpp
struct B1 {
    B1(int) {};
};

struct B2 {
    B2(double) {};
};

struct D1 : B1 {
    using B1::B1; // implicitly declares D1(int)
    int x;
};

void test() {
    D1 d(6); // OK: d.x is not initialized
    D1 e; // error: D1 has no default constructor
}

struct D2 : B2 {
    using B2::B2; // OK: implicitly declares D2(double)
    B1 b;
};

D2 f(1.0); // error: B1 has no default constructor

template< class T >
struct D : T {
    using T::T; // declares all constructors from class T
    ~D() { std::clog << "Destroying wrapper" << std::endl; }
};
```

Class template \( D \) wraps any class and forwards all of its constructors, while writing a message to the standard log whenever an object of class \( D \) is destroyed. — end example]
13 Overloading

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:

```c
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:

---

---

class X {
  static void f();
  void f(); // ill-formed
  void f() const; // ill-formed
  void f() const volatile; // ill-formed
  void g();
  void g() const; // OK: no static g
  void g() const volatile; // OK: no static g
};

§ 13.1
— 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
                      // has a ref-qualifier
};
— end example]

— Note: as specified in 8.3.5, function declarations that have equivalent parameter declarations declare the same function and therefore cannot be overloaded:

— Parameter declarations that differ only in the use of equivalent typedef “types” are equivalent. A typedef is not a separate type, but only a synonym for another type (7.1.3). [Example:

typedef int Int;

    void f(int i);
    void f(Int i);    // OK: redeclaration of f(int)
    void f(int i) { /* ... */ }
    void f(Int i) { /* ... */ } // error: redefinition of f(int)
— end example]

Enumerations, on the other hand, are distinct types and can be used to distinguish overloaded function declarations. [Example:

    enum E { a };

    void f(int i) { /* ... */ }
    void f(E i) { /* ... */ }
— end example]

— Parameter declarations that differ only in a pointer * versus an array [] are equivalent. That is, the array declaration is adjusted to become a pointer declaration (8.3.5). Only the second and subsequent array dimensions are significant in parameter types (8.3.4). [Example:

    int f(char*);
    int f(char[]);    // same as f(char*);
    int f(char[7]);  // same as f(char*);
    int f(char[9]);  // same as f(char*);

    int g(char*)[10]);
    int g(char[5][10]); // same as g(char(*)[10]);
    int g(char[7][10]); // same as g(char(*)[10]);
    int g(char*)[20]);  // different from g(char(*)[10]);
— end example]
— Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent. That is, the function type is adjusted to become a pointer to function type (8.3.5). [Example:

```c
void h(int());
void h(int (*)(()));  // redeclaration of h(int())
void h(int x());      // definition of h(int())
void h(int (*)(*)()); // ill-formed: redefinition of h(int())
```
— end example]

— Parameter declarations that differ only in the presence or absence of `const` and/or `volatile` are equivalent. That is, the `const` and `volatile` type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called. [Example:

```c
typedef const int cInt;

int f (int);
int f (const int);       // redeclaration of f(int)
int f (int) { ... }     // definition of f(int)
int f (cInt) { ... }    // error: redefinition of f(int)
```
— end example]

Only the `const` and `volatile` type-specifiers at the outermost level of the parameter type specification are ignored in this fashion; `const` and `volatile` type-specifiers buried within a parameter type specification are significant and can be used to distinguish overloaded function declarations. In particular, for any type T, “pointer to T,” “pointer to const T,” and “pointer to volatile T” are considered distinct parameter types, as are “reference to T,” “reference to const T,” and “reference to volatile T.”

— Two parameter declarations that differ only in their default arguments are equivalent. [Example: consider the following:

```c
void f (int i, int j);
void f (int i, int j = 99);  // OK: redeclaration of f(int, int)
void f (int i = 88, int j);  // OK: redeclaration of f(int, int)
void f ();                  // OK: overloaded declaration of f

void prog () {
  f (1, 2);                 // OK: call f(int, int)
  f (1);                    // OK: call f(int, int)
  f ();                     // Error: f(int, int) or f()?
}
```
— end example] — end note]

### 13.2 Declaration matching

Two function declarations of the same name refer to the same function if they are in the same scope and have equivalent parameter declarations (13.1). A function member of a derived class is not in the same scope as a function member of the same name in a base class. [Example:

---

123) When a parameter type includes a function type, such as in the case of a parameter type that is a pointer to function, the `const` and `volatile` type-specifiers at the outermost level of the parameter type specifications for the inner function type are also ignored.
struct B {
    int f(int);
};

struct D : B {
    int f(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
void f(char*);
void g() {
    extern void 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)
        callee(88, 99); // error: only callee(int) in scope
    }
}
```

— end example]

3 Different versions of an overloaded member function can be given different access rules. [Example:

```c
class buffer {
private:
    char* p;
    int size;
protected:
    buffer(int s, char* store) { size = s; p = store; }
public:
    buffer(int s) { p = new char[size = s]; }
};
```

— end example]

13.3 Overload resolution [over.match]

Overload resolution is a mechanism for selecting the best function to call given a list of expressions that are to be the arguments of the call and a set of candidate functions that can be called based on the context of the call. The selection criteria for the best function are the number of arguments, how well the arguments
match the parameter-type-list of the candidate function, how well (for non-static member functions) the object matches the implicit object parameter, and certain other properties of the candidate function. [Note: the function selected by overload resolution is not guaranteed to be appropriate for the context. Other restrictions, such as the accessibility of the function, can make its use in the calling context ill-formed. — end note]

Overload resolution selects the function to call in seven distinct contexts within the language:

— 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 a glvalue or class prvalue to which a reference (8.5.3) will be directly bound (13.3.1.6).

Each of these contexts defines the set of candidate functions and the list of arguments in its own unique way. But, once the candidate functions and argument lists have been identified, the selection of the best function is the same in all cases:

— 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 \textit{ref-qualifier} or with the \& \textit{ref-qualifier}
- “rvalue reference to \( cv\ X \)” for functions declared with the \&\& \textit{ref-qualifier}

where \( X \) is the class of which the function is a member and \( cv \) is the \textit{cv}-qualification on the member function declaration. \[ Example: \textit{for a const} member function of class \( X \), the extra parameter is assumed to have type “reference to \textit{const} \( X \)”.

During overload resolution, the implied object argument is indistinguishable from other arguments. The implicit object parameter, however, retains its identity since conversions on the corresponding argument shall obey these additional rules:

- no temporary object can be introduced to hold the argument for the implicit object parameter; and
- no user-defined conversions can be applied to achieve a type match with it.

For non-static member functions declared without a \textit{ref-qualifier}, an additional rule applies:

- even if the implicit object parameter is not \textit{const}-qualified, an rvalue can be bound to the parameter as long as in all other respects the argument can be converted to the type of the implicit object parameter. \[ Note: \textit{The fact that such an argument is an rvalue does not affect the ranking of implicit conversion sequences} \( 13.3.3 \). \]

Because other than in list-initialization only one user-defined conversion is allowed in an implicit conversion sequence, special rules apply when selecting the best user-defined conversion \( 13.3.3, 13.3.3.1 \). \[ Example: \]

```c++
class T {
public:
  T();
};

class C : T {
public:
  C(int);
};
T a = 1; // ill-formed: T(C(1)) not tried
```

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.\footnote{124} A given name can refer to one or more function templates and also to a set

\[\textit{end example}\]

\[\textit{end note}\]
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

1. In a function call (5.2.2)

   ```
   postfix-expression ( expression-list_opt )
   ```

   if the `postfix-expression` denotes a set of overloaded functions and/or function templates, overload resolution is applied as specified in 13.3.1.1.1. If the `postfix-expression` denotes an object of class type, overload resolution is applied as specified in 13.3.1.1.2.

   If the `postfix-expression` denotes the address of a set of overloaded functions and/or function templates, overload resolution is applied using that set as described above. If the function selected by overload resolution is a non-static member function, the program is ill-formed. [Note: the resolution of the address of an overload set in other contexts is described in 13.4. — end note]

2. **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:

   ```
   postfix-expression:
   postfix-expression . id-expression
   postfix-expression -> id-expression
   primary-expression
   ```

   These represent two syntactic subcategories of function calls: qualified function calls and unqualified function calls.

2. In qualified function calls, the name to be resolved is an `id-expression` and is preceded by an `->` or `.` operator. Since the construct `A->B` is generally equivalent to `(A).B`, the rest of Clause 13 assumes, without loss of generality, that all member function calls have been normalized to the form that uses an object and the `.` operator. Furthermore, Clause 13 assumes that the `postfix-expression` that is the left operand of the `.` operator has type “cv T” where T denotes a class. Under this assumption, the `id-expression` in the call is looked up as a member function of T following the rules for looking up names in classes (10.2). The function declarations found by that lookup constitute the set of candidate functions. The argument list is the `expression-list` in the call augmented by the addition of the left operand of the `.` operator in the normalized member function call as the implied object argument (13.3.1).

3. In unqualified function calls, the name is not qualified by an `->` or `.` operator and has the more general form of a `primary-expression`. The name is looked up in the context of the function call following the normal rules for name lookup in function calls (3.4). The function declarations found by that lookup constitute the set of candidate functions. Because of the rules for name lookup, the set of candidate functions consists (1) entirely of non-member functions or (2) entirely of member functions of some class T. In case (1), the argument list is the same as the `expression-list` in the call. In case (2), the argument list is the `expression-list` in the call augmented by the addition of an implied object argument as in a qualified function call. If the keyword `this` (9.3.2) is in scope and refers to class T, or a derived class of T, then the implied object argument is `(this)`. If the keyword `this` is not in scope or refers to another class, then a contrived object of type

---

125) Note that cv-qualifiers on the type of objects are significant in overload resolution for both glvalue and class prvalue objects.
T becomes the implied object argument\(^{126}\). If the argument list is augmented by a contrived object and overload resolution selects one of the non-static member functions of T, the call is ill-formed.

### 13.3.1.1.2 Call to object of class type

1. If the *primary-expression* E in the function call syntax evaluates to a class object of type “cv T”, then the set of candidate functions includes at least the function call operators of T. The function call operators of T are obtained by ordinary lookup of the name `operator()` in the context of `(E).operator()`.

2. In addition, for each non-explicit conversion function declared in T of the form

   ```
   operator conversion-type-id () attribute-specifier_opt cv-qualifier;
   ```

   where `cv-qualifier` is the same cv-qualification as, or a greater cv-qualification than, `cv`, and where `conversion-type-id` denotes the type “pointer to function of (P1,...,Pn) returning R”, or the type “reference to pointer to function of (P1,...,Pn) returning R”, or the type “reference to function of (P1,...,Pn) returning R”, a *surrogate call function* with the unique name `call-function` and having the form

   ```
   R call-function (conversion-type-id F, P1 a1, ... ,Pn an) { return F (a1,... ,an); }
   ```

   is also considered as a candidate function. Similarly, surrogate call functions are added to the set of candidate functions for each non-explicit conversion function declared in a base class of T provided the function is not hidden within T by another intervening declaration\(^{127}\).

3. If such a surrogate call function is selected by overload resolution, the corresponding conversion function will be called to convert E to the appropriate function pointer or reference, and the function will then be invoked with the arguments of the call. If the conversion function cannot be called (e.g., because of an ambiguity), the program is ill-formed.

4. The argument list submitted to overload resolution consists of the argument expressions present in the function call syntax preceded by the implied object argument `(E).` [Note: when comparing the call against the function call operators, the implied object argument is compared against the implicit object parameter of the function call operator. When comparing the call against a surrogate call function, the implied object argument is compared against the first parameter of the surrogate call function. The conversion function from which the surrogate call function was derived will be used in the conversion sequence for that parameter since it converts the implied object argument to the appropriate function pointer or reference required by that first parameter. — end note] [Example:

   ```
   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
   ```

---

\(^{126}\) An implied object argument must be contrived to correspond to the implicit object parameter attributed to member functions during overload resolution. It is not used in the call to the selected function. Since the member functions all have the same implicit object parameter, the contrived object will not be the cause to select or reject a function.

\(^{127}\) Note that this construction can yield candidate call functions that cannot be differentiated one from the other by overload resolution because they have identical declarations or differ only in their return type. The call will be ambiguous if overload resolution cannot select a match to the call that is uniquely better than such undifferentiable functions.
13.3.1.2 Operators in expressions

If no operand of an operator in an expression has a type that is a class or an enumeration, the operator is assumed to be a built-in operator and interpreted according to Clause 5. [Note: because . . *, and :: cannot be overloaded, these operators are always built-in operators interpreted according to Clause 5. ?: cannot be overloaded, but the rules in this subclause are used to determine the conversions to be applied to the second and third operands when they have class or enumeration type (5.16). — end note] [Example:

```c
struct String {
  String(const String&);
  String(char*);
  operator char*();
};
String operator + (const String&);  // ill-formed because neither
int I = 1 + 1;                      // operand has user-defined type
// Always evaluates to 2 even if
// user-defined types exist which
// would perform the operation.
}
```

— end example]

2 If either operand has a type that is a class or an enumeration, a user-defined operator function might be declared that implements this operator or a user-defined conversion can be necessary to convert the operand to a type that is appropriate for a built-in operator. In this case, overload resolution is used to determine which operator function or built-in operator is to be invoked to implement the operator. Therefore, the operator notation is first transformed to the equivalent function-call notation as summarized in Table 10 (where @ denotes one of the operators covered in the specified subclause).

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Expression</th>
<th>As member function</th>
<th>As non-member function</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.5.1</td>
<td>@a</td>
<td>(a).operator@()</td>
<td>operator@ (a)</td>
</tr>
<tr>
<td>13.5.2</td>
<td>a@b</td>
<td>(a).operator@ (b)</td>
<td>operator@ (a, b)</td>
</tr>
<tr>
<td>13.5.3</td>
<td>a=b</td>
<td>(a).operator= (b)</td>
<td></td>
</tr>
<tr>
<td>13.5.5</td>
<td>a[b]</td>
<td>(a).operator[] (b)</td>
<td></td>
</tr>
<tr>
<td>13.5.6</td>
<td>a-&gt;</td>
<td>(a).operator-&gt; ()</td>
<td></td>
</tr>
<tr>
<td>13.5.7</td>
<td>a@</td>
<td>(a).operator@ (0)</td>
<td>operator@ (a, 0)</td>
</tr>
</tbody>
</table>

3 For a unary operator @ with an operand of a type whose cv-unqualified version is T1, and for a binary operator @ with a left operand of a type whose cv-unqualified version is T1 and a right operand of a type whose cv-unqualified version is T2, three sets of candidate functions, designated member candidates, non-member candidates and built-in candidates, are constructed as follows:

— 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

§ 13.3.1.2 291
non-member functions in the lookup set that have a first parameter of type \( T_1 \) or “reference to (possibly cv-qualified) \( T_1 \)”, when \( T_1 \) is an enumeration type, or (if there is a right operand) a second parameter of type \( T_2 \) or “reference to (possibly cv-qualified) \( T_2 \)”, when \( T_2 \) is an enumeration type, are candidate functions.

— For the operator \( , \), the unary operator \&, or the operator \( \rightarrow \), 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.

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.

For all other operators, no such restrictions apply.

The set of candidate functions for overload resolution is the union of the member candidates, the non-member candidates, and the built-in candidates. The argument list contains all of the operands of the operator. The best function from the set of candidate functions is selected according to 13.3.2 and 13.3.3.

Example:

```cpp
struct A {
    operator int();
};
A operator+(const A&, const A&);
int main() {
    A a, b;
    a + b; // operator+(a,b) chosen over int(a) + int(b)
}
```

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.

The second operand of operator \( \rightarrow \) is ignored in selecting an operator-\( \rightarrow \) function, and is not an argument when the operator-\( \rightarrow \) function is called. When operator-\( \rightarrow \) returns, the operator \( \rightarrow \) is applied to the value returned, with the original second operand.

If the operator is the operator \( , \), the unary operator \&, or the operator \( \rightarrow \), and there are no viable functions, then the operator is assumed to be the built-in operator and interpreted according to Clause 5.

[Note: the lookup rules for operators in expressions are different than the lookup rules for operator function names in a function call, as shown in the following example:]

---

128) If the set of candidate functions is empty, overload resolution is unsuccessful.

129) If the value returned by the operator-\( \rightarrow \) function has class type, this may result in selecting and calling another operator-\( \rightarrow \) function. The process repeats until an operator-\( \rightarrow \) function returns a value of non-class type.
struct A { };
void operator + (A, A);

struct B {
    void operator + (B);
    void f ();
};

A a;

void B::f() {
    operator+ (a,a); // error: global operator hidden by member
    a + a; // OK: calls global operator+
}

— end note]

13.3.1.3 Initialization by constructor
[over.match.ctor]

1 When objects of class type are direct-initialized (8.5), or copy-initialized from an expression of the same or a derived class type (8.5), overload resolution selects the constructor. For direct-initialization, the candidate functions are all the constructors of the class of the object being initialized. For copy-initialization, the candidate functions are all the converting constructors (12.3.1) of that class. The argument list is the expression-list or assignment-expression of the initializer.

13.3.1.4 Copy-initialization of class by user-defined conversion
[over.match.copy]

1 Under the conditions specified in 8.5, as part of a copy-initialization of an object of class type, a user-defined conversion can be invoked to convert an initializer expression to the type of the object being initialized. Overload resolution is used to select the user-defined conversion to be invoked. Assuming that “cv1 T” is the type of the object being initialized, with T a class type, the candidate functions are selected as follows:

— The converting constructors (12.3.1) of T are candidate functions.

— When the type of the initializer expression is a class type “cv S”, the non-explicit conversion functions of S and its base classes are considered. When initializing a temporary to be bound to the first parameter of a copy constructor (12.8) called with a single argument in the context of direct-initialization, explicit conversion functions are also considered. Those that are not hidden within S and yield a type whose cv-unqualified version is the same type as T or is a derived class thereof are candidate functions. Conversion functions that return “reference to X” return lvalues or xvalues, depending on the type of reference, of type X and are therefore considered to yield X for this process of selecting candidate functions.

2 In both cases, the argument list has one argument, which is the initializer expression. [Note: this argument will be compared against the first parameter of the constructors and against the implicit object parameter of the conversion functions. — end note]

13.3.1.5 Initialization by conversion function
[over.match.conv]

1 Under the conditions specified in 8.5, as part of an initialization of an object of nonclass type, a conversion function can be invoked to convert an initializer expression of class type to the type of the object being initialized. Overload resolution is used to select the conversion function to be invoked. Assuming that “cv1 T” is the type of the object being initialized, and “cv S” is the type of the initializer expression, with S a class type, the candidate functions are selected as follows:
The conversion functions of \( S \) and its base classes are considered. Those non-explicit conversion functions that are not hidden within \( S \) and yield type \( T \) or a type that can be converted to type \( T \) via a standard conversion sequence (13.3.3.1.1) are candidate functions. For direct-initialization, those explicit conversion functions that are not hidden within \( S \) and yield type \( T \) or a type that can be converted to type \( T \) with a qualification conversion (4.4) are also candidate functions. Conversion functions that return a cv-qualified type are considered to yield the cv-unqualified version of that type for this process of selecting candidate functions. Conversion functions that return “reference to \( X \)” return lvalues or xvalues, depending on the type of reference, of type “\( cv2 \ X \)” and are therefore considered to yield \( X \) for this process of selecting candidate functions.

2. The argument list has one argument, which is the initializer expression. [Note: this argument will be compared against the implicit object parameter of the conversion functions. — end note]

13.3.1.6 Initialization by conversion function for direct reference binding [over.match.ref]

1. Under the conditions specified in 8.5.3, a reference can be bound directly to a glvalue or class prvalue that is the result of applying a conversion function to an initializer expression. Overload resolution is used to select the conversion function to be invoked. Assuming that “\( cv1 \ T \)” is the underlying type of the reference being initialized, and “\( cv \ S \)” is the type of the initializer expression, with \( S \) a class type, the candidate functions are selected as follows:

— 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 \( cv2 \ T2 \)” (when 8.5.3 requires an lvalue result) or “\( cv2 \ T2 \)” or “rvalue reference to \( cv2 \ T2 \)” (when 8.5.3 requires an rvalue result), where “\( cv1 \ T \)” is reference-compatible (8.5.3) with “\( cv2 \ T2 \)”, 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]

13.3.1.7 Initialization by list-initialization [over.match.list]

1. When objects of non-aggregate class type are list-initialized (8.5.4), overload resolution selects the constructor as follows, where \( T \) is the cv-unqualified class type of the object being initialized:

— If \( T \) has an initializer-list constructor (8.5.4), the argument list consists of the initializer list as a single argument; otherwise, the argument list consists of the elements of the initializer list.

— For direct-list-initialization, the candidate functions are all the constructors of the class \( T \).

— For copy-list-initialization, the candidate functions are all the constructors of \( T \). However, if an explicit constructor is chosen, the initialization is ill-formed. [Note: This restriction only applies if this initialization is part of the final result of overload resolution — end note]

13.3.2 Viable functions [over.match.viable]

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). For the purposes of overload resolution, the parameter list is truncated on the right, so that there are exactly \( m \) parameters.

Second, for \( F \) to be a viable function, there shall exist for each argument an *implicit conversion sequence* (13.3.3.1) that converts that argument to the corresponding parameter of \( F \). If the parameter has reference type, the implicit conversion sequence includes the operation of binding the reference, and the fact that an lvalue reference to non-\texttt{const} cannot be bound to an rvalue and that an rvalue reference cannot be bound to an lvalue can affect the viability of the function (see 13.3.3.1.4).

### 13.3.3 Best viable function

1. Define \( \text{ICS}_i(F) \) as follows:

   - if \( F \) is a static member function, \( \text{ICS}_i(F) \) is defined such that \( \text{ICS}_i(F) \) is neither better nor worse than \( \text{ICS}_i(G) \) for any function \( G \), and, symmetrically, \( \text{ICS}_i(G) \) is neither better nor worse than \( \text{ICS}_i(F) \); otherwise,

   - let \( \text{ICS}_i(F) \) denote the implicit conversion sequence that converts the \( i \)-th argument in the list to the type of the \( i \)-th parameter of viable function \( F \). 13.3.3.1 defines the implicit conversion sequences and 13.3.3.2 defines what it means for one implicit conversion sequence to be a better conversion sequence or worse conversion sequence than another.

Given these definitions, a viable function \( F_1 \) is defined to be a *better* function than another viable function \( F_2 \) if for all arguments \( i \), \( \text{ICS}_i(F_1) \) is not a worse conversion sequence than \( \text{ICS}_i(F_2) \), and then

- for some argument \( j \), \( \text{ICS}_j(F_1) \) is a better conversion sequence than \( \text{ICS}_j(F_2) \), or, if not that,

- 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] or, if not that,

- \( F_1 \) is a non-template function and \( F_2 \) is a function template specialization, or, if not that,

---

130) According to 8.3.6, parameters following the \((m+1)\)-st parameter must also have default arguments.

131) If a function is a static member function, this definition means that the first argument, the implied object argument, has no effect in the determination of whether the function is better or worse than any other function.
— 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. \[132\]

\[\text{Example:}\]

```cpp
void Fcn(const int*, short);
void Fcn(int*, int);

int i;
short s = 0;

void f() {
    Fcn(&i, s); // is ambiguous because
    // &i \rightarrow int* is better than &i \rightarrow const int*
    // but s \rightarrow short is also better than s \rightarrow int

    Fcn(&i, 1L); // calls Fcn(int*, int), because
    // &i \rightarrow int* is better than &i \rightarrow const int*
    // and 1L \rightarrow short and 1L \rightarrow int are indistinguishable

    Fcn(&i, 'c'); // calls Fcn(int*, int), because
    // &i \rightarrow int* is better than &i \rightarrow const int*
    // and c \rightarrow int is better than c \rightarrow short
}
```

— end example]

3 If the best viable function resolves to a function for which multiple declarations were found, and if at least two of these declarations — or the declarations they refer to in the case of using-declarations — specify a default argument that made the function viable, the program is ill-formed. \[\text{Example:}\]

```cpp
namespace A {
    extern "C" void f(int = 5);
}
namespace B {
    extern "C" void f(int = 5);
}
using A::f;
using B::f;

void use() {
    f(3); // OK, default argument was not used for viability
    f(); // Error: found default argument twice
}
```

— end example]

132) The algorithm for selecting the best viable function is linear in the number of viable functions. Run a simple tournament to find a function \(W\) that is not worse than any opponent it faced. Although another function \(F\) that \(W\) did not face might be at least as good as \(W\), \(F\) cannot be the best function because at some point in the tournament \(F\) encountered another function \(G\) such that \(F\) was not better than \(G\). Hence, \(W\) is either the best function or there is no best function. So, make a second pass over the viable functions to verify that \(W\) is better than all other functions.
13.3.3.1 Implicit conversion sequences

An implicit conversion sequence is a sequence of conversions used to convert an argument in a function call to the type of the corresponding parameter of the function being called. The sequence of conversions is an implicit conversion as defined in Clause 4, which means it is governed by the rules for initialization of an object or reference by a single expression (8.5, 8.5.3).

Implicit conversion sequences are concerned only with the type, cv-qualification, and value category of the argument and how these are converted to match the corresponding properties of the parameter. Other properties, such as the lifetime, storage class, alignment, or accessibility of the argument and whether or not the argument is a bit-field are ignored. So, although an implicit conversion sequence can be defined for a given argument-parameter pair, the conversion from the argument to the parameter might still be ill-formed in the final analysis.

A well-formed implicit conversion sequence is one of the following forms:

— a standard conversion sequence (13.3.3.1.1),
— a user-defined conversion sequence (13.3.3.1.2), or
— an ellipsis conversion sequence (13.3.3.1.3).

However, when considering the argument of a constructor or user-defined conversion function that is a candidate by 13.3.1.3 when invoked for the copying/moving of the temporary in the second step of a class copy-initialization, by 13.3.1.7 when passing the initializer list as a single argument or when the initializer list has exactly one element and a conversion to some class X or reference to (possibly cv-qualified) X is considered for the first parameter of a constructor of X, 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 considered.

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 a prvalue of the type of the parameter. [Note: when the parameter has a class type, this is a conceptual conversion defined for the purposes of Clause 13; the actual initialization is defined in terms of constructors and is not a conversion. — end note] Any difference in top-level cv-qualification is subsumed by the initialization itself and does not constitute a conversion. [Example: a parameter of type A can be initialized from an argument of type const A. The implicit conversion sequence for that case is the identity sequence; it contains no “conversion” from const A to A. — end example] When the parameter has a class type and the argument expression has the same type, the implicit conversion sequence is an identity conversion. When the parameter has a class type and the argument expression has a derived class type, the implicit conversion sequence is a derived-to-base Conversion from the derived class to the base class. [Note: there is no such standard conversion; this derived-to-base Conversion exists only in the description of implicit conversion sequences. — end note] A derived-to-base Conversion has Conversion rank (13.3.3.1.1).

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\(^{133}\). If a function that uses the ambiguous conversion sequence is selected as the best viable function, the call will be ill-formed because the conversion of one of the arguments in the call is ambiguous.

The three forms of implicit conversion sequences mentioned above are defined in the following subclauses.

### 13.3.3.1.1 Standard conversion sequences

Table 11 summarizes the conversions defined in Clause 4 and partitions them into four disjoint categories: Lvalue Transformation, Qualification Adjustment, Promotion, and Conversion. [Note: these categories are orthogonal with respect to value category, cv-qualification, and data representation: the Lvalue Transformations do not change the cv-qualification or data representation of the type; the Qualification Adjustments do not change the value category or data representation of the type; and the Promotions and Conversions do not change the value category or cv-qualification of the type. — end note]

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

---

\(^{133}\) The ambiguous conversion sequence is ranked with user-defined conversion sequences because multiple conversion sequences for an argument can exist only if they involve different user-defined conversions. The ambiguous conversion sequence is indistinguishable from any other user-defined conversion sequence because it represents at least two user-defined conversion sequences, each with a different user-defined conversion, and any other user-defined conversion sequence must be indistinguishable from at least one of them.

This rule prevents a function from becoming non-viable because of an ambiguous conversion sequence for one of its parameters. Consider this example,

```cpp
class B;
class A { A (B&); };
class B { operator A () ; };
class C { C (B&); };
void f(A) { }
void f(C) { }
B b;
f(b); // ambiguous because b → C via constructor and
// b → A via constructor or conversion function.
```

If it were not for this rule, \(f(A)\) would be eliminated as a viable function for the call \(f(b)\) causing overload resolution to select \(f(C)\) as the function to call even though it is not clearly the best choice. On the other hand, if an \(f(B)\) were to be declared then \(f(b)\) would resolve to that \(f(B)\) because the exact match with \(f(B)\) is better than any of the sequences required to match \(f(A)\).
Table 11 — Conversions

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Category</th>
<th>Rank</th>
<th>Subclause</th>
</tr>
</thead>
<tbody>
<tr>
<td>No conversions required</td>
<td>Identity</td>
<td></td>
<td>4.1</td>
</tr>
<tr>
<td>Lvalue-to-rvalue conversion</td>
<td>Lvalue Transformation</td>
<td>Exact Match</td>
<td>4.2</td>
</tr>
<tr>
<td>Array-to-pointer conversion</td>
<td></td>
<td></td>
<td>4.3</td>
</tr>
<tr>
<td>Function-to-pointer conversion</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Qualification conversions</td>
<td>Qualification Adjustment</td>
<td></td>
<td>4.4</td>
</tr>
<tr>
<td>Integral promotions</td>
<td></td>
<td>Promotion</td>
<td>4.5</td>
</tr>
<tr>
<td>Floating point promotion</td>
<td></td>
<td>Promotion</td>
<td>4.6</td>
</tr>
<tr>
<td>Integral conversions</td>
<td></td>
<td></td>
<td>4.7</td>
</tr>
<tr>
<td>Floating point conversions</td>
<td></td>
<td></td>
<td>4.8</td>
</tr>
<tr>
<td>Floating-integral conversions</td>
<td></td>
<td></td>
<td>4.9</td>
</tr>
<tr>
<td>Pointer conversions</td>
<td></td>
<td></td>
<td>4.10</td>
</tr>
<tr>
<td>Pointer to member conversions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boolean conversions</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

function (12.3.2), the initial standard conversion sequence converts the source type to the implicit object parameter of the conversion function.

2 The second standard conversion sequence converts the result of the user-defined conversion to the target type for the sequence. Since an implicit conversion sequence is an initialization, the special rules for initialization by user-defined conversion apply when selecting the best user-defined conversion for a user-defined conversion sequence (see 13.3.3 and 13.3.3.1).

3 If the user-defined conversion is specified by a specialization of a conversion function template, the second standard conversion sequence shall have exact match rank.

4 A conversion of an expression of class type to the same class type is given Exact Match rank, and a conversion of an expression of class type to a base class of that type is given Conversion rank, in spite of the fact that a copy/move constructor (i.e., a user-defined conversion function) is called for those cases.

13.3.3.1.3 Ellipsis conversion sequences

1 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

1 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.

When a parameter of reference type is not bound directly to an argument expression, the conversion sequence is the one required to convert the argument expression to the underlying type of the reference according to 13.3.3.1. Conceptually, this conversion sequence corresponds to copy-initializing a temporary of the underlying type with the argument expression. Any difference in top-level cv-qualification is subsumed by the initialization itself and does not constitute a conversion.

Except for an implicit object parameter, for which see 13.3.1, a standard conversion sequence cannot be formed if it requires binding an lvalue reference to non-\texttt{const} to an rvalue or binding an rvalue reference to an lvalue. [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 reference-compatible with added qualification influences the rank of a standard conversion; see 13.3.3.2 and 8.5.3.

13.3.3.1.5 List-initialization sequence [over.ics.list]

When an argument is an initializer list (8.5.4), it is not an expression and special rules apply for converting it to a parameter type.

If the parameter type is \texttt{std::initializer_list\langle X\rangle} and all the elements of the initializer list can be implicitly converted to \texttt{X}, the implicit conversion sequence is the worst conversion necessary to convert an element of the list to \texttt{X}. This conversion can be a user-defined conversion even in the context of a call to an initializer-list constructor. [Example:

\begin{verbatim}
void f(std::initializer_list\langle int\rangle);
 f( \{1,2,3\} ); // OK: f(initializer_list\langle int\rangle) identity conversion
 f( \{'a','b'\} ); // OK: f(initializer_list\langle int\rangle) integral promotion
 f( \{1.0\} ); // error: narrowing

struct A {
    A(std::initializer_list\langle double\rangle); // #1
    A(std::initializer_list\langle complex\langle double\rangle\rangle); // #2
    A(std::initializer_list\langle std::string\rangle); // #3
};
 A a{ 1.0,2.0 }; // OK, uses #1

void g(A);
g(\{ "foo", "bar" \}); // OK, uses #3
\end{verbatim}

— end example]

Otherwise, if the parameter is a non-aggregate class \texttt{X} and overload resolution per 13.3.1.7 chooses a single best constructor of \texttt{X} to perform the initialization of an object of type \texttt{X} from the argument initializer list, the
implicit conversion sequence is a user-defined conversion sequence. If multiple constructors are viable but none is better than the others, the implicit conversion sequence is the ambiguous conversion sequence. User-defined conversions are allowed for conversion of the initializer list elements to the constructor parameter types except as noted in 13.3.3.1. [Example:

```cpp
struct A {
    A(std::initializer_list<int>);
};
void f(A);
f( {'a', 'b'} ); // OK: f(A(std::initializer_list<int>)) user-defined conversion
```

```cpp
struct B {
    B(int, double);
};
void g(B);
g( {'a', 'b'} ); // OK: g(B(int, double)) user-defined conversion
g( {1.0, 1.0} ); // error: narrowing
```

```cpp
void f(B);
f( {'a', 'b'} ); // error: ambiguous f(A) or f(B)
```

```cpp
struct C {
    C(std::string);
};
void h(C);
h("foo"); // OK: h(C(std::string("foo")))
```

```cpp
struct D {
    C(A, C);
};
void i(D);
i({ {1,2}, {"bar"} }); // OK: i(D(A(std::initializer_list<int>{1,2}),C(std::string("bar"))))
```

— end example]  

4 Otherwise, if the parameter has an aggregate type which can be initialized from the initializer list according to the rules for aggregate initialization (8.5.1), the implicit conversion sequence is a user-defined conversion sequence. [Example:

```cpp
struct A {
    int m1;
    double m2;
};

void f(A);
f( {'a', 'b'} ); // OK: f(A(int,double)) user-defined conversion
f( {1.0} ); // error: narrowing
```

— end example]

5 Otherwise, if the parameter is a reference, see 13.3.3.1.4. [Note: The rules in this section will apply for initializing the underlying temporary for the reference. — end note] [Example:

```cpp
struct A {
    int m1;
    double m2;
};
```
void f(const A&);
f( {'a', 'b'} );       // OK: f(A(int,double)) user-defined conversion
f( {1.0} );           // error: narrowing

void g(const double &);
g({1});                // same conversion as int to double

— end example ]

6 Otherwise, if the parameter type is not a class:
   — if the initializer list has one element, the implicit conversion sequence is the one required to convert
     the element to the parameter type; [ Example:
     void f(int);
f( {'a'} );          // OK: same conversion as char to int
f( {1.0} );         // error: narrowing
— end example ]
   — if the initializer list has no elements, the implicit conversion sequence is the identity conversion.
     [ Example:
     void f(int);
f( { } );          // OK: identity conversion
— end example ]

7 In all cases other than those enumerated above, no conversion is possible.

13.3.3.2 Ranking implicit conversion sequences

1 13.3.3.2 defines a partial ordering of implicit conversion sequences based on the relationships better conversion
   sequence and better conversion. If an implicit conversion sequence S1 is defined by these rules to be a better
   conversion sequence than S2, then it is also the case that S2 is a worse conversion sequence than S1. If
   conversion sequence S1 is neither better than nor worse than conversion sequence S2, S1 and S2 are said to
   be indistinguishable conversion sequences.
2 When comparing the basic forms of implicit conversion sequences (as defined in 13.3.3.1)
   — a standard conversion sequence (13.3.3.1.1) is a better conversion sequence than a user-defined con-
     version 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 conver-
     sion sequence (13.3.3.1.3).
3 Two implicit conversion sequences of the same form are indistinguishable conversion sequences unless one of
   the following rules applies:
   — Standard conversion sequence S1 is a better conversion sequence than standard conversion sequence
     S2 if
     — S1 is a proper subsequence of S2 (comparing the conversion sequences in the canonical form
       defined by 13.3.3.1.1, excluding any Lvalue Transformation; the identity conversion sequence is
       considered to be a subsequence of any non-identity conversion sequence) or, if not that,
— the rank of \( S_1 \) is better than the rank of \( S_2 \), or \( S_1 \) and \( S_2 \) have the same rank and are distinguishable by the rules in the paragraph below, or, if not that,

— \( S_1 \) and \( S_2 \) differ only in their qualification conversion and yield similar types \( T_1 \) and \( T_2 \) (4.4), respectively, and the cv-qualification signature of type \( T_1 \) is a proper subset of the cv-qualification signature of type \( T_2 \). [Example:

\[
\begin{align*}
  \text{int } f(\text{const int } *) ; \\
  \text{int } f(\text{int } *) ; \\
  \text{int } i ; \\
  \text{int } j = f(i) ; & \text{ } / \text{ calls } f(\text{int } *) \\
\end{align*}
\]
— end example] or, if not that,

— \( S_1 \) and \( S_2 \) are reference bindings (8.5.3) and neither refers to an implicit object parameter of a non-static member function declared without a \textit{ref-qualifier}, and \( S_1 \) binds an rvalue reference to an rvalue and \( S_2 \) binds an lvalue reference.

[Example:

\[
\begin{align*}
  \text{int } i ; \\
  \text{int } f1() ; \\
  \text{int}&& f2() ; \\
  \text{int } g(\text{const int}&) ; \\
  \text{int } g(\text{const int}&&) ; \\
  \text{int } j = g(i) ; & \text{ } / \text{ calls } g(\text{const int}&) \\
  \text{int } k = g(f1()) ; & \text{ } / \text{ calls } g(\text{const int}&&) \\
  \text{int } l = g(f2()) ; & \text{ } / \text{ calls } g(\text{const int}&&) \\
\end{align*}
\]

struct \( A \) {
  \A& operator\ll\ll(\A&\&, \text{char}) ; \\
  \A() \ll \ll 1 ; & \text{ } / \text{ calls } A::\ll\ll(\text{int}) \\
  \A() \ll \ll 'c' ; & \text{ } / \text{ calls } \ll\ll(\A&\&, \text{char}) \\
  \A a ; \\
  a \ll \ll 1 ; & \text{ } / \text{ calls } A::\ll\ll(\text{int}) \\
  a \ll \ll 'c' ; & \text{ } / \text{ calls } A::\ll\ll(\text{int}) \\
  A().p() ; & \text{ } / \text{ calls } A::p()&& \\
  a.p() ; & \text{ } / \text{ calls } A::p()& \\
\};
\]
— end example] or, if not that,

— \( S_1 \) and \( S_2 \) 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 \( S_2 \) refers is more cv-qualified than the type to which the reference initialized by \( S_1 \) refers. [Example:

\[
\begin{align*}
  \text{int } f(\text{const int } &) ; \\
  \text{int } f(\text{int } &) ; \\
  \text{int } g(\text{const int } &) ; \\
  \text{int } g(\text{int}) ; \\
  \\
  \text{int } i ; \\
  \text{int } j = f(i) ; & \text{ } / \text{ calls } f(\text{int } &) \\
  \text{int } k = g(i) ; & \text{ } / \text{ ambiguous } \\
\end{align*}
\]
struct X {
  void f() const;
  void f();
};
void g(const X& a, X b) {
  a.f(); // calls X::f() const
  b.f(); // calls X::f()
}

— end example —

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 —

List-initialization sequence L1 is a better conversion sequence than list-initialization sequence L2 if L1 converts to std::initializer_list<X> for some X and L2 does not.

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 does not convert a pointer, a pointer to member, or std::nullptr_t to bool is better than one that does.

— 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 *); // calls 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&.

§ 13.3.3.2
conversion of \texttt{A::*} to \texttt{B::*} is better than conversion of \texttt{A::*} to \texttt{C::*},
— conversion of \texttt{C} to \texttt{B} is better than conversion of \texttt{C} to \texttt{A},
— conversion of \texttt{B*} to \texttt{A*} is better than conversion of \texttt{C*} to \texttt{A*},
— binding of an expression of type \texttt{B} to a reference of type \texttt{A&} is better than binding an expression of type \texttt{C} to a reference of type \texttt{A&},
— conversion of \texttt{B::*} to \texttt{C::*} is better than conversion of \texttt{A::*} to \texttt{C::*}, and
— conversion of \texttt{B} to \texttt{A} is better than conversion of \texttt{C} to \texttt{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 \texttt{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 \texttt{template-id} is then examined to see whether it identifies a single function template specialization. If it does, the \texttt{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 \texttt{F1} is eliminated if
the set contains a second function template specialization whose function template is more specialized than the function template of \( F_1 \) according to the partial ordering rules of 14.5.6.2. After such eliminations, if any, there shall remain exactly one selected function.

5  

[Example:]

```c
int f(double);
int f(int);
int (*pfd)(double) = &f; // selects f(double)
int (*pfi)(int) = &f; // selects f(int)
int (*pfe)(...) = &f; // error: type mismatch
int (&rfi)(int) = f; // selects f(int)
int (&rfd)(double) = f; // selects f(double)
void g() {
    (int (*)(int))&f; // cast expression as selector
}
```

The initialization of \( pfe \) is ill-formed because no \( f() \) with type \( \text{int}(...) \) has been declared, and not because of any ambiguity. For another example,

```c
struct X {
    int f(int);
    static int f(long);
};
int (X::*p1)(int) = &X::f; // OK
int (*p2)(int) = &X::f; // error: mismatch
int (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]

6  

[Note: if \( f() \) and \( g() \) are both overloaded functions, the cross product of possibilities must be considered to resolve \( f(&g) \), or the equivalent expression \( f(g) \). — end note]

7  

[Note: there are no standard conversions (Clause 4) of one pointer-to-function type into another. In particular, even if \( B \) is a public base of \( D \), we have]

```c
D* f();  // error
B* (*p1)() = &f;  // error
void g(D*);
void (*p2)(B*) = &g;  // error
```

— end note]

### 13.5 Overloaded operators

A function declaration having one of the following operator-function-ids as its name declares an operator function. A function template declaration having one of the following operator-function-ids as its name declares an operator function template. A specialization of an operator function template is also an operator function. An operator function is said to implement the operator named in its operator-function-id.
operator-function-id:
    operator operator

operator: one of
    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 3.7.4. The attributes and restrictions found in the rest of this subclause do not apply to them unless explicitly stated in 3.7.4.

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.

8 An operator function cannot have default arguments (8.3.6), except where explicitly stated below. Operator functions cannot have more or fewer parameters than the number required for the corresponding operator, as described in the rest of this subclause.
9 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

1 A prefix unary operator shall be implemented by a non-static member function (9.3) with no parameters or a non-member function with one parameter. Thus, for any prefix unary operator $\&$, $\&x$ can be interpreted as either $x.$operator$()$ or operator$()$($x$). If both forms of the operator function have been declared, the rules in 13.1.2 determine which, if any, interpretation is used. See 13.5.7 for an explanation of the postfix unary operators ++ and --.

2 The unary and binary forms of the same operator are considered to have the same name. [Note: consequently, a unary operator can hide a binary operator from an enclosing scope, and vice versa. — end note]

13.5.2 Binary operators

1 A binary operator shall be implemented either by a non-static member function (9.3) with one parameter or by a non-member function with two parameters. Thus, for any binary operator $x\&y$ can be interpreted as either $x.$operator$()$($y$) or operator$()$($x$, $y$). If both forms of the operator function have been declared, the rules in 13.1.2 determine which, if any, interpretation is used.

13.5.3 Assignment

1 An assignment operator shall be implemented by a non-static member function with exactly one parameter. Because a copy assignment operator operator$=$ is implicitly declared for a class if not declared by the user (12.8), a base class assignment operator is always hidden by the copy assignment operator of the derived class.

2 Any assignment operator, even the copy and move assignment operators, can be virtual. [Note: for a derived class D with a base class B for which a virtual copy/move assignment has been declared, the copy/move assignment operator in D does not override B’s virtual copy/move assignment operator. [Example:

```c
struct B {
    virtual int operator=(int);
    virtual B& operator=(const B&);
};
struct D : B {
    virtual int operator=(int);
    virtual D& operator=(const B&);
};
D dobj1;
D dobj2;
B* bptr = &dobj1;
void f() {
    bptr->operator=(99); // calls D::operator=(int)
    *bptr = 99; // ditto
    bptr->operator=(dobj2); // calls D::operator=(const B&)
    *bptr = dobj2; // ditto
    dobj1 = dobj2; // calls implicitly-declared
    // D::operator=(const D&)
}
```

§ 13.5.3
13.5.4 Function call

1 $\text{operator}()$ shall be a non-static member function with an arbitrary number of parameters. It can have default arguments. It implements the function call syntax

$$\text{postfix-expression ( expression-list_{opt} )}$$

where the $\text{postfix-expression}$ evaluates to a class object and the possibly empty $\text{expression-list}$ matches the parameter list of an $\text{operator}()$ member function of the class. Thus, a call $x(arg1, \ldots)$ is interpreted as $x.\text{operator}(\arg1, \ldots)$ for a class object $x$ of type $T$ if $T::\text{operator}()$ exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3.3).

13.5.5 Subscripting

1 $\text{operator}[\ ]$ shall be a non-static member function with exactly one parameter. It implements the subscripting syntax

$$\text{postfix-expression [ expression ]}$$

Thus, a subscripting expression $x[y]$ is interpreted as $x.\text{operator}()[y]$ for a class object $x$ of type $T$ if $T::\text{operator}()$ exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3.3).

13.5.6 Class member access

1 $\text{operator->}$ shall be a non-static member function taking no parameters. It implements the class member access syntax that uses $\text{->}$.

$$\text{postfix-expression -> template_{opt} id-expression}$$
$$\text{postfix-expression -> pseudo-destructor-name}$$

An expression $x->m$ is interpreted as $(x.\text{operator}->())->m$ for a class object $x$ of type $T$ if $T::\text{operator}->()$ exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3).}

13.5.7 Increment and decrement

1 The user-defined function called $\text{operator++}$ implements the prefix and postfix $++$ operator. If this function is a member function with no parameters, or a non-member function with one parameter of class or enumeration type, it defines the prefix increment operator $++$ for objects of that type. If the function is a member function with one parameter (which shall be of type $\text{int}$) or a non-member function with two parameters (the second of which shall be of type $\text{int}$), it defines the postfix increment operator $++$ for objects of that type. When the postfix increment is called as a result of using the $++$ operator, the $\text{int}$ argument will have value zero.$^{134}$

[Example:

```cpp
struct X {
    X& operator++();             // prefix ++a
    X operator++(int);           // postfix a++
};

struct Y { }
    Y& operator++(Y&);            // prefix ++b
    Y operator++(Y&, int);       // postfix b++
```

$^{134}$ Calling $\text{operator++}$ explicitly, as in expressions like $a.\text{operator++}(2)$, has no special properties: The argument to $\text{operator++}$ is 2.

§ 13.5.7
```c
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

2 The prefix and postfix decrement operators -- are handled analogously.

13.5.8 User-defined literals

```
literal-operator-id:
operator "" identifier
```

1 The identifier in a literal-operator-id is called a literal suffix identifier.

2 A declaration whose declarator-id is a literal-operator-id shall be a declaration of a namespace-scope function or function template (it could be a friend function (11.4)), an explicit instantiation or specialization of a function template, or a using-declaration (7.3.3). A function declared with a literal-operator-id is a literal operator. A function template declared with a literal-operator-id is a literal operator template.

3 The declaration of a literal operator shall have a parameter-declaration-clause equivalent to one of the following:

```
const char*
unsigned long long int
long double
char
wchar_t
char16_t
char32_t
const char*, std::size_t
const wchar_t*, std::size_t
const char16_t*, std::size_t
const char32_t*, std::size_t
```

4 A raw literal operator is a literal operator with a single parameter whose type is `const char*`.

5 The declaration of a literal operator template shall have an empty parameter-declaration-clause and its template-parameter-list shall have a single template-parameter that is a non-type template parameter pack (14.5.3) with element type `char`.

6 Literal operators and literal operator templates shall not have C language linkage.

7 [Note: literal operators and literal operator templates are usually invoked implicitly through user-defined literals (2.14.8). However, except for the constraints described above, they are ordinary namespace-scope functions and function templates. In particular, they are looked up like ordinary functions and function templates and they follow the same overload resolution rules. Also, they can be declared inline or constexpr, they may have internal or external linkage, they can be called explicitly, their addresses can be taken, etc. — end note]
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

\[ VQ \text{ & operator}++(VQ \text{ &}); \]
\[ T \text{ & operator}++(VQ \text{ &}, \text{ int}); \]

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

\[ VQ \text{ & operator}--(VQ \text{ &}); \]
\[ T \text{ & operator}--(VQ \text{ &}, \text{ int}); \]

For every pair \((T, VQ)\), where \(T\) is a cv-qualified or cv-unqualified object type, and \(VQ\) is either volatile or empty, there exist candidate operator functions of the form

\[ T \ast \text{ & operator}++(T \ast \text{ &}); \]
\[ T \ast \text{ & operator}--(T \ast \text{ &}); \]
\[ T \ast \text{ & operator}++(T \ast \text{ &}, \text{ int}); \]
\[ T \ast \text{ & operator}--(T \ast \text{ &}, \text{ int}); \]
For every cv-qualified or cv-unqualified object type \( T \), there exist candidate operator functions of the form

\[
T& \quad \text{operator*}(T*)
\]

For every function type \( T \) that does not have cv-qualifiers or a ref-qualifier, there exist candidate operator functions of the form

\[
T& \quad \text{operator*}(T*)
\]

For every type \( T \) there exist candidate operator functions of the form

\[
T* \quad \text{operator+}(T*)
\]

For every promoted arithmetic type \( T \), there exist candidate operator functions of the form

\[
T \quad \text{operator+}(T);
\]
\[
T \quad \text{operator-}(T);
\]

For every promoted integral type \( T \), there exist candidate operator functions of the form

\[
T \quad \text{operator}~(T);
\]

For every quintuple \((C1, C2, T, CV1, CV2)\), where \( C2 \) is a class type, \( C1 \) is the same type as \( C2 \) or is a derived class of \( C2 \), \( T \) is an object type or a function type, and \( CV1 \) and \( CV2 \) are cv-qualifier-seqs, there exist candidate operator functions of the form

\[
CV12 \quad T& \quad \text{operator->*}(CV1 C1*, \ CV2 T C2::*);
\]

where \( CV12 \) is the union of \( CV1 \) and \( CV2 \).

For every pair of promoted arithmetic types \( L \) and \( R \), there exist candidate operator functions of the form

\[
LR \quad \text{operator*}(L, R);
\]
\[
LR \quad \text{operator/}(L, R);
\]
\[
LR \quad \text{operator+}(L, R);
\]
\[
LR \quad \text{operator-}(L, R);
\]
\[
bool \quad \text{operator<}(L, R);
\]
\[
bool \quad \text{operator>(L, R)};
\]
\[
bool \quad \text{operator<=}(L, R);
\]
\[
bool \quad \text{operator>=}(L, R);
\]
\[
bool \quad \text{operator==}(L, R);
\]
\[
bool \quad \text{operator!=}(L, R);
\]

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

\[
T* \quad \text{operator+}(T*, \ std::ptrdiff_t);
\]
\[
T& \quad \text{operator[]}(T*, \ std::ptrdiff_t);
\]
\[
T* \quad \text{operator-}(T*, \ std::ptrdiff_t);
\]
\[
T* \quad \text{operator+}(\text{std::ptrdiff_t}, \ T*);
\]
\[
T& \quad \text{operator[]}(\text{std::ptrdiff_t}, \ T*);
\]

For every \( T \), where \( T \) is a pointer to object type, there exist candidate operator functions of the form

\[
\text{std::ptrdiff_t} \quad \text{operator-}(T, T);
\]

For every \( T \), where \( T \) is an enumeration type, a pointer type, or \texttt{std::nullptr_t}, there exist candidate operator functions of the form

\[
\text{§ 13.6 ~ 312}
\]
bool operator<(T, T);
bool operator>(T, T);
bool operator<=(T, T);
bool operator>=(T, T);
bool operator==(T, T);
bool operator!=(T, T);

16 For every pointer to member type \( T \) there exist candidate operator functions of the form

bool operator==(T, T);
bool operator!=(T, T);

17 For every pair of promoted integral types \( L \) and \( R \), there exist candidate operator functions of the form

\[
\begin{align*}
LR & \quad \text{operator}\%(L, R); \\
LR & \quad \text{operator}\&(L, R); \\
LR & \quad \text{operator}\&(L, R); \\
L & \quad \text{operator}<<(L, R); \\
L & \quad \text{operator}>>(L, R);
\end{align*}
\]

where \( LR \) is the result of the usual arithmetic conversions between types \( L \) and \( R \).

18 For every triple \( (L, VQ, R) \), where \( L \) is an arithmetic type, \( VQ \) is either \texttt{volatile} or empty, and \( R \) is a promoted arithmetic type, there exist candidate operator functions of the form

\[
\begin{align*}
VQ \ & L \ & \text{operator}=((VQ \ & L \ & , \ R); \\
VQ \ & L \ & \text{operator}**(VQ \ & L \ & , \ R); \\
VQ \ & L \ & \text{operator}*/=(VQ \ & L \ & , \ R); \\
VQ \ & L \ & \text{operator}+=(VQ \ & L \ & , \ R); \\
VQ \ & L \ & \text{operator}-=((VQ \ & L \ & , \ R);
\end{align*}
\]

19 For every pair \( (T, VQ) \), where \( T \) is any type and \( VQ \) is either \texttt{volatile} or empty, there exist candidate operator functions of the form

\[
T \ & VQ \ & \text{operator}=((T \ *VQ \ & , \ T *)
\]

20 For every pair \( (T, VQ) \), where \( T \) is an enumeration or pointer to member type and \( VQ \) is either \texttt{volatile} or empty, there exist candidate operator functions of the form

\[
VQ \ & T \ & \text{operator}((VQ \ & T \ & , \ T)
\]

21 For every pair \( (T, VQ) \), where \( T \) is a cv-qualified or cv-unqualified object type and \( VQ \) is either \texttt{volatile} or empty, there exist candidate operator functions of the form

\[
T \ & VQ \ & \text{operator}++((T \ & VQ \ & , \ std::ptrdiff_t)); \\
T \ & VQ \ & \text{operator}=(T \ & VQ \ & , \ std::ptrdiff_t);
\]

22 For every triple \( (L, VQ, R) \), where \( L \) is an integral type, \( VQ \) is either \texttt{volatile} or empty, and \( R \) is a promoted integral type, there exist candidate operator functions of the form

\[
\begin{align*}
VQ \ & L \ & \text{operator}%(VQ \ & L \ & , \ R ); \\
VQ \ & L \ & \text{operator}<<(VQ \ & L \ & , \ R); \\
VQ \ & L \ & \text{operator}+=(VQ \ & L \ & , \ R); \\
VQ \ & L \ & \text{operator}>>=(VQ \ & L \ & , \ R); \\
VQ \ & L \ & \text{operator}^=(VQ \ & L \ & , \ R); \\
VQ \ & L \ & \text{operator}!=(VQ \ & L \ & , \ R);
\end{align*}
\]
There also exist candidate operator functions of the form

```cpp
bool operator!(bool);
bool operator&&(bool, bool);
bool operator||(bool, bool);
```

For every pair of promoted arithmetic types \( L \) and \( R \), there exist candidate operator functions of the form

```cpp
LR operator?(bool, L, R);
```

where \( LR \) is the result of the usual arithmetic conversions between types \( L \) and \( R \). \[ Note: as with all these descriptions of candidate functions, this declaration serves only to describe the built-in operator for purposes of overload resolution. The operator “?” cannot be overloaded. — end note \]

For every type \( T \), where \( T \) is a pointer, pointer-to-member, or scoped enumeration type, there exist candidate operator functions of the form

```cpp
T operator?(bool, T, T);
```
14 Templates

1 A template defines a family of classes or functions or an alias for a family of types.

   template-declaration:
   template < template-parameter-list > declaration
   template-parameter-list:
   template-parameter
   template-parameter-list , template-parameter

   [Note: The > token following the template-parameter-list of a template-declaration may be the product of
   replacing a >> token by two consecutive > tokens (14.2). — end note]

   The declaration in a template-declaration shall
   — 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 dec-
   laration 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 func-
   tion 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, variable, enumeration,
   enumerator, namespace, or type in the same scope (3.3), except as specified in (14.5.5). Except that a
   function template can be overloaded either by (non-template) functions 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.
14.1 Template parameters

The syntax for template-parameters is:

```
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
```

[Note: The > token following the template-parameter-list of a type-parameter may be the product of replacing a >> token by two consecutive > tokens (14.2). — end note]

There is no semantic difference between class and typename in a template-parameter. typename followed by an unqualified-id names a template type parameter. typename followed by a qualified-id denotes the type in a non-type parameter-declaration. A storage class shall not be specified in a template-parameter declaration. [Note: a template parameter may be a class template. For example,

```c
template<class T> class myarray { /* ... */ };
```

```c
template<class K, class V, template<class T> class C = myarray>
class Map {
  C<K> key;
  C<V> value;
};
```

— end note]

A type-parameter whose identifier does not follow an ellipsis defines its identifier to be a typedef-name (if declared with class or typename) or template-name (if declared with template) in the scope of the template declaration. [Note: because of the name lookup rules, a template-parameter that could be interpreted as either a non-type template-parameter or a type-parameter (because its identifier is the name of an already existing class) is taken as a type-parameter. For example,

```c
class T { /* ... */ };
int i;
```

```c
template<class T, T i> void f(T t) {
  T t1 = i;    // template-parameters T and i
  ::T t2 = ::i;    // global namespace members T and i
}
```

Here, the template f has a type-parameter called T, rather than an unnamed non-type template-parameter of class T. — end note]

A non-type template-parameter shall have one of the following (optionally cv-qualified) types:

— integral or enumeration type,
— pointer to object or pointer to function,

135) Since template template-parameters and template template-arguments are treated as types for descriptive purposes, the terms non-type parameter and non-type argument are used to refer to non-type, non-template parameters and arguments.
— lvalue reference to object or lvalue 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 a prvalue. It shall not be assigned to or in any other way have its value changed. A non-type non-reference template-parameter cannot have its address taken. When a non-type non-reference template-parameter is used as an initializer for a reference, a temporary is always used. [Example:

```cpp
template<const X& x, int i> void f() {
    i++;
    // error: change of template-parameter value

    &x;
    // OK
    &i;
    // error: address of non-reference template-parameter

    int& ri = i;
    // error: non-const reference bound to temporary
    const int& crei = i;
    // OK: const reference bound to temporary
}
```
— end example]

7 A non-type template-parameter shall not be declared to have floating point, class, or void type. [Example:

```cpp
template<double d> class X;  // error
template<double* pd> class Y;  // OK
template<double& rd> class Z;  // OK
```
— end example]

8 A non-type template-parameter of type “array of T” or “function returning T” is adjusted to be of type “pointer to T” or “pointer to function returning T”, respectively. [Example:

```cpp
template<int *a> struct R { /* ... */ };  // OK
template<int b[5]> struct S { /* ... */ };  // OK due to parameter adjustment
int p;
R<&p> w;  // OK
S<&p> x;  // OK due to implicit argument conversion
int v[5];
R<&v> y;  // OK due to both adjustment and conversion
S<&v> z;  // OK due to both adjustment and conversion
```
— end example]

9 A default template-argument is a template-argument (14.3) specified after = in a template-parameter. A default template-argument may be specified for any kind of template-parameter (type, non-type, template) that is not a template parameter pack (14.5.3). A default template-argument may be specified in a template declaration. A default template-argument shall not be specified in the template-parameter-lists of the definition of a member of a class template that appears outside of the member’s class. A default template-argument shall not be specified in a friend class template declaration. If a friend function template declaration specifies a default template-argument, that declaration shall be a definition and shall be the only declaration of the function template in the translation unit.
10 The set of default *template-arguments* available for use with a template declaration or definition is obtained by merging the default arguments from the definition (if in scope) and all declarations in scope in the same way default function arguments are (8.3.6).  

```cpp
template<class T1, class T2 = int> class A;

template<class T1 = int, class T2> class A;
```
is equivalent to

```cpp
template<class T1 = int, class T2 = int> class A;
```

— end example

11 If a *template-parameter* of a class template has a default *template-argument*, each subsequent *template-parameter* shall either have a default *template-argument* supplied or be a template parameter pack. If a *template-parameter* of a class template is a template parameter pack, it shall be the last *template-parameter*.  

[Note: These are not requirements for function templates because template arguments might be deduced (14.8.2).]  

```cpp
template<class T1 = int, class T2> class B;  // error
```

— end example — end note

12 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

13 When parsing a default *template-argument* for a non-type *template-parameter*, the first non-nested `>` is taken as the end of the *template-parameter-list* rather than a greater-than operator.  

[Example:]

```cpp
template<int i = 3 > int i = 3 >
class X { /*...*/ };  // syntax error

template<int i = (3 > 4) > int i = (3 > 4)
class Y { /*...*/ };  // OK
```

— end example

14 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>
}
```
15 If a template-parameter is a type-parameter with an ellipsis prior to its optional identifier or is a parameter-declaration that declares a parameter pack (8.3.5), then the template-parameter is a template parameter pack (14.5.3).  [Example:

```cpp
template <class... Types> class Tuple;  // Types is a template type parameter pack
template <class T, int... Dims> struct multi_array; // Dims is a non-type template parameter pack
```

— end example]}

14.2 Names of template specializations

1 A template specialization (14.7) can be referred to by a template-id:

- simple-template-id:
  - template-name < template-argument-list_opt >
- template-id:
  - simple-template-id
  - operator-function-id < template-argument-list_opt >
  - literal-operator-id < template-argument-list_opt >

- template-name:
  - identifier
- template-argument-list:
  - template-argument ...opt
  - template-argument-list , template-argument ...opt
- template-argument:
  - constant-expression
  - type-id
  - id-expression

[Note: the name lookup rules (3.4) are used to associate the use of a name with a template declaration; that is, to identify a name as a template-name. — end note]

2 For a template-name to be explicitly qualified by the template arguments, the name must be known to refer to a template.

3 After name lookup (3.4) finds that a name is a template-name or that an operator-function-id or a literal-operator-id refers to a set of overloaded functions any member of which is a function template if this is followed by a <, the < is always taken as the delimiter of a template-argument-list and never as the less-than operator. When parsing a template-argument-list, the first non-nested >\textsuperscript{136} is taken as the ending delimiter rather than a greater-than operator. Similarly, the first non-nested >>= is treated as two consecutive but distinct > tokens, the first of which is taken as the end of the template-argument-list and completes the template-id. [Note: The second > token produced by this replacement rule may terminate an enclosing template-id construct or it may be part of a different construct (e.g. a cast). — end note]  [Example:

```cpp
template<int i> class X { /* ... */ };
X< 1>2 > x1;  // syntax error
X<(1>2)> x2;  // OK
```

136] 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.
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

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 (14.5.3), 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);
```
```cpp
Array<dcomplex> v3(40);

void bar() {
    v1[3] = 7;
    v2[3] = v3.elem(4) = dcomplex(7,8);
}

— end example ]

2 In a template-argument, an ambiguity between a type-id and an expression is resolved to a type-id, regardless of the form of the corresponding template-parameter. ¹³⁷ [Example:
```
template<class T> void f();
template<int I> void f();

void g() {
    f<int>()(); // int() is a type-id: call the first f()
}

— end example ]

3 The name of a template-argument shall be accessible at the point where it is used as a template-argument.  [Note: if the name of the template-argument is accessible at the point where it is used as a template-argument, there is no further access restriction in the resulting instantiation where the corresponding template-parameter name is used. — end note] [Example:
```
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:
```
template <template <class TT> class T> class A {
    typename T<int>::S s;
};

template <class U> class B {
    private:
        struct S { /* ... */};
    };

    A<B> b; // ill-formed: A has no access to B::S
```

¹³⁷ There is no such ambiguity in a default template-argument because the form of the template-parameter determines the allowable forms of the template-argument.

§ 14.3
4 When template argument packs or default template-arguments are used, a template-argument list can be empty. In that case the empty <> brackets shall still be used as the template-argument-list. [Example:

    template<class T = char> class String;
    String** p; // OK: String<char>
    String* q; // syntax error
    template<class ... Elements> class Tuple;
    Tuple** t; // OK: Elements is empty
    Tuple* u; // syntax error

    — end example]

5 An explicit destructor call (12.4) for an object that has a type that is a class template specialization may explicitly specify the template-arguments. [Example:

    template<class T> struct A {
        ~A();
    }
    void f(A<int>** p, A<int>** q) {
        p->A<int>::~A(); // OK: destructor call
        q->A<int>::~A<int>(); // OK: destructor call
    }

    — end example]

6 If the use of a template-argument gives rise to an ill-formed construct in the instantiation of a template specialization, the program is ill-formed.

7 When the template in a template-id is an overloaded function template, both non-template functions in the overload set and function templates in the overload set for which the template-arguments do not match the template-parameters are ignored. If none of the function templates have matching template-parameters, the program is ill-formed.

8 A template-argument followed by an ellipsis is a pack expansion (14.5.3).

14.3.1 Template type arguments [temp.arg.type]

1 A template-argument for a template-parameter which is a type shall be a type-id.

2 [Example:

    template <class T> class X { }
    template <class T> void f(T t) { }
    struct { } unnamed_obj;

    void f() {
        struct A { }
        enum { e1 }
        typedef struct { } B;
        B b;
        X<A> x1; // OK
        X<A*> x2; // OK
        X<B> x3; // OK
        f(e1); // OK
        f(unnamed_obj); // OK
        f(b); // OK
    }

    § 14.3.1 322
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:

```cpp
template<class T> struct A {
  static T t;
};
typedef int function();
A<function> a; // ill-formed: would declare A<function>::t
// as a static member function
```

— end example]

### 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 (including a constant expression of literal class type that can be used as an integral constant expression as described in 5.19); 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 (2.14.5) does not satisfy the requirements of any of these categories and thus is not an acceptable template-argument. | Example:

```cpp
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:

```cpp
template<int* p> class X { }

int a[10];
struct S { int m; static int s; } s;
X<&a[2]> x3; // error: address of array element
```
X<&s.m> x4;       // error: address of non-static member
X<&s.s> x5;       // error: &S::s must be used
X<&S::s> x6;      // OK: address of static member

— end example — end note

[Note: Temporaries, unnamed lvalues, and named lvalues that do not have external linkage are not acceptable template-arguments when the corresponding template-parameter has reference type. [Example:

```cpp
template<const int& CRI> struct B { /* ... */ };

B<1> b2;        // error: temporary would be required for template argument
int c = 1;
B<c> b1;        // OK
```

— end example — end note

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 template-parameter of pointer type. However, both (int*)0 and 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 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 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 std::nullptr_t, the null member pointer conversion (4.11) is applied.

[Example:
template<const int* pci> struct X { /* ... */ };  
int ai[10];  
X<ai> x;  // array to pointer and qualification conversions

struct Y { /* ... */ };  
template<const Y& b> struct Z { /* ... */ };  
Y y;  
Z<y> z;  // no conversion, but note extra cv-qualification

template<int (&pa)[5]> struct W { /* ... */ };  
int b[5];  
W<b> w;  // no conversion

void f(char);  
void f(int);  

template<void (*pf)(int)> struct A { /* ... */ };  
A<&f> a;  // selects f(int)

— end example]

14.3.3 Template template arguments [temp.arg.template]

A template-argument for a template template-parameter shall be the name of a class template or 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:

```
template<class T> class A { /* ... */ };  
int x;  
};  
template<class T> class A<T*> { /* partial specialization */  
long x;  
};  
template<template<class U> class V> class V> class C {  
V<int> y;  
V<int*> z;  
};  
C<A> c;  // V<int> within C<A> uses the primary template,  
// so c.y.x has type int  
// V<int*> within C<A> uses the partial specialization,  
// so c.z.x has type long
```

— end example]

[Example:

```
template<class T> class A { /* ... */ };  
```
template<class T, class U = T> class B { /* ... */ };  

template<template<class> class P> class X { /* ... */ };  

X<A> xa; // OK  
X<B> xb; // ill-formed: default arguments for the parameters of a template argument are ignored  
X<C> xc; // ill-formed: a template parameter pack does not match a template parameter  

Y<A> ya; // OK  
Y<B> yb; // OK  
Y<C> yc; // OK  

— end example  

14.4 Type equivalence [temp.type]  

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;  

template <template <class, class...> class TT, class T1, class... Rest>  
struct eval<TT<T1, Rest...>> { };  

template <class T1> struct A;  
template <class T1, class T2> struct B;  
template <int N> struct C;  
template <class T1, int N> struct D;  
template <class T1, class T2, int N = 17> struct E;  

eval<A<int>> eA; // OK: matches partial specialization of eval  
eval<B<int, float>> eB; // OK: matches partial specialization of eval  
eval<C<int>> eC; // error: \(C\) does not match \(TT\) in partial specialization  
eval<D<int, 17>> eD; // error: \(D\) does not match \(TT\) in partial specialization  
eval<E<int, float>> eE; // error: \(E\) does not match \(TT\) in partial specialization  
```

— end example  

1 Two template-ids refer to the same class or function if  

— their template-names, operator-function-ids, or literal-operator-ids refer to the same template and  
— their corresponding type template-arguments are the same type and  
— their corresponding non-type template arguments of integral or enumeration type have identical values and  
— their corresponding non-type template-arguments of pointer type refer to the same 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.

```cpp
template<template<class> class TT> 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:

```cpp
template<class T1, class T2, int I> class A<T1, T2, I> { };
```

— error

```cpp
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:

§ 14.5.1
```cpp
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 in the template parameter list of the member. Each template parameter pack shall be expanded with an ellipsis in the template argument list. [Example:

```cpp
template<class T1, class T2> struct A {
    void f1();
    void f2();
};

template<class T2, class T1> void A<T2,T1>::f1() { } // OK
template<class T2, class T1> void A<T1,T2>::f2() { } // error

template<class ... Types> struct B {
    void f3();
    void f4();
};

template<class ... Types> void B<Types ...>::f3() { } // OK
template<class ... Types> void B<Types>::f4() { } // error
```

— 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 [temp.mem.func]

1. A member function of a class template may be defined outside of the class template definition in which it is declared. [Example:

```cpp
template<class T> class Array {
    T* v;
    int sz;
public:
    explicit Array(int);
    T& operator[](int);
    T& elem(int i) { return v[i]; }
};
```
declares three function templates. The subscript function might be defined like this:

```cpp
template<class T> T& Array<T>::operator[](int i) {
  if (i<0 || sz<=i) error("Array: range error");
  return v[i];
}
```

— end example

2 The template-arguments for a member function of a class template are determined by the template-arguments of the type of the object for which the member function is called. [Example: the template-argument for Array<T>::operator [] () will be determined by the Array to which the subscripting operation is applied.

```cpp
Array<int> v1(20);
Array<dcomplex> v2(30);

v1[3] = 7;  // Array<int>::operator[]()

v2[3] = dcomplex(7,8); // Array<dcomplex>::operator[]()
```

— end example

14.5.1.2 Member classes of class templates [temp.mem.class]

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,

```cpp
template<class T> struct A {
  class B;
};
A<int>::B* b1;  // OK: requires A to be defined but not A::B

template<class T> class A<T>::B { };  // OK: requires A::B to be defined
```

— end note

14.5.1.3 Static data members of class templates [temp.static]

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:

```cpp
template<class T> class X {
  static T s;
};
template<class T> T X<T>::s = 0;
```

— end example

2 An explicit specialization of a static data member declared as an array of unknown bound can have a different bound from its definition, if any. [Example:

```cpp
template <class T> struct A {
  static int i[];
};
template <class T> int A<T>::i[4];  // 4 elements

template <> int A<int>::i[] = {1};  // OK: 1 element
```
14.5.2 Member templates

A template can be declared within a class or class template; such a template is called a member template. A member template can be defined within or outside its class definition or class template definition. A member template of a class template that is defined outside of its class template definition shall be specified with the template-parameters of the class template followed by the template-parameters of the member template.

```cpp
template<class T> struct string {
    template<class T2> int compare(const T2&);  // ... */
    template<class T2> string(const string<T2>& s) {}
};
```

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.

```cpp
template <class T> struct A {
    void f(int);
    template <class T2> void f(T2);
};
```

A member function template shall not be virtual.

```cpp
template <class T> struct AA {
    template <class C> virtual void g(C);  // error
    virtual void f();  // OK
};
```

A specialization of a member function template does not override a virtual function from a base class.

```cpp
class B {
    virtual void f(int);
};
```
class D : public B {
    template <class T> void f(T); // does not override B::f(int)
    void f(int i) { f<>(i); } // overriding function that calls
    // the template instantiation
};

— end example

A specialization of a conversion function template is referenced in the same way as a non-template conversion function that converts to the same type. [Example:

```c
struct A {
    template <class T> operator T*();
};

template <class T> A::operator T*(){ return 0; }
template <> A::operator char*(){ return 0; } // specialization
template A::operator void*(); // explicit instantiation
```

```c
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

1 A `template parameter pack` is a template parameter that accepts zero or more template arguments. [Example:

```c
template<class ... Types> struct Tuple { }
```

```c
Tuple<> t0; // Types contains no arguments
Tuple<int> t1; // Types contains one argument: int
Tuple<int, float> t2; // Types contains two arguments: int and float
Tuple<> error; // error: 0 is not a type

— end example
```

2 A `function parameter pack` is a function parameter that accepts zero or more function arguments. [Example:
template<class ... Types> void f(Types ... args);

f();   // OK: args contains no arguments
f(1);  // OK: args contains one argument: int
f(2, 1.0);  // OK: args contains two arguments: int and double

— end example

A parameter pack is either a template parameter pack or a function parameter pack.

A pack expansion is a sequence of tokens that names one or more parameter packs, followed by an ellipsis. The sequence of tokens is called the pattern of the expansion; its syntax depends on the context in which the expansion occurs. Pack expansions can occur in the following contexts:

— In an 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 a dynamic-exception-specification (15.4); the pattern is a type-id.
— In an attribute-list (7.6.1); the pattern is an attribute.
— In a capture-list (5.1.2); the pattern is a capture.

[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, unsigned>::type T2;  // error: different number of arguments specified for Args1 and Args2

    template<class ... Args> void g(Args ... args) {

§ 14.5.3
f(const_cast<const Args*>(&args)...); // OK: “Args” and “args” are expanded
f(5 ...); // error: pattern does not contain any parameter packs
f(args); // error: parameter pack “args” is not expanded
f(h(args ...) + args ...); // OK: first “args” expanded within h, second “args” expanded within f
}

— end example] 6

The instantiation of an expansion produces a list \( E_1 \oplus E_2 \oplus \ldots \oplus E_N \), where \( N \) is the number of elements in the pack expansion parameters and \( \oplus \) is the syntactically-appropriate separator for the list. 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] 1

14.5.4 Friends [temp.friend] 1

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]
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. For explicit specializations the corresponding member is the member (if any) that has the same name, kind (type, function, class template, or function template), template parameters, and signature as the member of the class template instantiation that would otherwise have been generated. [Example:

```cpp
template<class T> struct A {
    struct B { };
    void f();
    struct D {
        void g();
    };
};
template<> struct A<int> {
    struct B { };
    int f();
    struct D {
        void g();
    };
};

class C {
    template<class T> friend struct A<T>::B; // grants friendship to A<int>::B even though
    // it is not a specialization of A<T>::B
    template<class T> friend void A<T>::f(); // does not grant friendship to A<int>::f()
    // because its return type does not match
```
template<class T> friend void A<T>::D::g(); // does not grant friendship to A<int>::D::g()
    // because A<int>::D is not a specialization of A<T>::D
};
— end example]

6 [Note: a friend declaration may first declare a member of an enclosing namespace scope (14.6.5). — end note]

7 A friend template shall not be declared in a local class.

8 Friend declarations shall not declare partial specializations. [Example:

    template<class T> class A { }
    class X {
      template<class T> friend class A<T*>; // error
    }
— end example]

9 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 [temp.class.spec]

1 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 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).

3 [Example:

    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 declara-
tions declare partial specializations of the primary template. — end example]

4 The template parameters are specified in the angle bracket enclosed list that immediately follows the keyword
template. For partial specializations, the template argument list is explicitly written immediately following
the class template name. For primary templates, this list is implicitly described by the template parameter
list. Specifically, the order of the template arguments is the sequence in which they appear in the template
parameter list. [Example: the template argument list for the primary template in the example above is <T1,
T2, I>. — end example] [Note: the template argument list shall not be specified in the primary template
declaration. For example,

    template<class T1, class T2, int I> class A<T1, T2, I> { }; // error

§ 14.5.5  335
5 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>
template<class T> template<class T2>
struct A<T>::C::B<T2*> { };
A<short>::C::B<int*> absip; // uses partial specialization
```
— end example]

6 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]

7 A non-type argument is non-specialized if it is the name of a non-type parameter. All other non-type arguments are specialized.

8 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:
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. 

— 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  [temp.class.spec.match]

1 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.

2 A partial specialization matches a given actual template argument list if the template arguments of the partial specialization can be deduced from the actual template argument list (14.8.2). [Example:

A<int, int, 1> a1;  // uses #1
A<int, int*, 1> a2;  // uses #2, T is int, I is 1
A<int, char*, 5> a3;  // uses #4, T is char
A<int, char*, 1> a4;  // uses #5, T1 is int, T2 is char, I is 1
A<int*, int*, 2> a5;  // ambiguous: matches #3 and #5

— end example ]

3 A non-type template argument can also be deduced from the value of an actual template argument of a non-type parameter of the primary template. [Example: the declaration of a2 above. — end example ]

4 In a type name that refers to a class template specialization, (e.g., A<int, int, 1>) the argument list shall match the template parameter list of the primary template. The template arguments of a specialization are deduced from the arguments of the primary template.

138) There is no way in which they could be used.
14.5.5.2 Partial ordering of class template specializations [temp.class.order]

For two class template partial specializations, the first is at least as specialized as the second if, given the following rewrite to two function templates, the first function template is at least as specialized as the second according to the ordering rules for function templates (14.5.6.2):

— the first function template has the same template parameters as the first partial specialization and has a single function parameter whose type is a class template specialization with the template arguments of the first partial specialization, and

— the second function template has the same template parameters as the second partial specialization and has a single function parameter whose type is a class template specialization with the template arguments of the second partial specialization.

1 [Example:

```c
template<int I, int J, class T> class X { }
template<int I, int J> class X<I, J, int> { }; // #1
template<int I> class X<I, I, int> { }; // #2

template<int I, int J> void f(X<I, J, int>); // A
template<int I> void f(X<I, I, int>); // B
```

The partial specialization #2 is more specialized than the partial specialization #1 because the function template B is more specialized than the function template A according to the ordering rules for function templates. — end example]}

14.5.5.3 Members of class template specializations [temp.class.spec.mfunc]

1 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:

```c
// primary template
template<class T, int I> struct A {
   void f();
};

template<class T, int I> void A<T,I>::f() { }

// class template partial specialization
template<class T> struct A<T,2> {
   void f();
   void g();
   void h();
};

// member of class template partial specialization
template<class T> void A<T,2>::g() { }
```
// explicit specialization
template<> void A<char,2>::h() { }

int main() {
    A<char,0> a0;
    A<char,2> a2;
    a0.f(); // OK, uses definition of primary template’s member
    a2.g(); // OK, uses definition of
             // partial specialization’s member
    a2.h(); // OK, uses definition of
             // explicit specialization’s member
    a2.f(); // ill-formed, no definition of f for A<T,2>
             // the primary template is not used here
}

— end example

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:

    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:

    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 special-
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*)
}
```

```c
// 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
```
```c
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:

```c
template <int I, int J> void f(A<I+J>); // #1
```
```c
template <int K, int L> void f(A<K+L>); // same as #1
```

—139) 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.

§ 14.5.6.1 340
— end example] Two expressions involving template parameters that are not equivalent are functionally equivalent if, for any given set of template arguments, the evaluation of the expression results in the same value.

6 Two function templates are equivalent if they are declared in the same scope, have the same name, have identical template parameter lists, and have return types and parameter lists that are equivalent using the rules described above to compare expressions involving template parameters. Two function templates are functionally equivalent if they are equivalent except that one or more expressions that involve template parameters in the return types and parameter lists are functionally equivalent using the rules described above to compare expressions involving template parameters. If a program contains declarations of function templates that are functionally equivalent but not equivalent, the program is ill-formed; no diagnostic is required.

7 [Note: This rule guarantees that equivalent declarations will be linked with one another, while not requiring implementations to use heroic efforts to guarantee that functionally equivalent declarations will be treated as distinct. For example, the last two declarations are functionally equivalent and would cause a program to be ill-formed:

// Guaranteed to be the same
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+10>);

// Guaranteed to be different
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+11>);

// Ill-formed, no diagnostic required
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+1+2+3+4>);
— end note]

14.5.6.2 Partial ordering of function templates

If a function template is overloaded, the use of a function template specialization might be ambiguous because template argument deduction (14.8.2) may associate the function template specialization with more than one function template declaration. Partial ordering of overloaded function template declarations is used in the following contexts to select the function template to which a function template specialization refers:

— during overload resolution for a call to a function template specialization (13.3.3);
— when the address of a function template specialization is taken;
— when a placement operator delete that is a function template specialization is selected to match a placement operator new (3.7.4.2, 5.3.4);
— when a friend function declaration (14.5.4), an explicit instantiation (14.7.2) or an explicit specialization (14.7.3) refers to a function template specialization.

Partial ordering selects which of two function templates is more specialized than the other by transforming each template in turn (see next paragraph) and performing template argument deduction using the function 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.

§ 14.5.6.2
To produce the transformed template, for each type, non-type, or template template parameter (including template parameter packs (14.5.3) thereof) synthesize a unique type, value, or class template respectively and substitute it for each occurrence of that parameter in the function type of the template.

Using the transformed function template’s function parameter list, or in the case of a conversion function its transformed return type, perform type deduction against the function parameter list (or return type) of the other function. The mechanism for performing these deductions is given in 14.8.2.4.

[Example:

```cpp
template<class T> struct A { A(); };  
template<class T> void f(T);  
template<class T> void f(T*);  
template<class T> void f(const T*);  

template<class T> void g(T);  
template<class T> void g(T&);  

template<class T> void h(const T&);  
template<class T> void h(A<T>&);  

void m() {  
    const int *p;  
    f(p);  // f(const T*) is more specialized than f(T) or f(T*)  
    float x;  
    g(x);  // Ambiguous: g(T) or g(T&)  
    A<int> z;  
    h(z);  // overload resolution selects h(A<T>&)  
    const A<int> z2;  
    h(z2);  // h(const T&) is called because h(A<T>&) is not callable
}
```

— end example]

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
}
```

— end example]

### 14.5.7 Template aliases

A template-declaration in which the declaration is an alias-declaration (clause 7) declares the identifier to be a template alias. A template alias is 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;

template<class T>  
void process(Vec<T>& v)  
{ /* ... */ }

template<class T>  
void process(vector<T, Alloc<T>>& w)  
{ /* ... */ } // error: redefinition

template<template<class> class TT>  
void f(TT<int>);

f(v); // error: Vec not deduced

template<template<class,class> class TT>  
void g(TT<int, Alloc<int>>);

g(v); // OK: TT = vector
```

— end example]  

14.6 Name resolution [temp.res]

1 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.

2 A name used in a template declaration or definition and that is dependent on a template-parameter is assumed not to name a type unless the applicable name lookup finds a type name or the name is qualified by the keyword `typename`. [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;
        T*A* 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:
```
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 typename-specifier. If the qualified-id in a typename-specifier does not denote a type, the program is ill-formed.

**typename-specifier:**

\[
\text{typename} : \text{opt} \quad \text{nested-name-specifier} \quad \text{identifier} \\
\text{typename} : \text{opt} \quad \text{nested-name-specifier} \quad \text{template} \quad \text{opt} \quad \text{simple-template-id}
\]

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:

```cpp
struct A {
    struct X { };
    int X;
};
struct B {
    struct X { };
};
template<class T> void f(T t) {
    typename T::X x;
}
void foo() {
    A a;
    B b;
    f(b); // OK: T::X refers to B::X
    f(a); // error: T::X refers to the data member A::X not the struct A::X
}
```

— end example]

5 A qualified name used as the name in a mem-initializer-id, a base-specifier, or an elaborated-type-specifier is implicitly assumed to name a type, without the use of the typename keyword. [Note: the typename keyword is not permitted by the syntax of these constructs. — end note]

6 If, for a given set of template arguments, a specialization of a template is instantiated that refers to a qualified-id that denotes a type, and the nested-name-specifier of the qualified-id depends on a template parameter, the qualified-id shall either be prefixed by typename or shall be used in a context in which it implicitly names a type as described above. [Example:

```cpp
template <class T> void f(int i) {
    T::x * i; // T::x must not be a type
}
```

struct Foo {
    typedef int x;
}
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:

```
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:

```
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:

```
struct Bar {
    static int const x = 5;
};

int main() {
    f<Bar>(1); // OK
    f<Foo>(1); // error: Foo::x is a type
}
— end example]

§ 14.6
#include <iostream>
using namespace std;

template<class T> class Set {
    T* p;
    int cnt;
public:
    Set();
    Set<T>(const Set<T>&);
    void printall() {
        for (int i = 0; i<cnt; i++)
            cout << p[i] << 'n';
    }
};

In the example, i is the local variable i declared in printall, cnt is the member cnt declared in Set, and cout is the standard output stream declared in ostream. 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]

14.6.1 Locally declared names [temp.local]

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.

2 Within the scope of a class template specialization or partial specialization, when the injected-class-name is not followed by a <, it is equivalent to the injected-class-name followed by the template-parameters 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]

3 The injected-class-name of a class template or class template specialization can be used either with or without a template-argument-list wherever it is in scope. [Example:

```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]

4 A lookup that finds an injected-class-name (10.2) can result in an ambiguity in certain cases (for example, if it is found in more than one base class). If all of the injected-class-names that are found refer to specializations of the same class template, and if the name is followed by a template-argument-list, the reference refers to the class template itself and not a specialization thereof, and is not ambiguous. [Example:

```cpp
template <class T> struct Base { public:
};
template <class T> struct Derived: Base<int>, Base<char> {
    typename Derived::Base b;  // error: ambiguous
    typename Derived::Base<double> d;  // OK
};
```

— end example]

5 When the normal name of the template (i.e., the name from the enclosing scope, not the injected-class-name) is used 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;
    X<int>* p3;
    ::X* p4;  // error: missing template argument list
};
```

— end example]
6 A *template-parameter* shall not be redeclared within its scope (including nested scopes). A *template-parameter* shall not have the same name as the template name. [Example:

```c
#include <iostream>

int main() {
  template<class T, int i> class Y {
    int T; // error: template-parameter redeclared
    void f() {
      char T; // error: template-parameter redeclared
    }
  };

  template<class X> class X; // error: template-parameter redeclared

  — end example]
```

7 In the definition of a member of a class template that appears outside of the class template definition, the name of a member of this template hides the name of a *template-parameter*. [Example:

```c
#include <iostream>

int main() {
  template<class T> struct A {
    struct B { /* ... */ }; // A's B, not the template parameter
    void f();
  };

  template<class B> void A<B>::f() {
    B b; // error: template-parameter redeclared
  }

  — end example]
```

8 In the definition of a member of a class template that appears outside of the namespace containing the class template definition, the name of a *template-parameter* hides the name of a member of this namespace. [Example:

```c
#include <iostream>

namespace N {
  class C { }
  template<class T> class B {
    void f(T);
  };
}

template<class C> void N::B<C>::f(C) {
  C b; // C is the template parameter, not N::C
}

— end example]
```

9 In the definition of a class template or in the definition of a member of such a template that appears outside of the template definition, for each 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 (3.3.10). [Example:

```c
#include <iostream>

struct A {
  struct B { /* ... */ }; // A's B, not the template parameter
  int a;
  int Y;
};

template<class B, class a> struct X : A {
  B b; // A's B
```

§ 14.6.1
1 Inside a template, some constructs have semantics which may differ from one instantiation to another. Such a construct depends on the template parameters. In particular, types and expressions may depend on the type and/or value of template parameters (as determined by the template arguments) and this determines the context for name lookup for certain names. Expressions may be type-dependent (on the type of a template parameter) or value-dependent (on the value of a non-type template parameter). In an expression of the form:

postfix-expression ( expression-list_opt )

where the postfix-expression is an unqualified-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.

2 [Example:

```cpp
template<class T> struct X : B<T> {
    typename T::A* pa;
    void f(B<T>* pb) {
        static int i = B<T>::i;
        pb->j++;
    }
};
```

the base class name B<T>, the type name T::A, the names B<T>::i and pb->j explicitly depend on the template-parameter. — end example]

3 In the definition of a class or class template, if a base class 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;
};
```

```cpp
int a;
```

§ 14.6.2
template<class T> struct Y : T {
    struct B { /* */ ... */ }; // The B defined in Y
    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]

In the definition of a class template, a nested class of a class template, a member of a class template, or a member of a nested class of a class template, a name refers to the current instantiation if it is

— the injected-class-name (9) of the class template or nested class,
— in the definition of a primary class template, the name of the class template followed by the template argument list of the primary template (as described below) enclosed in <>,
— in the definition of a nested class of a class template, the name of the nested class referenced as a member of the current instantiation, or
— in the definition of a partial specialization, the name of the class template followed by the template argument list of the partial specialization enclosed in <>. If the nth template parameter is a parameter pack, the nth template argument is a pack expansion (14.5.3) whose pattern is the name of the parameter pack.

The template argument list of a primary template is a template argument list in which the nth template argument has the value of the nth template parameter of the class template. If the nth template parameter is a template parameter pack (14.5.3), the nth template argument is a pack expansion (14.5.3) whose pattern is the name of the template parameter pack.

A template argument that is equivalent to a template parameter (i.e., has the same constant value or the same type as the template parameter) can be used in place of that template parameter in a reference to the current instantiation. In the case of a non-type template argument, the argument must have been given the value of the template parameter and not an expression in which the template parameter appears as a subexpression. [Example:

```cpp
template <class T> class A {
    A* p1; // A is the current instantiation
    A<T>* p2; // A<T> is the current instantiation
    A<T>* p3; // A<T> is not the current instantiation
    ::A<T>* p4; // ::A<T> is the current instantiation
    class B {
        B* p1; // B is the current instantiation
        A<T>::B* p2; // A<T>::B is the current instantiation
        typename A<T>::B* p3; // A<T>::B is not the current instantiation
    };
};

template <class T> class A<T*> {
    A<T*>* p1; // A<T*> is the current instantiation
};
```
A<T>* p2;  // A<T> is not the current instantiation

};

template <class T1, class T2, int I> struct B {
  B<T1, T2, I>* b1;  // refers to the current instantiation
  B<T2, T1, I>* b2;  // 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]

4 A name is a member of the current instantiation if it is

— An unqualified name that, when looked up, refers to a member of a class template. [Note: this can only occur when looking up a name in a scope enclosed by the definition of a class template. — end note]

— A qualified-id in which the nested-name-specifier refers to the current instantiation.

[Example:

template <class T> class A {
  static const int i = 5;
  int n1[i];  // i refers to a member of the current instantiation
  int n2[A::i];  // A::i refers to a member of the current instantiation
  int n3[A<T>::i];  // A<T>::i refers to a member of the current instantiation
  int f();
};

template <class T> int A<T>::f() {
  return i;  // i refers to a member of the current instantiation
}

— end example]

5 A name is a member of an unknown specialization if the name is a qualified-id in which the nested-name-specifier names a dependent type that is not the current instantiation.

6 A type is dependent if it is

— a template parameter,
— a member of an unknown specialization,
— a nested class that is a member of the current instantiation,
— a cv-qualified type where the cv-unqualified type is dependent,
— a compound type constructed from any dependent type,
— an array type constructed from any dependent type or whose size is specified by a constant expression that is value-dependent,

§ 14.6.2.1
— a *simple-template-id* in which either the template name is a template parameter or any of the template arguments is a dependent type or an expression that is type-dependent or value-dependent, or
— 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

> 1 Except as described below, an expression is type-dependent if any subexpression is type-dependent.

> 2 *this* is type-dependent if the class type of the enclosing member function is dependent (14.6.2.1).

> 3 An id-expression is type-dependent if it contains

> — an identifier associated by name lookup with one or more declarations declared with a dependent type,

> — a template-id that is dependent,

> — a conversion-function-id that specifies a dependent type, or

> — a nested-name-specifier or a qualified-id that names a member of an unknown specialization;

or if it names a static data member of the current instantiation that has type “array of unknown bound of T” for some T (14.5.1.3). Expressions of the following forms are type-dependent only if the type specified by the *type-id*, *simple-type-specifier* or *new-type-id* is dependent, even if any subexpression is type-dependent:

```
simple-type-specifier ( expression-list_opt )
::opt new new-placement_opt new-type-id new-initializer_opt
::opt new new-placement_opt ( type-id ) new-initializer_opt
dynamic_cast < type-id > ( expression )
static_cast < type-id > ( expression )
const_cast < type-id > ( expression )
reinterpret_cast < type-id > ( expression )
(type-id) cast-expression
```

> 4 Expressions of the following forms are never type-dependent (because the type of the expression cannot be dependent):

```
literal
postfix-expression . pseudo-destructor-name
postfix-expression -> pseudo-destructor-name
sizeof unary-expression
sizeof ( type-id )
sizeof ... ( identifier )
alignof ( type-id )
typeid ( expression )
typeid ( type-id )
::opt delete cast-expression
::opt delete [ ] cast-expression
throw assignment-expression_opt
noexcept ( expression )
```

[Note: For the standard library macro offsetof, see 18.2. — 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 [temp.dep.constexpr]

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.

Expressions of the following form are value-dependent if the unary-expression or expression is type-dependent or the type-id is dependent:

```
sizeof unary-expression
sizeof (type-id)
alignment (type-id)
noexcept (expression)
```

[Note: For the standard library macro offsetof, see 18.2. — 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
noexcept (expression)
```

4 Expressions of the following form are value-dependent:

```
sizeof ... (identifier)
```

14.6.2.4 Dependent template arguments [temp.dep.temp]

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.
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 or a decltype-specifier that denotes a dependent type.

14.6.3 Non-dependent names [temp.nondep]

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
  }
};

void g(int);  // not in scope at the point of the template
  // definition, not considered for the call g(1)
```

— end example]

14.6.4 Dependent name resolution [temp.dep.res]

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 [temp.point]

For a function template specialization, a member function template specialization, or a specialization for a member function or static data member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization and the context from which it is referenced depends on a template parameter, the point of instantiation of the specialization is the point of instantiation of the enclosing specialization. Otherwise, the point of instantiation for such a specialization immediately follows the namespace scope declaration or definition that refers to the specialization.

2 If a function template or member function of a class template is called in a way which uses the definition of a default argument of that function template or member function, the point of instantiation of the default argument is the point of instantiation of the function template or member function specialization.

3 For a class template specialization, a class member template specialization, or a specialization for a class member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization, if the context from which the specialization is referenced depends on a template parameter, and if the specialization is not instantiated previous to the instantiation of the enclosing template, the point of instantiation is immediately before the point of instantiation of the enclosing template. Otherwise, the point of instantiation for such a specialization immediately precedes the namespace scope declaration or definition that refers to the specialization.
If a virtual function is implicitly instantiated, its point of instantiation is immediately following the point of instantiation of its enclosing class template specialization.

An explicit instantiation definition is an instantiation point for the specialization or specializations specified by the explicit instantiation.

The instantiation context of an expression that depends on the template arguments is the set of declarations with external linkage declared prior to the point of instantiation of the template specialization in the same translation unit.

A specialization for a function template, a member function template, or of a member function or static data member of a class template may have multiple points of instantiations within a translation unit. A specialization for a class template has at most one point of instantiation within a translation unit. A specialization for any template may have points of instantiation in multiple translation units. If two different points of instantiation give a template specialization different meanings according to the one definition rule (3.2), the program is ill-formed, no diagnostic required.

14.6.4.2 Candidate functions

For a function call that depends on a template parameter, if the function name is an unqualified-id 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 from the template definition context are found.

— For the part of the lookup using associated namespaces (3.4.2), only function declarations found in either the template definition context or the template instantiation context are found.

If the call would be ill-formed or would find a better match had the lookup within the associated namespaces considered all the function declarations with external linkage introduced in those namespaces in all translation units, not just considering those declarations found in the template definition and template instantiation contexts, then the program has undefined behavior.

14.6.5 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
}
```

(continued)
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.

A function instantiated from a function template is called an instantiated function. A class instantiated from a class template is called an instantiated class. A member function, a member class, or a static data member of a class template instantiated from the member definition of the class template is called, respectively, an instantiated member function, member class or static data member. A member function instantiated from a member function template is called an instantiated member function. A member class instantiated from a member class template is called an instantiated member class.

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<int>::x = 1; // specialize for T == int
```

— end example]

— end example]

4 An instantiated template specialization can be either implicitly instantiated (14.7.1) for a given argument list or be explicitly instantiated (14.7.2). A specialization is a class, function, or class member that is either instantiated or explicitly specialized (14.7.3).

5 For a given template and a given set of `template-arguments`,

— an explicit instantiation definition shall appear at most once in a program,
— an explicit specialization shall be defined at most once in a program (according to 3.2), and
— both an explicit instantiation and a declaration of an explicit specialization shall not appear in a program unless the explicit instantiation follows a declaration of the explicit specialization.

An implementation is not required to diagnose a violation of this rule.

§ 14.7

356
6 Each class template specialization instantiated from a template has its own copy of any static members.

[Example:

```cpp
template<class T> class X {
    static T s;
};
template<class T> T X<T>::s = 0;
X<int> aa;
X<char*> bb;
```

`X<int>` has a static member `s` of type `int` and `X<char*>` has a static member `s` of type `char*`. — end example]

14.7.1 Implicit instantiation

1 Unless a class template specialization has been explicitly instantiated (14.7.2) or explicitly specialized (14.7.3), the class template specialization is implicitly instantiated when the specialization is referenced in a context that requires a completely-defined object type or when the completeness of the class type affects the semantics of the program. 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>::g() required
}
```

Nothing in this example requires `class Z<double>`, `Z<int>::g()`, or `Z<char>::f()` to be implicitly instantiated. — end example]

§ 14.7.1 357
A class template specialization is implicitly instantiated if the class type is used in a context that requires a completely-defined object type or if the completeness of the class type might affect the semantics of the program. [Note: in particular, if the semantics of an expression depend on the member or base class lists of a class template specialization, the class template specialization is implicitly generated. For instance, deleting a pointer to class type depends on whether or not the class declares a destructor, and conversion between pointer to class types depends on the inheritance relationship between the two classes involved.—end note] [Example:

```cpp
template<class T> class B { /* ... */
    template<class T> class D : public B<T> { /* ... */

    void f(void*);
    void f(B<int>**);

    void g(D<int>** p, D<char>** pp, D<double>** ppp) {
        f(p); // instantiation of D<int> required: call f(B<int>*)
        B<char>* q = pp; // instantiation of D<char> required:
            // convert D<char>** to B<char>**
        delete ppp; // instantiation of D<double> required
    }

    —end example]
```

If the overload resolution process can determine the correct function to call without instantiating a class template definition, it is unspecified whether that instantiation actually takes place. [Example:

```cpp
template <class T> struct S {
    operator int();
};

void f(int);
void f(S<int>&);
void f(S<float>);

void g(S<int>& sr) {
    f(sr); // instantiation of S<int> allowed but not required
                // instantiation of S<float> allowed but not required
}

—end example]

If an implicit instantiation of a class template specialization is required and the template is declared but not defined, the program is ill-formed. [Example:

```cpp
template<class T> class X;

X<char> ch; // error: definition of X required

—end example]
```

The implicit instantiation of a class template does not cause any static data members of that class to be implicitly instantiated.

If a function template or a member function template specialization is used in a way that involves overload resolution, a declaration of the specialization is implicitly instantiated (14.8.3).
An implementation shall not implicitly instantiate a function template, a 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 {
    public:
        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
    f(a);        // ill-formed; `ydef` is not declared
}
```

— end example]

§ 14.7.1
13 [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 or member function of a class 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

namespace N {  
template<class T> void f(T&) { }  
}  
template void N::f<int>(int&);  
```

— end example]

3 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.

4 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 function template, a member function template, or a member function or static data member of a class template shall be present in every translation unit in which it is explicitly instantiated.

5 An explicit instantiation of a class or function template specialization is placed in the namespace in which the template is defined. An explicit instantiation for a member of a class template is placed in the namespace where the enclosing class template is defined. An explicit instantiation for a member template is placed in the namespace where the enclosing class or class template is defined. [Example:

```cpp
namespace N {
    template<class T> class Y { void mf() { }};
}

template class Y<int>;      // error: class template Y not visible
    // in the global namespace
using N::Y;
template class Y<int>;      // error: explicit instantiation outside of the
    // namespace of the template

template class N::Y<char>;
template void N::Y<double>::mf();  // OK: explicit instantiation in namespace N

− end example ]

6 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:

```cpp
template<class T> class Array { /* ... */};
template<class T> void sort(Array<T>& v);

// instantiate sort(Array<int>&) - template-argument deduced
template void sort<>((Array<int>&);

− end example ]

7 An explicit instantiation that names a class template specialization is also an explicit instantiation of the same kind (declaration or definition) of each of its members (not including members inherited from base classes) that has not been previously explicitly specialized in the translation unit containing the explicit instantiation, except as described below. [Note: In addition, it will typically be an explicit instantiation of certain implementation-dependent data about the class. − end note]
An explicit instantiation definition that names a class template specialization explicitly instantiates the class template specialization and is an explicit instantiation definition of only those members whose definition is visible at the point of instantiation.

Except for inline functions and class template specializations, explicit instantiation declarations have the effect of suppressing the implicit instantiation of the entity to which they refer. [Note: The intent is that an inline function that is the subject of an explicit instantiation declaration will still be implicitly instantiated when used so that the body can be considered for inlining, but that no out-of-line copy of the inline function would be generated in the translation unit. — end note]

If an entity is the subject of both an explicit instantiation declaration and an explicit instantiation definition in the same translation unit, the definition shall follow the declaration. An entity that is the subject of an explicit instantiation declaration and that is also used in the translation unit shall be the subject of an explicit instantiation definition somewhere in the program; otherwise the program is ill-formed, no diagnostic required. [Note: This rule does apply to inline functions even though an explicit instantiation declaration of such an entity has no other normative effect. This is needed to ensure that if the address of an inline function is taken in a translation unit in which the implementation chose to suppress the out-of-line body, another translation unit will supply the body. — end note] An explicit instantiation declaration shall not name a specialization of a template with internal linkage.

The usual access checking rules do not apply to names used to specify explicit instantiations. [Note: In particular, the template arguments and names used in the function declarator (including parameter types, return types and exception specifications) may be private types or objects which would normally not be accessible and the template may be a member template or member function which would not normally be accessible. — end note]

An explicit instantiation does not constitute a use of a default argument, so default argument instantiation is not done. [Example:

```cpp
char* p = 0;
template<class T> T g(T = &p);
template int g<int>(int); // OK even though &p isn't an int.
```

— end example]

### 14.7.3 Explicit specialization [temp.expl.spec]

An explicit specialization of any of the following:
- non-deleted function template
- class template
- non-deleted member function of a class template
- static data member of a class template
- member class of a class template
- member class template of a class or class template
- non-deleted member function template of a class or class template

can be declared by a declaration introduced by `template<>`; that is:

```cpp
explicit-specialization:
  template <> declaration
```

[Example:}
template<class T> class stream;

template<> class stream<char> { /* ... */ };

template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v) { /* ... */ }

template<> void sort<char*>(Array<char*>&) ;

Given these declarations, stream<char> will be used as the definition of streams of chars; other streams will be handled by class template specializations instantiated from the class template. Similarly, sort<char*> will be used as the sort function for arguments of type Array<char*>; other Array types will be sorted by functions generated from the template. — end example

2 An explicit specialization shall be declared in a namespace enclosing the specialized template. An explicit specialization whose declarator-id is not qualified shall be declared in the nearest enclosing namespace of the template, or, if the namespace is inline (7.3.1), any namespace from its enclosing namespace set. Such a declaration may also be a definition. If the declaration is not a definition, the specialization may be defined later (7.3.1.2).

3 A declaration of a function template or class template being explicitly specialized shall precede the declaration of the explicit specialization. [ Note: a declaration, but not a definition of the template is required. — end note ] The definition of a class or class template shall precede the declaration of an explicit specialization for a member template of the class or class template. [ Example:

    template<> class X<int> { /* ... */ } ; // error: X not a template

    template<class T> class X ;

    template<> class X<char*> { /* ... */ } ; // OK: X is a template

— end example ]

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 precede the explicit specialization for the member of the class template. If such an explicit specialization for the member of a class template names an implicitly-declared special member function (Clause 12), the program is ill-formed.

5 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:

    template<class T> struct A {
        void f(T) { /* ... */ }
    } ;

    template<> struct A<int> {
        void f(int);  
    } ;

§ 14.7.3 363
void h() {
    A<int> a;
    a.f(16); // A<int>::f must be defined somewhere
}

// explicit specialization syntax not used for a member of
// explicitly specialized class template specialization
void A<int>::f(int) { /* ... */ }

— end example ]

6 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:

    template<class T> class Array { /* ... */ };  
    template<class T> void sort(Array<T>& v) { /* ... */ }

    void f(Array<String>& v) {
        sort(v);  // use primary template
        // sort(Array<T>&), T is String
    }

    template<> void sort<String>(Array<String>& v); // error: specialization
    // after use of primary template
    template<> void sort<char*>(Array<char*>& v);  // OK: sort<char*> not yet used

— end example ]

7 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 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.

8 A template explicit specialization is in the scope of the namespace in which the template was defined. [ Example:

    namespace N {
        template<class T> class X { /* ... */ };  
        template<class T> class Y { /* ... */ };  

        template<> class X<int> { /* ... */ };  // OK: specialization
        // in same namespace
        template<> class Y<double>;          // forward declare intent to
        // specialize for double

§ 14.7.3 364
template<> class N::Y<double> { /* */ };  // OK: specialization
  // in same namespace

— end example []

9 A *simple-template-id* that names a class template explicit specialization that has been declared but not defined can be used exactly like the names of other incompletely-defined classes (3.9). [Example:

    template<class T> class X;       // X is a class template
    template<> class X<int>;

    X<int>* p;              // OK: pointer to declared class X<int>
    X<int> x;               // error: object of incomplete class X<int>

— end example []

10 A trailing *template-argument* can be left unspecified in the *template-id* naming an explicit function template specialization provided it can be deduced from the function argument type. [Example:

    template<class T> class Array { /* */ };
    template<class T> void sort(Array<T>& v);

    // explicit specialization for sort(Array<int>&)
    // with deduced template-argument of type int
    template<> void sort(Array<int>&);

— end example []

11 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).

12 An explicit specialization of a function template is inline only if it is declared with the *inline* specifier or defined as deleted, and independently of whether its function template is inline. [Example:

    template<class T> void f(T) { /* */ }
    template<class T> inline T g(T) { /* */ }

    template<> inline void f<>(int) { /* */ }  // OK: inline
    template<> int g<>(int) { /* */ }          // OK: not inline

— end example []

13 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: the definition of a static data member of a template that requires default initialization must use a *braced-init-list*:

    template<> X Q<int>::::x;       // declaration
    template<> X Q<int>::::x ();    // error: declares a function
    template<> X Q<int>::::x {};    // definition

— end note []

14 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:

§ 14.7.3
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, class X2> void A<int>::g1(int, X2);

// member specialization even if defined in class definition
template<> void A<int>::h(int) { }

— end example]

A member or a member template may be nested within many enclosing class templates. In an explicit specialization for such a member, the member declaration shall be preceded by a template<> for each enclosing class template that is explicitly specialized. [Example:

    template<class T1> class A {
        template<class T2> class B {
            void mf();
        };
        template<> template<> class A<int>::B<double>;
        template<> template<> void A<char>::B<char>::mf();
    };

— end example]

In an explicit specialization declaration for a member of a class template or a member template that appears in namespace scope, the member template and some of its enclosing class templates may remain unspecialized, except that the declaration shall not explicitly specialize a class member template if its enclosing class templates are not explicitly specialized as well. In such explicit specialization declaration, the keyword template followed by a template-parameter-list shall be provided instead of the template<> preceding the explicit specialization declaration of the member. The types of the template-parameters in the template-parameter-list shall be the same as those specified in the primary template definition. [Example:  

    template <class T1> class A {
        template<class T2> class B {
            template<class T3> void mf1(T3);
            void mf2();
        };
        template <> template <class X>
class A<int>::B {
   template <class T> void mf1(T);
};
template <> template <> template<class T>
void A<int>::B<double>::mf1(T t) { }
template <class Y> template <>
void A<Y>::B<double>::mf2() { } // ill-formed; B<double> is specialized but
// its enclosing class template A is not

— end example]

A specialization of a member function template or member class template of a non-specialized class template
is itself a template.

An explicit specialization declaration shall not be a friend declaration.

Default function arguments shall not be specified in a declaration or a definition for one of the following explicit specializations:
— the explicit specialization of a function template;
— the explicit specialization of a member function template;
— the explicit specialization of a member function of a class template where the class template specialization to which the member function specialization belongs is implicitly instantiated. [Note: default function arguments may be specified in the declaration or definition of a member function of a class template specialization that is explicitly specialized. — end note]

14.8 Function template specializations

A function instantiated from a function template is called a function template specialization; so is an explicit specialization of a function template. Template arguments can be explicitly specified when naming the function template specialization, deduced from the context (e.g., deduced from the function arguments in a call to the function template specialization, see 14.8.2), or obtained from default template arguments.

Each function template specialization instantiated from a template has its own copy of any static variable. [Example:

template<class T> void f(T* p) {
   static T s;
};

void g(int a, char* b) {
   f(&a); // calls f<int>(int*)
   f(&b); // calls f<char*>(char*)
}

Here f<int>(int*) has a static variable s of type int and f<char*>(char*) has a static variable s of type char*. — end example]

14.8.1 Explicit template argument specification

Template arguments can be specified when referring to a function template specialization by qualifying the function template name with the list of template-arguments in the same way as template-arguments are specified in uses of a class template specialization. [Example:
template<class T> void sort(Array<T>& v);
void f(Array<dcomplex>& cv, Array<int>& ci) {
    sort<dcomplex>(cv); // sort(Array<dcomplex>&)
    sort<int>(ci); // sort(Array<int>&)
}

and

template<class U, class V> U convert(V v);

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 (14.5.3) not
otherwise deduced will be deduced to an empty sequence of template arguments. If all of the template
arguments can be deduced, they may all be omitted; in this case, the empty template argument list <>
itself may also be omitted. In contexts where deduction is done and fails, or in contexts where deduction
is not done, if a template argument list is specified and it, along with any default template arguments,
identifies a single function template specialization, then the template-id is an lvalue for the function template
specialization. [ Example:

    template<class X, class Y> X f(Y);
    template<class X, class Y, class ... Z> X g(Y);
    void h() {
        int i = f<int>(5.6); // Y is deduced to be double
        int j = f(5.6); // ill-formed: X cannot be deduced
        f<void>(f<int, bool>); // Y for outer f deduced to be
                                // int (*)(bool)
        f<void>(f<int>); // ill-formed: f<int> does not denote a
                         // single function template specialization
        int k = g<int>(5.6); // Y is deduced to be double, Z is deduced to an empty sequence
        f<void>(g<int, bool>); // Y for outer f is deduced to be
                                // int (*)(bool), Z is deduced to an empty sequence
    }

— end example ]

4 [ Note: An empty template argument list can be used to indicate that a given use refers to a specialization
of a function template even when a normal (i.e., non-template) function is visible that would otherwise be
used. For example:

§ 14.8.1
5 Template arguments that are present shall be specified in the declaration order of their corresponding `template-parameters`. The template argument list shall not specify more `template-arguments` than there are corresponding `template-parameters` unless one of the `template-parameters` is a template parameter pack.  

```
// Example:

template<class X, class Y, class Z> X f(Y, Z);

void g() {
  f<int, char*, double>("aa", 3.0);
  f<int, char*>("aa", 3.0);  // Z is deduced to be double
  f<int>("aa", 3.0);          // Y is deduced to be const char*, and
    // Z is deduced to be double
  f("aa", 3.0);              // error: X cannot be deduced
  f2<char, short, int, long>(); // OK
}
```

— end example

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,

```
// template <class T> void f(T);

class Complex {
  Complex(double);
};

void g() {
  f<Complex>(1);          // OK, means f<Complex>(Complex(1))
}
```

— end note

7 [Note: because the explicit template argument list follows the function template name, and because conversion member function templates and constructor member function templates are called without using a function name, there is no way to provide an explicit template argument list for these function templates.

— end note

8 [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:]
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

1 When a function template specialization is referenced, all of the template arguments shall have values. The values can be explicitly specified or, in some cases, be deduced from the use or obtained from default template-arguments. [Example:

    void f(Array<dcomplex>& cv, Array<int>& ci) {  
        sort(cv);  
        // calls sort(Array<dcomplex>&)  
        sort(ci);  
        // calls sort(Array<int>&)  
    }  
    
    and  
    
    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.

— The specified template argument values are substituted for the corresponding template parameters as specified below.

After this substitution is performed, the function parameter type adjustments described in 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:

```cpp
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:

```cpp
template <class T, class U = double>
void f(T t = 0, U u = 0);

void g() {
  f(1, 'c'); // f<int, char>(1, 'c')
  f(1); // f<int, double>(1, 0)
  f(); // error: T cannot be deduced
  f<int>(); // f<int, double>(0, 0)
  f<int, char>(); // f<int, char>(0, 0)
}
— end example]
```
When all template arguments have been deduced or obtained from default template arguments, all uses of template parameters in 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.

At certain points in the template argument deduction process it is necessary to take a function type that makes use of template parameters and replace those template parameters with the corresponding template arguments. This is done at the beginning of template argument deduction when any explicitly specified template arguments are substituted into the function type, and again at the end of template argument deduction when any template arguments that were deduced or obtained from default arguments are substituted.

The substitution occurs in all types and expressions that are used in the function type and in template parameter declarations. The expressions include not only constant expressions such as those that appear in array bounds or as nontype template arguments but also general expressions (i.e., non-constant expressions) inside `sizeof`, `decltype`, and other contexts that allow non-constant expressions. [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 or expression. — end note]

If a substitution results in an invalid type or expression, type deduction fails. An invalid type or expression is one that would be ill-formed if written using the substituted arguments. Access checking is not done as part of the substitution process. Consequently, when deduction succeeds, an access error could still result when the function is instantiated. Only invalid types and expressions in the immediate context of the function type and its template parameter types can result in a deduction failure. [Note: The evaluation of the substituted types and expressions can result in side effects such as the instantiation of class template specializations and/or function template specializations, the generation of implicitly-defined functions, etc. Such side effects are not in the “immediate context” and can result in the program being ill-formed. — end note]

Example:

```cpp
struct X { };  
struct Y {
    Y(X) {}
};

template <class T> auto f(T t1, T t2) -> decltype(t1 + t2);  // #1  
X f(Y, Y);  // #2  
X x1, x2;  
X x3 = f(x1, x2);  // deduction fails on #1 (cannot add X+X), calls #2
```

— end example

[Note: Type deduction may fail for the following reasons:

— Attempting to instantiate a pack expansion containing multiple parameter packs of differing lengths.
— Attempting to create an array with an element type that is `void`, a function type, a reference type, or an abstract class type, or attempting to create an array with a size that is zero or negative. [Example:

```cpp
template <class T> int f(T[5]);  
int I = f<int>(0);  
int j = f<void>(0);  // invalid array
```

— end example]

— Attempting to use a type that is not a class type in a qualified name. [Example:
template <class T> int f(typename T::B*);
int i = f<int>(0);

— end example ]

— Attempting to use a type in a nested-name-specifier of a qualified-id when that type does not contain the specified member, or
— the specified member is not a type where a type is required, or
— the specified member is not a template where a template is required, or
— the specified member is not a non-type where a non-type is required.

[ Example:

template <int I> struct X { };
template <template <class T> class> struct Z { };
template <class T> void f(typename T::Y*){}
template <class T> void g(X<T::N>*){}
template <class T> void h(Z<T::template TT>*){}
struct A {};
struct B { int Y; };
struct C {
  typedef int N;
};
struct D {
  typedef int TT;
};

int main() {
  // Deduction fails in each of these cases:
  f<A>(0); // A does not contain a member Y
  f<B>(0); // The Y member of B is not a type
  g<C>(0); // The N member of C is not a non-type
  h<D>(0); // The TT member of D is not a template
}
— end example ]

— Attempting to create a pointer to reference type.

— Attempting to create a reference to void.

— Attempting to create “pointer to member of T” when T is not a class type. [ Example:

template <class T> int f(int T::*);
int i = f<int>(0);

— end example ]

— Attempting to give an invalid type to a non-type template parameter. [ Example:

template <class T, T> struct S {};
template <class T> int f(S<T, T>()*);
struct X {};
int i0 = f<X>(0);

— end example ]

§ 14.8.2 373
— Attempting to perform an invalid conversion in either a template argument expression, or an expression used in the function declaration.  [Example:

```cpp
template <class T, T*> int f(int);
int i2 = f<int,1>(0);  // can't conv 1 to int*
```
— end example]

— Attempting to create a function type in which a parameter has a type of `void`, or in which the return type is a function type or array type.

— Attempting to create a function type in which a parameter type or the return type is an abstract class type (10.4).

— end note]

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. If removing references and cv-qualifiers from \(P\) gives `std::initializer_list<\(P'\)` for some \(P'\) and the argument is an initializer list (8.5.4), then deduction is performed instead for each element of the initializer list, taking \(P'\) as a function template parameter type and the initializer element as its argument. Otherwise, an initializer list argument causes the parameter to be considered a non-deduced context (14.8.2.5). [Example:

```cpp
template<class T> void f(std::initializer_list<T>);
f({1,2,3});  // \(T\) deduced to int
f({1,"asdf"});  // error: \(T\) deduced to both int and const char*
```

```cpp
template<class T> void g(T);
g({1,2,3});  // error: no argument deduced for \(T\)
```
— end example] For a function parameter pack that occurs at the end of the `parameter-declaration-list`, 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. For a function parameter pack that does not occur at the end of the `parameter-declaration-list`, the type of the parameter pack is a non-deduced context. [Example:

```cpp
template<class ... Types> void f(Types&& ...);
template<class T1, class ... Types> void g(T1, Types ...);
```

```cpp
void h(int x, float& y) {
    const int z = x;
}
```
f(x, y, z);  // Types is deduced to int, float, const int
g(x, y, z);  // T1 is deduced to int; Types is deduced to float, int
}

— 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.

3 If P is a cv-qualified type, the top level cv-qualifiers of P’s type are ignored for type deduction. If P is a reference type, the type referred to by P is used for type deduction. If P is an rvalue reference to a cv-unqualified template parameter and the argument is an lvalue, the type “lvalue reference to A” is used in place of A for type deduction. [ Example:

   template <class T> int f(T&&);
   template <class T> int g(const T&&);
   int i;
   int n1 = f(i);  // calls f<int&>(int&)
   int n2 = f(0);  // calls f<int>(int&&)
   int n3 = g(i);  // error: would call g<int>(const int&&), which
   // would bind an rvalue reference to an lvalue

   — end example ]

4 In general, the deduction process attempts to find template argument values that will make the deduced A identical to A (after the type A is transformed as described above). However, there are three cases that allow a difference:
   — 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 ]

5 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]

8

[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]

9

[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]

1 Template arguments can be deduced from the type specified when taking the address of an overloaded function (13.4). The function template’s function type and the specified type are used as the types of P and A, and the deduction is done as described in 14.8.2.5.

14.8.2.3 Deducing conversion function template arguments
[temp.deduct.conv]

1 Template argument deduction is done by comparing the return type of the conversion function template (call it P; see 8.5, 13.3.1.5, and 13.3.1.6 for the determination of that type) with the type that is required as the result of the conversion (call it A) as described in 14.8.2.5.

2 If P is a reference type, the type referred to by P is used in place of P for type deduction and for any further references to or transformations of P in the remainder of this section.

3 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.

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.

These alternatives are considered only if type deduction would otherwise fail. If they yield more than one possible deduced \( A \), the type deduction fails.

When the deduction process requires a qualification conversion for a pointer or pointer to member type as described above, the following process is used to determine the deduced template argument values:

If \( A \) is a type

\[
\text{cv}_{1,0} \text{ “pointer to ...” cv}_{1,n-1} \text{ “pointer to” cv}_{1,n} T1
\]

and \( P \) is a type

\[
\text{cv}_{2,0} \text{ “pointer to ...” cv}_{2,n-1} \text{ “pointer to” cv}_{2,n} T2
\]

The cv-unqualified \( T1 \) and \( T2 \) are used as the types of \( A \) and \( P \) respectively for type deduction. [Example:

```cpp
struct A {
    template <class T> operator T***();
};
A a;
const int * const * const * p1 = a;  // T is deduced as int, not const int
```

— end example]

### 14.8.2.4  Deducing template arguments during partial ordering

Template argument deduction is done by comparing certain types associated with the two function templates being compared.

Two sets of types are used to determine the partial ordering. For each of the templates involved there is the original function type and the transformed function type. [Note: the creation of the transformed type is described in 14.5.6.2. — end note] The deduction process uses the transformed type as the argument template and the original type of the other template as the parameter template. This process is done twice for each type involved in the partial ordering comparison: once using the transformed template-1 as the argument template and template-2 as the parameter template and again using the transformed template-2 as the argument template and template-1 as the parameter template.

The types used to determine the ordering depend on the context in which the partial ordering is done:

- In the context of a function call, the function parameter types are used.
- In the context of a call to a conversion operator, the return types of the conversion function templates are used.
— In other contexts (14.5.6.2) the function template’s function type is used.

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:

---

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:

---

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 { };
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&amp; ...>); // #3

g(Tuple<>());    // calls #1
g(Tuple<int, float>()); // calls #2
```
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.
A function parameter for which the associated argument is an initializer list (8.5.4) but the parameter does not have `std::initializer_list` or reference to possibly cv-qualified `std::initializer_list` type. [Example:

```cpp
template<class T> void g(T);
g({1,2,3}); // error: no argument deduced for T
```
— end example]

A function parameter pack that does not occur at the end of the `parameter-declaration-clause`.

6 When a type name is specified in a way that includes a non-deduced context, all of the types that comprise that type name are also non-deduced. However, a compound type can include both deduced and non-deduced types. [Example: If a type is specified as `A<T>::B<T2>`, both `T` and `T2` are non-deduced. Likewise, if a type is specified as `A<\langle I+J\rangle::X<T>\rangle`, `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:

§ 14.8.2.5
template <class T> struct B { }

template <class T> struct D : public B<T> {};

struct D2 : public B<int> {};

template <class T> void f(B<T>&) {}

void t() {
  D<int> d;
  D2   d2;
  f(d);    // calls f(B<int>&)
  f(d2);   // calls f(B<int>&)
}

— end example

8 A template type argument T, a template template argument TT or a template non-type argument i can be deduced if P and A have one of the following forms:

T
cv-list T
T*
T&
T&&
T[integer-constant ]
template-name <T> (where template-name refers to a class template)
type (T)
T()
T(T)
T type ::*
type T::*
T T::*
T (type ::*)(())
type (T::*)(())
type (type ::*)(T)
type (T::*)(T)
T (type ::*)(T)
T (T::*)(())
T (T::*)(T)
type [i]
template-name <i> (where template-name refers to a class template)
TT<i>
TT<i>
TT<>

where (T) represents a parameter-type-list where at least one parameter type contains a T, and () represents a parameter-type-list where no parameter type contains a T. Similarly, <T> represents template argument lists where at least one argument contains a T, <i> represents template argument lists where at least one argument contains an i and <> represents template argument lists where no argument contains a T or an i.

9 If P has a form that contains <T> or <i>, then each argument P_i of the respective template argument list P is compared with the corresponding argument A_i of the corresponding template argument list of A. If the template argument list of P contains a pack expansion that is not the last template argument, the entire template argument list is a non-deduced context. If P_i is a pack expansion, then the pattern of P_i is compared with each remaining argument in the template argument list of A. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by P_i.

10 Similarly, if P has a form that contains (T), then each parameter type P_i of the respective parameter-type-list of P is compared with the corresponding parameter type A_i of the corresponding parameter-type-list of A. If
the parameter-declaration corresponding to \( P_i \) is a function parameter pack, then the type of its declarator-id is compared with each remaining parameter type in the parameter-type-list of \( A \). Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack. [Note: A function parameter pack can only occur at the end of a parameter-declaration-list (8.3.5). — end note]

11 These forms can be used in the same way as \( T \) is for further composition of types. [Example:

\[
X<int> (*)(char[6])
\]

is of the form

\[
template-name <T> (*)(type[i])
\]

which is a variant of

\[
type (*)(T)
\]

where type is \( X<int> \) and \( T \) is \( char[6] \). — end example]

12 Template arguments cannot be deduced from function arguments involving constructs other than the ones specified above.

13 A template type argument cannot be deduced from the type of a non-type template-argument.

14 [Example:

\[
template<class T, T i> void f(double a[10][i]);
int v[10][20];
f(v); // error: argument for template-parameter \( T \) cannot be deduced
\]

— end example]

15 [Note: except for reference and pointer types, a major array bound is not part of a function parameter type and cannot be deduced from an argument:

\[
template<int i> void f1(int a[10][i]);
template<int i> void f2(int a[i][20]);
template<int i> void f3(int (&a)[i][20]);
\]

\[
void g() {
int v[10][20];
f1(v); // OK: \( i \) deduced to be 20
f1<20>(v); // OK
f2(v); // error: cannot deduce template-argument \( i \)
f2<20>(v); // OK
f3(v); // OK: \( i \) deduced to be 10
}
\]

— end example]

16 If, in the declaration of a function template with a non-type template parameter, the non-type template parameter is used in a subexpression in the function parameter list, the expression is a non-deduced context as specified above. [Example:

\[
template <int i> class A { /*...*/};
template <int i> void g(A<i+1>);
template <int i> void f(A<i>, A<i+1>);
void k() {
    A<i> a1;
}
\]
A<2> a2;
g(a1); // error: deduction fails for expression i+1
g<0>(a1); // OK
f(a1, a2); // OK
}

— end example] — end note] [ Note: template parameters do not participate in template argument deduction if they are used only in non-deduced contexts. For example,

```cpp
template<int i, typename T>
T deduce(typename A<T>::X x, // T is not deduced here
        T t, // but T is deduced here
typename B<i>::Y y); // i is not deduced here
A<int> a;
B<77> b;
```

```cpp
int x = deduce<77>(a.xm, 62, b.ym);
// T is deduced to be int, a.xm must be convertible to
// A<int>::X
// i is explicitly specified to be 77, b.ym must be convertible
// to B<77>::Y
```

— end note]

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. 141 [ 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
}
```

```cpp
template<const short cs> class B { };  
template<short s> void g(B<s>);  
void k2() {
    B<1> b;
    g(b); // OK: cv-qualifiers are ignored on template parameter types
}
```

— end example]

18 A template-argument can be deduced from a function, pointer to function, or pointer to member function type.

[ Example:

```cpp
template<class T> void f(void(*)(T,int));  
template<class T> void foo(T,int);  
void g(int,int);
```

---

141) Although the template-argument corresponding to a template-parameter of type bool may be deduced from an array bound, the resulting value will always be true because the array bound will be non-zero.
void 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 —

19 A template type-parameter cannot be deduced from the type of a function default argument. [Example:

    template <class T> void f(T = 5, T = 7);
    void g() {
        f(1); // OK: call f<int>(1,7)
        f(); // error: cannot deduce T
        f<int>(); // OK: call f<int>(5,7)
    }

— end example —

20 The template-argument corresponding to a template template-parameter is deduced from the type of the template-argument of a class template specialization used in the argument list of a function call. [Example:

    template <template <class T> class X> struct A { };
    template <template <class T> class X> void f(A<X>) { }
    template<class T> struct B { };
    A<B> ab;
    f(ab); // calls f(A<B>)

— end example —

21 [Note: Template argument deduction involving parameter packs (14.5.3) can deduce zero or more arguments for each parameter pack. — end note] [Example:

    template<class> struct X { };
    template<class R, class ... ArgTypes> struct X<R(int, ArgTypes ...)> { };
    template<class ... Types> struct Y { };
    template<class T, class ... Types> struct Y<T, Types& ...> { };

    template<class ... Types> int f(void (*)(Types ...));
    void g(int, float);

    X<int> x1; // uses primary template
    X<int(float, double)> x2; // uses partial specialization; ArgTypes contains float, double
    X<int(float, int)> x3; // uses primary template
    Y<x1> y1; // use primary template; Types is empty
    Y<int(float, double)> y2; // uses partial specialization; T is int&, Types contains float, double
    Y<int(float, double)> y3; // uses primary template; Types contains int, float, double
    int fv = f(g); // OK; Types contains int, float

— end example —

22 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

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 13.3.3.142

[ Example:

template<class T> T max(T a, T b) { return a>b?a:b; }

void f(int a, int b, char c, char d) {
    int m1 = max(a,b); //max(int a, int b)
    char m2 = max(c,d); //max(char a, char b)
    int m3 = max(a,c); // error: cannot generate max(int,char)
}

2 Adding the non-template function

    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) {

142) The parameters of function template specializations contain no template parameter types. The set of conversions allowed
on deduced arguments is limited, because the argument deduction process produces function templates with parameters that
either match the call arguments exactly or differ only in ways that can be bridged by the allowed limited conversions. Non-
deduced arguments allow the full range of conversions. Note also that 13.3.3 specifies that a non-template function will be given
preference over a template specialization if the two functions are otherwise equally good candidates for an overload match.

§ 14.8.3 385
Here is an example involving conversions on a function argument not involved in template-parameter deduction:

```c
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]

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:

```c
void g() {
    f("Annemarie"); // call of f<const char*>
}
```

— end example]

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]
15 Exception handling

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.

\[
\begin{align*}
\text{try-block:} & \quad \text{try} \quad \text{compound-statement} \quad \text{handler-seq} \\
\text{function-try-block:} & \quad \text{try} \quad \text{ctor-initializer}_{\text{opt}} \quad \text{compound-statement} \quad \text{handler-seq} \\
\text{handler-seq:} & \quad \text{handler} \quad \text{handler-seq}_{\text{opt}} \\
\text{handler:} & \quad \text{catch} \left( \text{exception-declaration} \right) \quad \text{compound-statement} \\
\text{exception-declaration:} & \quad \text{attribute-specifier}_{\text{opt}} \quad \text{type-specifier-seq} \quad \text{declarator} \\
& \quad \text{attribute-specifier}_{\text{opt}} \quad \text{type-specifier-seq} \quad \text{abstract-declarator}_{\text{opt}} \\
\text{throw-expression:} & \quad \text{throw} \quad \text{assignment-expression}_{\text{opt}} 
\end{align*}
\]

The optional attribute-specifier in an exception-declaration appertains to the formal parameter of the catch clause (15.3).

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:

```c
void f() {
    goto 11; // Illeg. formed
    goto 12; // Illeg. formed
    try {
        goto 11; // OK
        goto 12; // Illeg. formed
        11: ;
    } catch (...) {
        12: ;
        goto 11; // Illeg. formed
        goto 12; // OK
    }
}
```

—end example— A goto, break, return, or continue statement can be used to transfer control out of a try block or handler. When this happens, each variable declared in the try block will be destroyed in the context that directly contains its declaration. [Example:
lab: try {
    T1 t1;
    try {
        T2 t2;
        if (condition)
            goto lab;
    } catch(...) { /* handler 2 */ }
    } catch(...) { /* handler 1 */ }

Here, executing goto lab; will destroy first t2, then t1, assuming the condition does not declare a variable. Any exception raised while destroying t2 will result in executing handler 2; any exception raised while destroying t1 will result in executing handler 1. — end example]

4 A function-try-block associates a handler-seq with the ctor-initializer, if present, and the compound-statement. An exception thrown during the execution of the initializer expressions in the ctor-initializer or during the execution of the compound-statement transfers control to a handler in a function-try-block in the same way as an exception thrown during the execution of a try-block transfers control to other handlers. [Example:

```cpp
int f(int);
class C {
    int i;
    double d;
public:
    C(int, double);
};

C::C(int ii, double id)
try : i(f(ii)), d(id) {
    // constructor statements
}
catch (...) {
    // handles exceptions thrown from the ctor-initializer
    // and from the constructor statements
}
```
— end example]

15.1 Throwing an exception [except.throw]

1 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:

```cpp
try {
    // ...
}
catch(const char* p) {
    // handle character string exceptions here
}
```

and

```cpp
class Overflow {
public:
```

§ 15.1 388
Overflow(char,double,double);
};

void f(double x) {
    throw Overflow('+',x,3.45e107);
}

can be caught by a handler for exceptions of type Overflow

try {
    f(1.2);
} catch(Overflow& oo) {
    // handle exceptions of type Overflow here
}

— end example ]

2 When an exception is thrown, control is transferred to the nearest handler with a matching type (15.3); “nearest” means the handler for which the compound-statement or ctor-initializer following the try keyword was most recently entered by the thread of control and not yet exited.

3 A throw-expression initializes a temporary object, called the exception object, the type of which is determined by removing any top-level cv-qualifiers from the static type of the operand of throw and adjusting the type from “array of T” or “function returning T” to “pointer to T” or “pointer to function returning T”, respectively. The temporary is an lvalue and is used to initialize the variable named in the matching handler (15.3). If the type of the exception object would be an incomplete type or a pointer to an incomplete type other than (possibly cv-qualified) void the program is ill-formed. Except for these restrictions and the restrictions on type matching mentioned in 15.3, the operand of throw is treated exactly as a function argument in a call (5.2.2) or the operand of a return statement.

4 The memory for the exception object is allocated in an unspecified way, except as noted in 3.7.4.1. If a handler exits by rethrowing, control is passed to another handler for the same exception. The exception object is destroyed after either the last remaining active handler for the exception exits by any means other than rethrowing, or the last object of type std::exception_ptr (18.8.5) that refers to the exception object is destroyed, whichever is later. In the former case, the destruction occurs when the handler exits, immediately after the destruction of the object declared in the exception-declaration in the handler, if any. In the latter case, the destruction occurs before the destructor of std::exception_ptr returns. The implementation may then deallocate the memory for the exception object; any such deallocation is done in an unspecified way.

5 When the thrown object is a class object, the copy/move constructor and the destructor shall be accessible, even if the copy/move operation is elided (12.8).

6 An exception is considered caught when a handler for that exception becomes active (15.3). [Note: an exception can have active handlers and still be considered uncaught if it is rethrown. — end note]

7 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:

try {
    // ...
} catch (...) { // catch all exceptions
    // respond (partially) to exception

§ 15.1 389
throw;  // pass the exception to some
        // other handler
    }

— end example]

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]

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.

An object that is partially constructed or partially destroyed will have destructors executed for all of its fully constructed 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. If the object was allocated in a new-expression, the matching deallocation function (3.7.4.2, 5.3.4, 12.5), if any, is called to free the storage occupied by the object.

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]

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 or an rvalue reference type. The exception-declaration shall not denote a pointer or reference to an incomplete type, other than void*, const void*, volatile void*, or const volatile void*.

A handler of type “array of T” or “function returning T” is adjusted to be of type “pointer to T” or “pointer to function returning T”, respectively.

A handler is a match for an exception object of type E if

— 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:
class Matherr { /* ... */ virtual void vf(); };  
class Overflow: public Matherr { /* ... */ };  
class Underflow: public Matherr { /* ... */ };  
class Zerodivide: public Matherr { /* ... */ };  

void f() {  
    try {  
        g();  
    } catch (Overflow oo) {  
        // ...  
    } catch (Matherr mm) {  
        // ...  
    }  
}  

Here, the Overflow handler will catch exceptions of type Overflow and the Matherr handler will catch exceptions of type Matherr and of all types publicly derived from Matherr including exceptions of type Underflow and Zerodivide. — end example

4 The handlers for a try block are tried in order of appearance. That makes it possible to write handlers that can never be executed, for example by placing a handler for a derived class after a handler for a corresponding base class.

5 A ... in a handler’s exception-declaration functions similarly to ... in a function parameter declaration; it specifies a match for any exception. If present, a ... handler shall be the last handler for its try block.

6 If no match is found among the handlers for a try block, the search for a matching handler continues in a dynamically surrounding try block.

7 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 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 with static storage duration are not caught by a function-try-block on main(). Exceptions
thROWN in destructors of objects with thread storage duration or in constructors of namespace-scope objects with thread storage duration are not caught by a function-try-block on the initial function of the thread.

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 The object declared in an exception-declaration or, if the exception-declaration does not specify a name, a temporary (12.2) is copy-initialized (8.5) from the exception object. 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.

17 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.

```
exception-specification:
  dynamic-exception-specification
  noexcept-specification

dynamic-exception-specification:
  throw ( type-id-list_opt )

type-id-list:
  type-id ...opt
  type-id-list , type-id ...opt

noexcept-specification:
  noexcept ( constant-expression )
  noexcept
```

In a noexcept-specification, the constant-expression, if supplied, shall be a constant expression (5.19) that is contextually converted to bool (Clause 4). A noexcept-specification noexcept is equivalent to noexcept(true).

An exception-specification shall appear only on a function declarator for a function type, pointer to function type, reference to function type, or pointer to member function type that is the top-level type of a declaration or definition, or on such a type appearing as a parameter or return type in a function declarator. An exception-specification shall not appear in a typedef declaration or alias-declaration. [Example:

```
void f() throw(int); // OK
void (*fp)() throw (int); // OK
void g(void pfa() throw(int)); // OK
typedef int (*pf)() throw(int); // ill-formed
```

— end example] A type denoted in 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*. A type cv T, “array of T”, or “function returning T” denoted in an exception-specification is adjusted to type T, “pointer to T”, or “pointer to function returning T”, respectively.
Two exception-specifications are compatible if:

- both are non-throwing (see below), regardless of their form,
- both have the form `noexcept(constant-expression)` and the `constant-expressions` are equivalent,
- one exception-specification is a `noexcept-specification` allowing all exceptions and the other is of the form `throw(type-id-list)`, or
- both are `dynamic-exception-specifications` that have the same set of adjusted types.

If any declaration of a function has an exception-specification that is not a `noexcept-specification` allowing all exceptions, all declarations, including the definition and any explicit specialization, of that function shall have a compatible exception-specification. If any declaration of a pointer to function, reference to function, or pointer to member function has a exception-specification, all occurrences of that declaration shall have a compatible exception-specification. In an explicit instantiation an exception-specification may be specified, but is not required. If an exception-specification is specified in an explicit instantiation directive, it shall be compatible with the exception-specifications of other declarations of that function. A diagnostic is required only if the exception-specifications are not compatible within a single translation unit.

If a virtual function has an exception-specification, all declarations, including the definition, of any function that overrides that virtual function in any derived class shall only allow exceptions that are allowed by the exception-specification of the base class virtual function. [Example:

```cpp
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:

```cpp
class A { /* ... */ };  
void (*pf1)();          // no exception specification
void (*pf2)() throw(A);

void f() {
    pf1 = pf2;          // OK: pf1 is less restrictive
    pf2 = pf1;          // error: pf2 is more restrictive
}
```

— end example]

In such an assignment or initialization, exception-specifications on return types and parameter types shall be compatible. In other assignments or initializations, exception-specifications shall be compatible.

An exception-specification can include the same type more than once and can include classes that are related by inheritance, even though doing so is redundant. [Note: An exception-specification can also include the class `std::bad_exception` (18.8.2.1). — end note]
A function is said to allow an exception of type \( E \) if its dynamic-exception-specification contains a type \( T \) for which a handler of type \( T \) would be a match (15.3) for an exception of type \( E \).

Whenever an exception is thrown and the search for a handler (15.3) encounters the outermost block of a function with an exception-specification that does not allow the exception, then,

- if the exception-specification is a dynamic-exception-specification, the function `std::unexpected()` is called (15.5.2),
- otherwise, the function `std::terminate()` is called (15.5.1).

[Example:

```c
class X { };
class Y { };
class Z: public X { };
class W { };

void f() throw (X, Y) {
    int n = 0;
    if (n) throw X();    // OK
    if (n) throw Z();    // also OK
    throw W();          // will call std::unexpected()
}
```

— end example]

The function `std::unexpected()` may throw an exception that will satisfy the exception-specification for which it was invoked, and in this case the search for another handler will continue at the call of the function with this exception-specification (see 15.5.2), or it may call `std::terminate()`.

An implementation shall not reject an expression merely because when executed it throws or might throw an exception that the containing function does not allow. [Example:

```c
extern void f() throw(X, Y);

void g() throw(X) {
    f();        // OK
}
```

the call to \( f \) is well-formed even though when called, \( f \) might throw exception \( Y \) that \( g \) does not allow.

— end example]

A function with no exception-specification or with an exception-specification of the form `noexcept(constant-expression)` where the constant-expression yields `false` allows all exceptions. An exception-specification is non-throwing if it is of the form `throw()`, `noexcept`, or `noexcept(constant-expression)` where the constant-expression yields `true`. A function with a non-throwing exception-specification does not allow any exceptions.

An exception-specification is not considered part of a function’s type.

An implicitly declared special member function (Clause 12) shall have an exception-specification. If \( f \) is an implicitly declared default constructor, copy constructor, move constructor, destructor, copy assignment operator, or move 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:
struct A {
    A();
    A(const A&) throw();
    A(A&&) throw();
    ~A() throw(X);
};
struct B {
    B() throw();
    B(const B&) throw();
    B(B&&) throw(Y);
    ~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(D&&) throw(Y);
    // Implicit declaration of D::~D() throw(X, Y);
};

Furthermore, if A::~A() or B::~B() were virtual, D::~D() would not be as restrictive as that of A::~A, and the program would be ill-formed since a function that overrides a virtual function from a base class shall have an exception-specification at least as restrictive as that in the base class. — end example]

In a dynamic-exception-specification, a type-id followed by an ellipsis is a pack expansion (14.5.3).

[ Note: The use of dynamic-exception-specifications is deprecated (see Annex D). — end note]

15.5 Special functions [except.special]

The functions std::terminate() (15.5.1) and std::unexpected() (15.5.2) are used by the exception handling mechanism for coping with errors related to the exception handling mechanism itself. The function std::current_exception() (18.8.5) and the class std::nested_exception (18.8.6) can be used by a program to capture the currently handled exception.

15.5.1 The std::terminate() function [except.terminate]

In the following situations exception handling must be abandoned for less subtle error handling techniques:

— when the exception handling mechanism, after completing evaluation of the expression to be thrown but before the exception is caught (15.1), calls a function that exits via an uncaught exception,143

— when the exception handling mechanism cannot find a handler for a thrown exception (15.3), or

— when the search for a handler (15.3) encounters the outermost block of a function with a noexcept specification that does not allow the exception (15.4), or

— when the destruction of an object during stack unwinding (15.2) terminates by throwing an exception, or

— when initialization of a non-local variable with static or thread storage duration (3.6.2, 3.6.3) terminates by throwing an exception, or

— when destruction of an object with static or thread storage duration exits using an exception (3.6.3), or

143) 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.
— when execution of a function registered with `std::atexit` exits using an exception (18.5), or
— when a `throw-expression` with no operand attempts to rethrow an exception and no exception is being handled (15.1), or
— when `std::unexpected` throws an exception which is not allowed by the previously violated `exception-specification`, and `std::bad_exception` is not included in that `exception-specification` (15.5.2), or
— when the implementation’s default unexpected exception handler is called (18.8.2.2).

In such cases, `std::terminate()` is called (18.8.3). In the situation where no matching handler is found, it is implementation-defined whether or not the stack is unwound before `std::terminate()` is called. In the situation where the search for a handler (15.3) encounters the outermost block of a function with a `noexcept-specification` that does not allow the exception (15.4), it is implementation-defined whether 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

1. If a function with an `exception-specification` throws an exception that is not listed in the `exception-specification`, the function `std::unexpected()` is called (18.8.2) immediately after completing the stack unwinding for the former function.

2. [Note: By default, `std::unexpected()` calls `std::terminate()`, but a program can install its own handler function (18.8.2.3). In either case, the constraints in the following paragraph apply. — end note]

3. 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` (18.8.2.1) 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.

4. 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

1. 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 (18.8.4). 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.(193,901),(417,915)(193,910),(417,923) 396
16 Preprocessing directives [cpp]

1 A preprocessing directive consists of a sequence of preprocessing tokens that satisfies the following constraints: The first token in the sequence is a # preprocessing token that (at the start of translation phase 4) 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. Thus, preprocessing directives are commonly called “lines.” These “lines” have no other syntactic significance, as all white space is equivalent except in certain situations during preprocessing (see the # character string literal creation operator in 16.3.2, for example).
A text line shall not begin with a \# preprocessing token. A non-directive shall not begin with any of the directive names appearing in the syntax.

When in a group that is skipped (16.1), the directive syntax is relaxed to allow any sequence of preprocessing tokens to occur between the directive name and the following new-line character.

The only white-space characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the introducing \# preprocessing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other white-space characters in translation phase 3).

The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting translation unit.

The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

[Example: In:]

```c
#define EMPTY
EMPTY  #  include <file.h>
```

the sequence of preprocessing tokens on the second line is not a preprocessing directive, because it does not begin with a # at the start of translation phase 4, even though it will do so after the macro EMPTY has been replaced. — end example]

16.1 Conditional inclusion

The expression that controls conditional inclusion shall be an integral constant expression except that identifiers (including those lexically identical to keywords) are interpreted as described below and it may contain unary operator expressions of the form

\[
\text{defined } \text{identifier}
\]

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 (in the list of preprocessing tokens that will become the controlling expression) after all macro replacements have occurred shall be in the lexical form of a token (2.7).

Preprocessing directives of the forms

\[
\# if \quad \text{constant-expression new-line groupopt}
\]

\[
\#elif \quad \text{constant-expression new-line groupopt}
\]

check whether the controlling constant expression evaluates to nonzero.

Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the defined unary operator), just as in normal text. If the token defined is generated as a result of this replacement process or use of the defined unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined. After all replacements due to macro expansion and 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 18.3. For the purposes of this token conversion and evaluation all signed and unsigned integer types act as if they have the same representation as, respectively, intmax_t or uintmax_t (18.4). This includes interpreting character literals, which may involve converting escape sequences into execution character set members. Whether the numeric value for these character literals matches the value obtained when an identical character literal occurs in an expression (other than within a #if or #elif directive) is implementation-defined. Also, whether a single-character character literal may have a negative value is implementation-defined. Each subexpression with type bool is subjected to integral promotion before processing continues.

---

145) Because the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names — there simply are no keywords, enumeration constants, etc.

146) An alternative token (2.6) is not an identifier, even when its spelling consists entirely of letters and underscores. Therefore it is not subject to this replacement.

147) Thus on an implementation where \texttt{std::numeric_limits<int>::max()} is 0x7FFF and \texttt{std::numeric_limits<unsigned int>::max()} is 0xFFFF, the integer literal 0x8000 is signed and positive within a #if expression even though it is unsigned in translation phase 7 (2.2).

148) Thus, the constant expression in the following #if directive and if statement is not guaranteed to evaluate to the same value in these two contexts.

\[
\#if \ 'z' - 'a' == 25 \\
\text{if ('z' - 'a' == 25)
\]

§ 16.1
Preprocessing directives of the forms

```
#define identifier new-line group_opt
#define !identifier new-line group_opt
```

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent
to \#if defined identifier and \#if !defined identifier respectively.

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.\(^{149}\)

16.2 Source file inclusion \[cpp.include\]

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 se-
quence 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.\(^{150}\) The method by which a sequence
of preprocessing tokens between a < and a > preprocessing token pair or a pair of " characters is combined
into a single header name preprocessing token is implementation-defined.

5 The implementation shall provide unique mappings for sequences consisting of one or more nondigits or
digits (2.11) followed by a period (.) and a single nondigit. The first character shall not be a digit. The
implementation may ignore distinctions of alphabetical case.

\(^{149}\) As indicated by the syntax, a preprocessing token shall not follow a \#else or \#endif directive before the terminating
new-line character. However, comments may appear anywhere in a source file, including within a preprocessing directive.

\(^{150}\) Note that adjacent string literals are not concatenated into a single string literal (see the translation phases in 2.2); thus,
an expansion that results in two string literals is an invalid directive.
6 A #include preprocessing directive may appear in a source file that has been read because of a #include directive in another file, up to an implementation-defined nesting limit.

7 [Note: Although an implementation may provide a mechanism for making arbitrary source files available to the < > search, in general programmers should use the < > form for headers provided with the implementation, and the " " form for sources outside the control of the implementation. For instance:

```c
#include <stdio.h>
#include <unistd.h>
#include "usefullib.h"
#include "myprog.h"
```

— end note]

8 [Example: This illustrates macro-replaced #include directives:

```c
#if VERSION == 1
    #define INCFILE "vers1.h"
#elif VERSION == 2
    #define INCFILE "vers2.h" // and so on
#else
    #define INCFILE "versN.h"
#endif
#include INCFILE
```

— end example]

16.3 Macro replacement [cpp.replace]

1 Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

2 An identifier currently defined as an object-like macro 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.

3 There shall be white-space between the identifier and the replacement list in the definition of an object-like macro.

4 If the identifier-list in the macro definition does not end with an ellipsis, the number of arguments (including those arguments consisting of no preprocessing tokens) in an invocation of a function-like macro shall equal the number of parameters in the macro definition. Otherwise, there shall be 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.

5 The identifier ____VA_ARGS____ shall occur only in the replacement-list of a function-like macro that uses the ellipsis notation in the parameters.

6 A parameter identifier in a function-like macro shall be uniquely declared within its scope.

7 The identifier immediately following the define is called the macro name. There is one name space for macro names. Any white-space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro.
8 If a \# preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive could begin, the identifier is not subject to macro replacement.

9 A preprocessing directive of the form

\begin{verbatim}
  \# define identifier replacement-list new-line
\end{verbatim}

defines an object-like macro that causes each subsequent instance of the macro name\(^{151}\) to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive.\(^{152}\) The replacement list is then rescanned for more macro names as specified below.

10 A preprocessing directive of the form

\begin{verbatim}
  \# 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
\end{verbatim}

defines a function-like macro with parameters, whose use is similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the \#define preprocessing directive. Each subsequent instance of the function-like macro name followed by a ( as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is terminated by the matching ) preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

11 The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives,\(^{153}\) the behavior is undefined.

12 If there is a \ldots 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 \ldots).

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. 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.

\(^{151}\) Since, by macro-replacement time, all character literals and string literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see \section{2.2, translation phases}), they are never scanned for macro names or parameters.

\(^{152}\) An alternative token (\section{2.6}) 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.

\(^{153}\) Despite the name, a non-directive is a preprocessing directive.
2 An identifier \_\_VA_ARGS\_\_ that occurs in the replacement list shall be treated as if it were a parameter, and the variable arguments shall form the preprocessing tokens used to replace it.

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 A character string literal is a string-literal with no prefix. If, in the replacement list, a parameter is immediately preceded by a \# preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument. Each occurrence of white space between the argument’s preprocessing tokens becomes a single space character in the character string literal. White space before the first preprocessing token and after the last preprocessing token comprising the argument is deleted. Otherwise, the original spelling of each preprocessing token in the argument is retained in the character string literal, except for special handling for producing the spelling of string literals and character literals: a \ character is inserted before each " and \ character of a character literal or string literal (including the delimiting " characters). If the replacement that results is not a valid character string literal, the behavior is undefined. The character string literal corresponding to an empty argument is "". The order of evaluation of \# and ## operators is unspecified.

16.3.3 The ## operator

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
                        // char p[] = "x ## y";
```

The expansion produces, at various stages:

```c
join(x, y)
in_between(x hash_hash y)
in_between(x ## y)
```

154 Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.
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 placemarker preprocessing tokens are removed. Then the resulting preprocessing token sequence is rescanned, along with all subsequent preprocessing tokens of the source file, for more macro names to replace.

2 If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file’s preprocessing tokens), it is not replaced. Furthermore, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.

3 The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one, but all pragma unary operator expressions within it are then processed as specified in 16.9 below.

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
   ```
   # undef identifier new-line
   ```
causes the specified identifier no longer to be defined as a macro name. It is ignored if the specified identifier is not currently defined as a macro name.

3 [Note: The simplest use of this facility is to define a “manifest constant,” as in
   ```
   #define TABSIZE 100
   int table[TABSIZE];
   ```
   — end note]

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.
   ```
   #define max(a, b) ((a) > (b) ? (a) : (b))
   ```
The parentheses ensure that the arguments and the resulting expression are bound properly.

5 To illustrate the rules for redefinition and reexamination, the sequence
   ```
   #define x
   #define f(a) f(x * (a))
   #undef x
   ```
#define x 2
#define g f
#define z 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
(p)^m(m);

results in
f(2 * (y+1)) + f(2 * (f(2 * (z[0]))) % f(2 * (0)) + t(1);
f(2 * (2+(3,4)-0,1)) | f(2 * (~5)) & f(2 * (0,1))\'^m(0,1);

int i[] = { 1, 23, 4, 5, };
char c[2][6] = { "hello", "" };

6 To illustrate the rules for creating character string literals and concatenating tokens, the sequence

#define str(s) # s
#define xstr(s) str(s)
#define debug(s, t) printf("x" # s "= %d, x" # t "= %s", \
    x ## s, x ## t)
#define INCFILE(n) vers ## n
#define glue(a, b) a ## b
#define xglue(a, b) glue(a, b)
#define HIGHLOW "hello"
#define LOW LOW ", world"

debug(1, 2);
fputs(str(strncmp("abc\0d", "abc", '\4') == 0) str(: @
    " strncmp("abc\0d", "abc", '\4') == 0: @
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW)

results in
printf("x" "1" "= \%d, x" "2" "= \%s", x1, x2);
fputs("strncmp("abc\0d", "abc", '\4') == 0 " : \0n", s);
#include "vers2.h" (after macro replacement, before file access)
"hello";
"hello" ", world"
or, after concatenation of the character string literals,
printf("x1= \%d, x2= \%s", x1, x2);
fputs("strncmp("abc\0d", "abc", '\4') == 0: \0n", s);
#include "vers2.h" (after macro replacement, before file access)
"hello"
"hello, world"

Space around the # and ## tokens in the macro definition is optional.

7 To illustrate the rules for placemaker preprocessing tokens, the sequence

```c
#define t(x,y,z) x ## y ## z
int j[] = { t(1,2,3), t(4,5), t(6,7,8), t(9,10,11), t(12,13,14) };
```

results in

```c
int j[] = { 123, 456, 789, 101112 };  
```

8 To demonstrate the redefinition rules, the following sequence is valid.

```c
#define OBJ_LIKE (1-1)
#define OBJ_LIKE /* white space */ (1-1) /* other */
#define FUNC_LIKE(a) ( a )
#define FUNC_LIKE(a) ( /* note the white space */ \  
                       a /* other stuff on this line */)

But the following redefinitions are invalid:

```c
#define OBJ_LIKE (0) // different token sequence
#define OBJ_LIKE (1 - 1) // different white space
#define FUNC_LIKE(b) ( a ) // different parameter usage
#define FUNC_LIKE(b) ( b ) // different parameter spelling
```

9 Finally, to show the variable argument list macro facilities:

```c
#define debug(...) fprintf(stderr, _ _VA_ARGS_ _)
#define showlist(...) puts(#_ _VA_ARGS_ _)
#define report(test, ...) ((test) ? puts(#test) : printf(_ _VA_ARGS_ _))

debug("Flag");
printf("X = %d\n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

results in

```c
fprintf(stderr, "Flag" );
printf(stderr, "X = %d\n", x );
puts("The first, second, and third items." );
((x>y) ? puts("x>y") : printf("x is %d but y is %d", x, y));
```

— end note]

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 (2.2) while processing the source file to the current token.
A preprocessing directive of the form

```
# line digit-sequence new-line
```

causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer). If the digit sequence specifies zero or a number greater than 2147483647, the behavior is undefined.

A preprocessing directive of the form

```
# line digit-sequence " s-char-sequenceopt " new-line
```

sets the presumed line number similarly and changes the presumed name of the source file to be the contents of the character string literal.

A preprocessing directive of the form

```
# line pp-tokens new-line
```

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after line on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined; otherwise, the result is processed as appropriate.

### 16.5 Error directive

A preprocessing directive of the form

```
# error pp-tokensopt new-line
```

causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens, and renders the program ill-formed.

### 16.6Pragma directive

A preprocessing directive of the form

```
# pragma pp-tokensopt new-line
```

causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any pragma that is not recognized by the implementation is ignored.

### 16.7 Null directive

A preprocessing directive of the form

```
# 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.\footnote{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.}

\[155\]
_DATE_

The date of translation of the source file: a character string literal of the form "Mmm dd yyyy", where the names of the months are the same as those generated by the asctime function, and the first character of dd is a space character if the value is less than 10. If the date of translation is not available, an implementation-defined valid date shall be supplied.

.FILE_

The presumed name of the current source file (a character string literal).

.LINE_

The presumed line number (within the current source file) of the current source line (an integer 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 shall be supplied.

2 The following macro names are conditionally defined by the implementation:

.STDC_

Whether __STDC__ is predefined and if so, what its value is, are implementation-defined.

.STDC_MB_MIGHT_NEQ_WC_

The integer constant 1, intended to indicate that, in the encoding for wchar_t, a member of the basic character set need not have a code value equal to its value when used as the lone character in an ordinary character literal.

.STDC_VERSION_

Whether __STDC_VERSION__ is predefined and if so, what its value is, are implementation-defined.

.STDC_ISO_10646_

An integer constant of the form yyyy-mmL (for example, 199712L). If this symbol is defined, then every character in the Unicode required set, when stored in an object of type wchar_t, has the same value as the short identifier of that character. The Unicode required set consists of all the characters that are defined by ISO/IEC 10646, along with all amendments and technical corrigenda as of the specified year and month.

3 The values of the predefined macros (except for __FILE__ and __LINE__) remain constant throughout the translation unit.

4 If any of the pre-defined macro names in this subclause, or the identifier defined, is the subject of a #define or a #undef preprocessing directive, the behavior is undefined. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

16.9 Pragma operator

A unary operator expression of the form:

Pragma ( string-literal )

156) The presumed source file name and line number can be changed by the #line directive.
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]
17 Library introduction

17.1 General

This Clause describes the contents of the C\texttt{+} standard library, how a well-formed C++ program makes use of the library, and how a conforming implementation may provide the entities in the library.

The following subclauses describe the definitions (17.3), method of description (17.5), and organization (17.6.1) of the library. Clause 17.6, Clauses 18 through 30, and Annex D specify the contents of the library, as well as library requirements and constraints on both well-formed C++ programs and conforming implementations.

Detailed specifications for each of the components in the library are in Clauses 18–30, as shown in Table 12.

Table 12 — Library categories

<table>
<thead>
<tr>
<th>Clause</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>Language support library</td>
</tr>
<tr>
<td>19</td>
<td>Diagnostics library</td>
</tr>
<tr>
<td>20</td>
<td>General utilities library</td>
</tr>
<tr>
<td>21</td>
<td>Strings library</td>
</tr>
<tr>
<td>22</td>
<td>Localization library</td>
</tr>
<tr>
<td>23</td>
<td>Containers library</td>
</tr>
<tr>
<td>24</td>
<td>Iterators library</td>
</tr>
<tr>
<td>25</td>
<td>Algorithms library</td>
</tr>
<tr>
<td>26</td>
<td>Numerics library</td>
</tr>
<tr>
<td>27</td>
<td>Input/output library</td>
</tr>
<tr>
<td>28</td>
<td>Regular expressions library</td>
</tr>
<tr>
<td>29</td>
<td>Atomic operations library</td>
</tr>
<tr>
<td>30</td>
<td>Thread support library</td>
</tr>
</tbody>
</table>

The language support library (Clause 18) provides components that are required by certain parts of the C++ language, such as memory allocation (5.3.4, 5.3.5) and exception processing (Clause 15).

The diagnostics library (Clause 19) provides a consistent framework for reporting errors in a C++ program, including predefined exception classes.

The general utilities library (Clause 20) includes components used by other library elements, such as a predefined storage allocator for dynamic storage management (3.7.4).

The strings library (Clause 21) provides support for manipulating text represented as sequences of type \texttt{char}, sequences of type \texttt{char16\_t}, sequences of type \texttt{char32\_t}, sequences of type \texttt{wchar\_t}, and sequences of any other character-like type.

The localization library (Clause 22) provides extended internationalization support for text processing.

The containers (Clause 23), iterators (Clause 24), and algorithms (25) libraries provide a C++ program with access to a subset of the most widely used algorithms and data structures.

The numerics library (Clause 26) provides numeric algorithms and complex number components that extend support for numeric processing. The \texttt{valarray} component provides support for \textit{n}-at-a-time processing.
potentially implemented as parallel operations on platforms that support such processing. The random number component provides facilities for generating pseudo-random numbers.

11 The input/output library (Clause 27) provides the `iostream` components that are the primary mechanism for C++ program input and output. They can be used with other elements of the library, particularly strings, locales, and iterators.

12 The regular expressions library (Clause 28) provides regular expression matching and searching.

13 The atomic operations library (Clause 29) allows more fine-grained concurrent access to shared data than is possible with locks.

14 The thread support library (Clause 30) provides components to create and manage threads, including mutual exclusion and interthread communication.

17.2 The C standard library

1 The C++ standard library also makes available the facilities of the C standard library, suitably adjusted to ensure static type safety.

2 The descriptions of many library functions rely on the C standard library for the signatures and semantics of those functions. In all such cases, any use of the `restrict` qualifier shall be omitted.

17.3 Definitions

17.3.1 arbitrary-positional stream

a stream (described in Clause 27) that can seek to any integral position within the length of the stream. Every arbitrary-positional stream is also a repositionable stream.

17.3.2 blocked thread

a thread that is waiting for some condition (other than the availability of a processor) to be satisfied before it can continue execution.\(^\text{157}\) 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.

17.3.3 character

in Clauses 21, 22, 27, and 28, 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.

17.3.4 character container type

a class or a type used to represent a `character`. It is used for one of the template parameters of the string, `iostream`, and regular expression class templates. A character container type shall be a POD (3.9) type.

17.3.5

\(^\text{157}\) This definition is taken from POSIX.
comparison function
an operator function (13.5) for any of the equality (5.10) or relational (5.9) operators.

17.3.6 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.

17.3.7 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.

17.3.8 default behavior
a description of replacement function and handler function semantics. Any specific behavior provided by the implementation, within the scope of the required behavior.

17.3.9 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).

17.3.10 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.

17.3.11 modifier function
a class member function (9.3), other than constructors, assignment, or destructor, that alters the state of an object of the class.

17.3.12 move assignment
use of a move assignment operator.

17.3.13 move assignment operator
an assignment operator which accepts only an rvalue argument of the type being assigned to and might modify the argument as a side effect during the assignment.

§ 17.3
17.3.14  move constructor
a constructor which accepts only an rvalue argument of the type being constructed and might modify the
argument as a side effect during construction.

17.3.15  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.

17.3.16  NTCTS
a sequence of values that have character type, that precede the terminating null character type value charT().

17.3.17  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).

17.3.18  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 (2.2)
and resolving the definitions of all translation units (3.5).

17.3.19  repositional stream
a stream (described in Clause 27) that can seek to a position that was previously encountered.

17.3.20  required behavior
a description of replacement function and handler function semantics, applicable to both the behavior pro-
vided 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.

17.3.21  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.

17.3.22  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.

17.3.23 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.

17.4 Additional definitions
1.3 defines additional terms used elsewhere in this International Standard.

17.5 Method of description (Informative)
This subclause describes the conventions used to specify the C++ standard library. 17.5.1 describes the structure of the normative Clauses 18 through 30 and Annex D. 17.5.2 describes other editorial conventions.

17.5.1 Structure of each clause
1 Each library clause contains the following elements, as applicable:

— Summary
— Requirements
— Detailed specifications
— References to the Standard C library

17.5.1.2 Summary
The Summary provides a synopsis of the category, and introduces the first-level subclauses. Each subclause also provides a summary, listing the headers specified in the subclause and the library entities provided in each header.

2 Paragraphs labelled “Note(s):” or “Example(s):” are informative, other paragraphs are normative.

3 The contents of the summary and the detailed specifications include:

— macros
— values
— types
— classes and class templates
— functions and function templates
— objects

158) To save space, items that do not apply to a Clause are omitted. For example, if a Clause does not specify any requirements, there will be no “Requirements” subclause.
17.5.1.3 Requirements

Requirements describe constraints that shall be met by a C++ program that extends the standard library. Such extensions are generally one of the following:

- Template arguments
- Derived classes
- Containers, iterators, and algorithms that meet an interface convention

The string and iostream components use an explicit representation of operations required of template arguments. They use a class template `char_traits` to define these constraints.

Interface convention requirements are stated as generally as possible. Instead of stating “class X has to define a member function `operator++()`,” the interface requires “for any object `x` of class `X`, `++x` is defined.” That is, whether the operator is a member is unspecified.

Requirements are stated in terms of well-defined expressions that define valid terms of the types that satisfy the requirements. For every set of well-defined expression requirements there is a table that specifies an initial set of the valid expressions and their semantics. Any generic algorithm (Clause 25) that uses the well-defined expression requirements is described in terms of the valid expressions for its formal type parameters.

Template argument requirements are sometimes referenced by name. See 17.5.2.1.

In some cases the semantic requirements are presented as C++ code. Such code is intended as a specification of equivalence of a construct to another construct, not necessarily as the way the construct must be implemented.\(^{159}\)

17.5.1.4 Detailed Specifications

The detailed specifications each contain the following elements:

- 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

Descriptions of class member functions follow the order (as appropriate):\(^{160}\)

- constructor(s) and destructor
- copying, moving & assignment functions
- comparison functions
- modifier functions
- observer functions
- operators and other non-member functions

\(^{159}\) Although in some cases the code given is unambiguously the optimum implementation.

\(^{160}\) To save space, items that do not apply to a class are omitted. For example, if a class does not specify any comparison functions, there will be no “Comparison functions” subclause.
Descriptions of function semantics contain the following elements (as appropriate):\(^{161}\)

- **Requires:** the preconditions for calling the function
- **Effects:** the actions performed by the function
- **Synchronization:** the synchronization operations (1.10) applicable to the function
- **Postconditions:** the observable results established by the function
- **Returns:** a description of the value(s) returned by the function
- **Throws:** any exceptions thrown by the function, and the conditions that would cause the exception
- **Complexity:** the time and/or space complexity of the function
- **Remarks:** additional semantic constraints on the function
- **Error conditions:** the error conditions for error codes reported by the function.
- **Notes:** non-normative comments about the function

Whenever the **Effects:** element specifies that the semantics of some function \( F \) are Equivalent to some code sequence, then the various elements are interpreted as follows. If \( F \)'s semantics specifies a **Requires:** element, then that requirement is logically imposed prior to the equivalent-to semantics. Next, the semantics of the code sequence are determined by the **Requires:**, **Effects:**, **Postconditions:**, **Returns:**, **Throws:**, **Complexity:**, **Remarks:**, **Error conditions:**, and **Notes:** specified for the function invocations contained in the code sequence. The value returned from \( F \) is specified by \( F \)'s **Returns:** element, or if \( F \) has no **Returns:** element, a non-void return from \( F \) is specified by the **Returns:** elements in the code sequence. If \( F \)'s semantics contains a **Throws:**, **Postconditions:**, or **Complexity:** element, then that supersedes any occurrences of that element in the code sequence.

For non-reserved replacement and handler functions, Clause 18 specifies two behaviors for the functions in question: their required and default behavior. The default behavior describes a function definition provided by the implementation. The required behavior describes the semantics of a function definition provided by either the implementation or a C++ program. Where no distinction is explicitly made in the description, the behavior described is the required behavior.

If the formulation of a complexity requirement calls for a negative number of operations, the actual requirement is zero operations.\(^{162}\)

Complexity requirements specified in the library clauses are upper bounds, and implementations that provide better complexity guarantees satisfy the requirements.

Error conditions specify conditions where a function may fail. The conditions are listed, together with a suitable explanation, as the **enum class errc** constants (19.5) that could be used as an argument to function **make_error_condition** (19.5.3.5).

### 17.5.2 Other conventions

(1) This subclause describes several editorial conventions used to describe the contents of the C++ standard

---

\(^{161}\) To save space, items that do not apply to a function are omitted. For example, if a function does not specify any further preconditions, there will be no “Requires” paragraph.

\(^{162}\) This simplifies the presentation of complexity requirements in some cases.
library. These conventions are for describing implementation-defined types (17.5.2.1), and member functions (17.5.2.2).

### 17.5.2.1 Type descriptions

#### 17.5.2.1.1 General

The Requirements subclauses may describe names that are used to specify constraints on template arguments. These names are used in library Clauses to describe the types that may be supplied as arguments by a C++ program when instantiating template components from the library.

Certain types defined in Clause 27 are used to describe implementation-defined types. They are based on other types, but with added constraints.

#### 17.5.2.1.2 Enumerated types

Several types defined in Clause 27 are enumerated types. Each enumerated type may be implemented as an enumeration or as a synonym for an enumeration.

The enumerated type `enumerated` can be written:

```cpp
enum enumerated { V0 , V1 , V2 , V3 , ... .};
static const enumerated C0 (V0 );
static const enumerated C1 (V1 );
static const enumerated C2 (V2 );
static const enumerated C3 (V3 );
... .
```

Here, the names `C0`, `C1`, etc. represent enumerated elements for this particular enumerated type. All such elements have distinct values.

#### 17.5.2.1.3 Bitmask types

Several types defined in Clauses 18 through 30 and Annex D are bitmask types. Each bitmask type can be implemented as an enumerated type that overloads certain operators, as an integer type, or as a `bitset` (20.5).

The bitmask type `bitmask` can be written:

```cpp
enum bitmask { 
    V0 = 1 << 0, V1 = 1 << 1, V2 = 1 << 2, V3 = 1 << 3, ... .
};
static const bitmask C0 (V0 );
static const bitmask C1 (V1 );
static const bitmask C2 (V2 );
static const bitmask C3 (V3 );
... .
```

// For exposition only. 
// `int_type` is an integral type capable of 
// representing all values of bitmask
bitset operator& (bitmask X , bitmask Y ) {

---

163) Examples from 20.2 include: EqualityComparable, LessThanComparable, CopyConstructable, etc. Examples from 24.2 include: InputIterator, ForwardIterator, Function, Predicate, etc.

164) Such as an integer type, with constant integer values (3.9.1).
Here, the names $C_0$, $C_1$, etc. represent bitmask elements for this particular bitmask type. All such elements have distinct values such that, for any pair $C_i$ and $C_j$, $C_i \& C_j$ is nonzero and $C_i \& C_j$ is zero.

The following terms apply to objects and values of bitmask types:

1. To **set** a value $Y$ in an object $X$ is to evaluate the expression $X \| Y$.
2. To **clear** a value $Y$ in an object $X$ is to evaluate the expression $X \&= \sim Y$.
3. The value $Y$ is **set** in the object $X$ if the expression $X \& Y$ is nonzero.

## 17.5.2.1.4 Character sequences

1. The C standard library makes widespread use of characters and character sequences that follow a few uniform conventions:

   - A **letter** is any of the 26 lowercase or 26 uppercase letters in the basic execution character set.

   - The **decimal-point character** is the (single-byte) character used by functions that convert between a (single-byte) character sequence and a value of one of the floating-point types. It is used in the character sequence to denote the beginning of a fractional part. It is represented in Clauses 18 through 30 and Annex D by a period, ‘.’, which is also its value in the "C" locale, but may change during program execution by a call to `setlocale(int, const char*)`, or by a change to a `locale` object, as described in Clauses 22.3 and 27.

   - A **character sequence** is an array object (8.3.4) $A$ that can be declared as $T A[N]$, where $T$ is any of the types `char`, `unsigned char`, or `signed char` (3.9.1), optionally qualified by any combination of `[character.seq]"C0", "C1", etc. represent bitmask elements for this particular bitmask type. All such elements have distinct values such that, for any pair $C_i$ and $C_j$, $C_i \& C_j$ is nonzero and $C_i \& C_j$ is zero.

   **The following terms apply to objects and values of bitmask types:**

   - To **set** a value $Y$ in an object $X$ is to evaluate the expression $X \| Y$.
   - To **clear** a value $Y$ in an object $X$ is to evaluate the expression $X \&= \sim Y$.
   - The value $Y$ is **set** in the object $X$ if the expression $X \& Y$ is nonzero.

   **17.5.2.1.4 Character sequences**

1. The C standard library makes widespread use of characters and character sequences that follow a few uniform conventions:

   - A **letter** is any of the 26 lowercase or 26 uppercase letters in the basic execution character set.

   - The **decimal-point character** is the (single-byte) character used by functions that convert between a (single-byte) character sequence and a value of one of the floating-point types. It is used in the character sequence to denote the beginning of a fractional part. It is represented in Clauses 18 through 30 and Annex D by a period, ‘.’, which is also its value in the "C" locale, but may change during program execution by a call to `setlocale(int, const char*)`, or by a change to a `locale` object, as described in Clauses 22.3 and 27.

   - A **character sequence** is an array object (8.3.4) $A$ that can be declared as $T A[N]$, where $T$ is any of the types `char`, `unsigned char`, or `signed char` (3.9.1), optionally qualified by any combination of

---

165) Note that this definition differs from the definition in ISO C 7.1.1.
166) declared in `<clocale>` (22.6).
const or volatile. The initial elements of the array have defined contents up to and including an element determined by some predicate. A character sequence can be designated by a pointer value \( S \) that points to its first element.

17.5.2.1.4.1 Byte strings

A **null-terminated byte string**, or NTBS, is a character sequence whose highest-addressed element with defined content has the value zero (the *terminating null* character); no other element in the sequence has the value zero.\(^{167}\)

The **length** of an NTBS is the number of elements that precede the terminating null character. An **empty** NTBS has a length of zero.

The **value** of an NTBS is the sequence of values of the elements up to and including the terminating null character.

A **static** NTBS is an NTBS with static storage duration.\(^{168}\)

17.5.2.1.4.2 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.\(^{169}\)

A **static** NTMBS is an NTMBS with static storage duration.

17.5.2.1.4.3 char16_t sequences

A **char16-character sequence** is an array object (8.3.4) \( T A[N] \), where \( T \) is type char16_t (3.9.1), optionally qualified by any combination of const or volatile. The initial elements of the array have defined contents up to and including an element determined by some predicate. A char16-character sequence can be designated by a pointer value \( S \) that designates its first element.

A **null-terminated char16-character string**, or NTC16s, is a char16-character sequence whose highest-addressed element with defined content has the value zero.\(^{170}\)

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.

The **value** of an NTC16s is the sequence of values of the elements up to and including the terminating null character.

A **static** NTC16s is an NTC16s with static storage duration.\(^{171}\)

17.5.2.1.4.4 char32_t sequences

A **char32-character sequence** is an array object (8.3.4) \( T A[N] \), where \( T \) is type char32_t (3.9.1), optionally qualified by any combination of const or volatile. The initial elements of the array have defined contents up to and including an element determined by some predicate. A char32-character sequence can be designated by a pointer value \( S \) that designates its first element.

\(^{167}\) Many of the objects manipulated by function signatures declared in <cstring> (21.7) 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.\(^{168}\) A string literal, such as \"abc\", is a static NTBS.\(^{169}\) 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.\(^{170}\) Many of the objects manipulated by function signatures declared in <cuchar> are char16-character sequences or NTC16s.\(^{171}\) A char16_t string literal, such as \u"abc\", is a static NTC16s.
A null-terminated char32-character string, or NTC32s, is a char32-character sequence whose highest-addressed element with defined content has the value zero.\footnote{172)}

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.

The value of an NTC32s is the sequence of values of the elements up to and including the terminating null character.

A static NTC32s is an NTC32s with static storage duration.\footnote{173)}

\subsection{17.5.2.1.4.5 Wide-character sequences} \footnote{[wide.characters]}

A wide-character sequence is an array object (8.3.4) \(A\) that can be declared as \(T\ A[N]\), where \(T\) is type wchar_t (3.9.1), optionally qualified by any combination of \texttt{const} or \texttt{volatile}. The initial elements of the array have defined contents up to and including an element determined by some predicate. A wide-character sequence can be designated by a pointer value \(S\) that designates its first element.

A null-terminated wide-character string, or NTWCS, is a wide-character sequence whose highest-addressed element with defined content has the value zero.\footnote{174)}

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.

The value of an NTWCS is the sequence of values of the elements up to and including the terminating null character.

A static NTWCS is an NTWCS with static storage duration.\footnote{175)}

\subsection{17.5.2.2 Functions within classes} \footnote{[functions.within.classes]}

For the sake of exposition, Clauses 18 through 30 and Annex D do not describe copy/move constructors, assignment operators, or (non-virtual) destructors with the same apparent semantics as those that can be generated by default (12.1, 12.4, 12.8).

It is unspecified whether the implementation provides explicit definitions for such member function signatures, or for virtual destructors that can be generated by default.

\subsection{17.5.2.3 Private members} \footnote{[objects.within.classes]}

Clauses 18 through 30 and Annex D do not specify the representation of classes, and intentionally omit specification of class members (9.2). An implementation may define static or non-static class members, or both, as needed to implement the semantics of the member functions specified in Clauses 18 through 30 and Annex D.

Objects of certain classes are sometimes required by the external specifications of their classes to store data, apparently in member objects. For the sake of exposition, some subclauses provide representative declarations, and semantic requirements, for private member objects of classes that meet the external specifications of the classes. The declarations for such member objects and the definitions of related member types are followed by a comment that ends with \textit{exposition only}, as in:

\begin{verbatim}
    streambuf* sb; // exposition only
\end{verbatim}

\footnote{172)} Many of the objects manipulated by function signatures declared in \texttt{<cuchar>} are char32-character sequences or NTC32s.

\footnote{173)} A char32_t string literal, such as U"abc", is a static NTC32s.

\footnote{174)} Many of the objects manipulated by function signatures declared in \texttt{<cwchar>} are wide-character sequences or NTWCSs.

\footnote{175)} A wide string literal, such as L"abc" is a static NTWCS.
An implementation may use any technique that provides equivalent external behavior.

17.6 Library-wide requirements

This subclause specifies requirements that apply to the entire C++ standard library. Clauses 18 through 30 and Annex D specify the requirements of individual entities within the library.

2 Requirements specified in terms of interactions between threads do not apply to programs having only a single thread of execution.

3 Within this subclause, 17.6.1 describes the library’s contents and organization, 17.6.2 describes how well-formed C++ programs gain access to library entities, 17.6.3 describes constraints on well-formed C++ programs, and 17.6.4 describes constraints on conforming implementations.

17.6.1 Library contents and organization

1 17.6.1.1 describes the entities defined in the C++ standard library. 17.6.1.2 lists the standard library headers and some constraints on those headers. 17.6.1.3 lists requirements for a freestanding implementation of the C++ standard library.

17.6.1.1 Library contents

1 The C++ standard library provides definitions for the following types of entities: macros, values, types, templates, classes, functions, objects.

2 All library entities except macros, operator new and operator delete are defined within the namespace std or namespaces nested within namespace std. It is unspecified whether names declared in a specific namespace are declared directly in that namespace or in an inline namespace inside that namespace.

3 Whenever a name \( x \) defined in the standard library is mentioned, the name \( x \) is assumed to be fully qualified as ::std::x, unless explicitly described otherwise. For example, if the Effects section for library function \( F \) is described as calling library function \( G \), the function ::std::G is meant.

17.6.1.2 Headers

1 Each element of the C++ standard library is declared or defined (as appropriate) in a header.

2 The C++ standard library provides 52 C++ library headers, as shown in Table 13.

3 The facilities of the C standard Library are provided in 26 additional headers, as shown in Table 14.

4 Except as noted in Clauses 18 through 30 and Annex D, the contents of each header \( \text{cname} \) shall be the same as that of the corresponding header \( \text{name.h} \), as specified in the C standard library (1.2) or the C Unicode TR, as appropriate, as if by inclusion. In the C++ standard library, however, the declarations (except for names which are defined as macros in C) are within namespace scope (3.3.6) of the namespace std. It is unspecified whether these names are first declared within the global namespace scope and are then injected into namespace std by explicit using-declarations (7.3.3).

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]

---

176) The C standard library headers (Annex D.6) also define names within the global namespace, while the C++ headers for C library facilities (17.6.1.2) may also define names within the global namespace.

177) This gives implementors freedom to use inline namespaces to support multiple configurations of the library.

178) 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).
Table 13 — C++ library headers

<table>
<thead>
<tr>
<th>&lt;algorithm&gt;</th>
<th>&lt;fstream&gt;</th>
<th>&lt;list&gt;</th>
<th>&lt;regex&gt;</th>
<th>&lt;typeindex&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;array&gt;</td>
<td>&lt;functional&gt;</td>
<td>&lt;locale&gt;</td>
<td>&lt;set&gt;</td>
<td>&lt;typeinfo&gt;</td>
</tr>
<tr>
<td>&lt;atomic&gt;</td>
<td>&lt;future&gt;</td>
<td>&lt;map&gt;</td>
<td>&lt;sstream&gt;</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>&lt;bitset&gt;</td>
<td>&lt;initializer_list&gt;</td>
<td>&lt;memory&gt;</td>
<td>&lt;stack&gt;</td>
<td>&lt;unordered_map&gt;</td>
</tr>
<tr>
<td>&lt;chrono&gt;</td>
<td>&lt;iomanip&gt;</td>
<td>&lt;mutex&gt;</td>
<td>&lt;stdexcept&gt;</td>
<td>&lt;unordered_set&gt;</td>
</tr>
<tr>
<td>&lt;codecvt&gt;</td>
<td>&lt;ios&gt;</td>
<td>&lt;new&gt;</td>
<td>&lt;streambuf&gt;</td>
<td>&lt;utility&gt;</td>
</tr>
<tr>
<td>&lt;complex&gt;</td>
<td>&lt;iosfwd&gt;</td>
<td>&lt;numeric&gt;</td>
<td>&lt;string&gt;</td>
<td>&lt;valarray&gt;</td>
</tr>
<tr>
<td>&lt;condition_variable&gt;</td>
<td>&lt;iostream&gt;</td>
<td>&lt;ostream&gt;</td>
<td>&lt;sstream&gt;</td>
<td>&lt;vector&gt;</td>
</tr>
<tr>
<td>&lt;deque&gt;</td>
<td>&lt;istream&gt;</td>
<td>&lt;queue&gt;</td>
<td>&lt;system_error&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;exception&gt;</td>
<td>&lt;iterator&gt;</td>
<td>&lt;random&gt;</td>
<td>&lt;thread&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;forward_list&gt;</td>
<td>&lt;limits&gt;</td>
<td>&lt;ratio&gt;</td>
<td>&lt;tuple&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Table 14 — C++ headers for C library facilities

<table>
<thead>
<tr>
<th>&lt;cassert&gt;</th>
<th>&lt;cfloat&gt;</th>
<th>&lt;cmath&gt;</th>
<th>&lt;cstddef&gt;</th>
<th>&lt;ctgmath&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;ccomplex&gt;</td>
<td>&lt;cinttypes&gt;</td>
<td>&lt;ctjmp&gt;</td>
<td>&lt;cstint&gt;</td>
<td>&lt;ctime&gt;</td>
</tr>
<tr>
<td>&lt;ctype&gt;</td>
<td>&lt;ciso646&gt;</td>
<td>&lt;csignal&gt;</td>
<td>&lt;csdio&gt;</td>
<td>&lt;cuchar&gt;</td>
</tr>
<tr>
<td>&lt;cerrno&gt;</td>
<td>&lt;climits&gt;</td>
<td>&lt;cstdarg&gt;</td>
<td>&lt;cstdlib&gt;</td>
<td>&lt;cwchar&gt;</td>
</tr>
<tr>
<td>&lt;cfenv&gt;</td>
<td>&lt;cstdlib&gt;</td>
<td>&lt;cstring&gt;</td>
<td>&lt;cwctype&gt;</td>
<td></td>
</tr>
</tbody>
</table>

6 Names that are defined as functions in C shall be defined as functions in the C++ standard library.¹⁷⁹

7 Identifiers that are keywords or operators in C++ shall not be defined as macros in C++ standard library headers.¹⁸⁰

8 D.6, C standard library headers, describes the effects of using the name.h (C header) form in a C++ program.¹⁸¹

17.6.1.3 Freestanding implementations [compliance]

1 Two kinds of implementations are defined: hosted and freestanding (1.4). For a hosted implementation, this International Standard describes the set of available headers.

2 A freestanding implementation has an implementation-defined set of headers. This set shall include at least the headers shown in Table 15.

3 The supplied version of the header <cstdlib> shall declare at least the functions abort, atexit, at_quick_exit, exit, and quick_exit (18.5). The supplied version of the header <thread> shall meet the same requirements as for a hosted implementation or including it shall have no effect. The other headers listed in this table shall meet the same requirements as for a hosted implementation.

17.6.2 Using the library [using]

17.6.2.1 Overview [using.overview]

This section describes how a C++ program gains access to the facilities of the C++ standard library. 17.6.2.2

¹⁷⁹) 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.

¹⁸⁰) In particular, including the standard header <iso646.h> or <ciso646> has no effect.

¹⁸¹) The "h" headers dump all their names into the global namespace, whereas the newer forms keep their names in namespace std. Therefore, the newer forms are the preferred forms for all uses except for C++ programs which are intended to be strictly compatible with C.
Table 15 — C++ headers for freestanding implementations

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.2 Types</td>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
<tr>
<td>18.3 Implementation properties</td>
<td><code>&lt;limits&gt;</code></td>
</tr>
<tr>
<td>18.5 Start and termination</td>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
<tr>
<td>18.6 Dynamic memory management</td>
<td><code>&lt;new&gt;</code></td>
</tr>
<tr>
<td>18.7 Type identification</td>
<td><code>&lt;typeinfo&gt;</code></td>
</tr>
<tr>
<td>18.8 Exception handling</td>
<td><code>&lt;exception&gt;</code></td>
</tr>
<tr>
<td>18.9 Initializer lists</td>
<td><code>&lt;initializer_list&gt;</code></td>
</tr>
<tr>
<td>18.10 Other runtime support</td>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
<tr>
<td>20.7 Type traits</td>
<td><code>&lt;type_traits&gt;</code></td>
</tr>
<tr>
<td>30.3 Threads</td>
<td><code>&lt;thread&gt;</code></td>
</tr>
</tbody>
</table>

describes effects during translation phase 4, while 17.6.2.3 describes effects during phase 8 (2.2).

17.6.2.2 Headers

1. The entities in the C++ standard library are defined in headers, whose contents are made available to a translation unit when it contains the appropriate `#include` preprocessing directive (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`.182

3. A translation unit shall include a header only outside of any external declaration or definition, and shall include the header lexically before the first reference in that translation unit to any of the entities declared in that header.

17.6.2.3 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.183

3. Objects and functions defined in the library and required by a C++ program are included in the program prior to program startup.

See also: replacement functions (17.6.3.6), run-time changes (17.6.3.7).

17.6.3 Constraints on programs

1. This section describes restrictions on C++ programs that use the facilities of the C++ standard library. The following subclauses specify constraints on the program’s use of namespaces (17.6.3.2.1), its use of various reserved names (17.6.3.3), its use of headers (17.6.3.4), its use of standard library classes as base

182) This is the same as the Standard C library.
183) The only reliable way to declare an object or function signature from the Standard C library is by including the header that declares it, notwithstanding the latitude granted in 7.1.7 of the C Standard.

§ 17.6.3.1
classes (17.6.3.5), its definitions of replacement functions (17.6.3.6), and its installation of handler functions during execution (17.6.3.7).

17.6.3.2 Namespace use

17.6.3.2.1 Namespace std

1 The behavior of a C++ program is undefined if it adds declarations or definitions to namespace std or to a namespace within namespace std unless otherwise specified. A program may add a template specialization for any standard library template to namespace std only if the declaration depends on a user-defined type and the specialization meets the standard library requirements for the original template and is not explicitly prohibited.\(^\text{184}\)

2 The behavior of a C++ program is undefined if it declares

— 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 a template defined in the standard library only if the declaration depends on the name of a user-defined type and the instantiation meets the standard library requirements for the original template.

3 A translation unit shall not declare namespace std to be an inline namespace (7.3.1).

17.6.3.2.2 Namespace posix

1 The behavior of a C++ program is undefined if it adds declarations or definitions to namespace posix or to a namespace within namespace posix unless otherwise specified. The namespace posix is reserved for use by ISO/IEC 9945 and other POSIX standards.

17.6.3.3 Reserved names

1 The C++ standard library reserves the following kinds of names:

— macros
— global names
— names with external linkage

2 If a program declares or defines a name in a context where it is reserved, other than as explicitly allowed by this Clause, its behavior is undefined.

17.6.3.3.1 Macro names

1 A translation unit that includes a standard library header shall not \#define or \#undef names declared in any standard library header.

\(^{184}\) Any library code that instantiates other library templates must be prepared to work adequately with any user-supplied specialization that meets the minimum requirements of the Standard.
A translation unit shall not \#define or \#undef names lexically identical to keywords.

### 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 (2.12) 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.

### 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,\(^\text{185}\) both in namespace std and in the global namespace.

Each global function signature declared with external linkage in a header is reserved to the implementation to designate that function signature with external linkage.\(^\text{186}\)

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,\(^\text{187}\) or as a name of namespace scope in the global namespace.

### Types

For each type T from the Standard C library,\(^\text{188}\) the types ::T and std::T are reserved to the implementation and, when defined, ::T shall be identical to std::T.

### User-defined literal suffixes

Literal suffix identifiers that do not start with an underscore are reserved for future standardization.

### Headers

If a file with a name equivalent to the derived file name for one of the C++ standard library headers is not provided as part of the implementation, and a file with that name is placed in any of the standard places for a source file to be included (16.2), the behavior is undefined.

### Derived classes

Virtual member function signatures defined for a base class in the C++ standard library may be overridden in a derived class defined in the program (10.3).

### Replacement functions

Clauses 18 through 30 and Annex D describe the behavior of numerous functions defined by the C++ standard library. Under some circumstances, however, certain of these function descriptions also apply to replacement

\(^{185}\) The list of such reserved names includes errno, declared or defined in <cerrno>.

\(^{186}\) The list of such reserved function signatures with external linkage includes setjmp(jmp_buf), declared or defined in <csetjmp>, and va_end(va_list), declared or defined in <cstdarg>.

\(^{187}\) The function signatures declared in <cuchar>, <wchar>, and <cwctype> are always reserved, notwithstanding the restrictions imposed in subclause 4.5.1 of Amendment 1 to the C Standard for these headers.

\(^{188}\) These types are clock_t, div_t, FILE, fpos_t, lconv, ldiv_t, mbstate_t, ptdiff_t, sig_atomic_t, size_t, time_t, tm, va_list, wctrans_t, wctype_t, and wint_t.
functions defined in the program (17.3).

2 A C++ program may provide the definition for any of eight dynamic memory allocation function signatures declared in header `<new>` (3.7.4, Clause 18):

   - `operator new(std::size_t)`
   - `operator new(std::size_t, const std::nothrow_t&)`
   - `operator new[](std::size_t)`
   - `operator new[](std::size_t, const std::nothrow_t&)`
   - `operator delete(void*)`
   - `operator delete(void*, const std::nothrow_t&)`
   - `operator delete[](void*)`
   - `operator delete[](void*, const std::nothrow_t&)`

3 The program’s definitions are used instead of the default versions supplied by the implementation (18.6). Such replacement occurs prior to program startup (3.2, 3.6). The program’s definitions shall not be specified as `inline`. No diagnostic is required.

17.6.3.7 Handler functions

1 The C++ standard library provides default versions of the following handler functions (Clause 18):
   - `unexpected_handler`
   - `terminate_handler`

2 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 18.6.2, Storage allocation errors, and 18.8, Exception handling.

17.6.3.8 Other functions

1 In certain cases (replacement functions, handler functions, operations on types used to instantiate standard library template components), the C++ standard library depends on components supplied by a C++ program. If these components do not meet their requirements, the Standard places no requirements on the implementation.

2 In particular, the effects are undefined in the following cases:
   - for replacement functions (18.6.1), if the installed replacement function does not implement the semantics of the applicable `Required behavior: paragraph`
   - for handler functions (18.6.2.3, 18.8.3.1, 18.8.2.2), 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 (20.2.5, 23.2,
24.2, 26.2). 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 exits via 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.

17.6.3.9 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.

— If a function argument binds to an rvalue reference parameter, the implementation may assume that
this parameter is a unique reference to this argument. [Note: If the parameter is a generic parameter of
the form T&& and an lvalue of type A is bound, the argument binds to an lvalue reference (14.8.2.1) and
thus is not covered by the previous sentence. — end note] [Note: If a program casts an lvalue to an
rvalue while passing that lvalue to a library function (e.g. by calling the function with the argument
move(x)), the program is effectively asking that function to treat that lvalue as a temporary. The
implementation is free to optimize away aliasing checks which might be needed if the argument was
an lvalue. — end note]

17.6.3.10 Shared objects and the library

The behavior of a program is undefined if calls to standard library functions from different threads may
introduce a data race. The conditions under which this may occur are specified in 17.6.4.8. [Note: Modifying
an object of a standard library type that is shared between threads risks undefined behavior unless objects
of that type are explicitly specified as being sharable without data races or the user supplies a locking
mechanism. — end note]

[Note: In particular, the program is required to ensure that completion of the constructor of any object of
a class type defined in the standard library happens before any other member function invocation on that
object and, unless otherwise specified, to ensure that completion of any member function invocation other
than destruction on such an object happens before destruction of that object. This applies even to objects
such as mutexes intended for thread synchronization. — end note]

17.6.3.11 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.

17.6.4 Conforming implementations

17.6.4.1 Overview

This section describes the constraints upon, and latitude of, implementations of the C++ standard library.

17.6.4.2 Headers

A C++ header may include other C++ headers. A C++ header shall provide the declarations and definitions that appear in its synopsis. A C++ header shown in its synopsis as including other C++ headers shall provide the declarations and definitions that appear in the synopses of those other headers.

Certain types and macros are defined in more than one header. Every such entity shall be defined such that any header that defines it may be included after any other header that also defines it (3.2).

The C standard headers (D.6) shall include only their corresponding C++ standard header, as described in 17.6.1.2.

17.6.4.3 Restrictions on macro definitions

The names and global function signatures described in 17.6.1.1 are reserved to the implementation.

All object-like macros defined by the C standard library and described in this Clause as expanding to integral constant expressions are also suitable for use in #if preprocessing directives, unless explicitly stated otherwise.

17.6.4.4 Global and 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).

A call to a global or non-member function signature described in Clauses 18 through 30 and Annex D shall behave as if the implementation declared no additional global or non-member function signatures.\(^\text{189}\)

An implementation shall not declare a global or non-member function signature with additional default arguments.

Unless otherwise specified, global and non-member functions in the standard library shall not use functions from another namespace which are found through argument-dependent name lookup (3.4.2). [Note: The phrase “unless otherwise specified” is intended to allow argument-dependent lookup in cases like that of ostream_iterators: Effects:

```cpp
*out_stream << value;
if (delim != 0)
    *out_stream << delim;
return (*this);
```

\(^{189}\) A valid C++ program always calls the expected library global or non-member function. An implementation may also define additional global or non-member functions that would otherwise not be called by a valid C++ program.
17.6.4.5 Member functions

1. It is unspecified whether any member functions in the C++ standard library are defined as inline (7.1.2).

2. An implementation may declare additional non-virtual member function signatures within a class:
   - by adding arguments with default values to a member function signature;\(^{190}\) \(\text{[Note: An implementation may not add arguments with default values to virtual, global, or non-member functions. — end note]}\)
   - by replacing a member function signature with default values by two or more member function signatures with equivalent behavior; and
   - 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 as if the implementation declares no additional member function signatures.\(^{191}\)

17.6.4.6 constexpr functions and constructors

1. Within any header that provides any non-defining declarations of constexpr functions or constructors an implementation shall provide corresponding definitions.

17.6.4.7 Reentrancy

1. Except where explicitly specified in this standard, it is implementation-defined which functions in the Standard C++ library may be recursively reentered.

17.6.4.8 Data race avoidance

1. This section specifies requirements that implementations shall meet to prevent data races (1.10). Every standard library function shall meet each requirement unless otherwise specified. Implementations may prevent data races in cases other than those specified below.

2. A C++ standard library function shall not directly or indirectly access objects (1.10) accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function’s arguments, including this.

3. A C++ standard library function shall not directly or indirectly modify objects (1.10) accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function’s non-const arguments, including this.

4. \(\text{[Note: This means, for example, that implementations can’t use a static object for internal purposes without synchronization because it could cause a data race even in programs that do not explicitly share objects between threads. — end note]}\)

5. A C++ standard library function shall not access objects indirectly accessible via its arguments or via elements of its container arguments except by invoking functions required by its specification on those container elements.

6. Operations on iterators obtained by calling a standard library container or string member function may access the underlying container, but shall not modify it. \(\text{[Note: In particular, container operations that invalidate iterators conflict with operations on iterators associated with that container. — end note]}\)

\(^{190}\) Hence, the address of a member function of a class in the C++ standard library has an unspecified type.

\(^{191}\) 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.
Implementations may share their own internal objects between threads if the objects are not visible to users and are protected against data races.

Unless otherwise specified, C++ standard library functions shall perform all operations solely within the current thread if those operations have effects that are visible (1.10) to users.

[Note: This allows implementations to parallelize operations if there are no visible side effects. — end note]

### 17.6.4.9 Protection within classes [protection.within.classes]

It is unspecified whether any function signature or class described in Clauses 18 through 30 and Annex D is a friend of another class in the C++ standard library.

### 17.6.4.10 Derived classes [derivation]

An implementation may derive any class in the C++ standard library from a class with a name reserved to the implementation.

Certain classes defined in the C++ standard library are required to be derived from other classes in the C++ standard library. An implementation may derive such a class directly from the required base or indirectly through a hierarchy of base classes with names reserved to the implementation.

In any case:

- Every base class described as virtual shall be virtual;
- Every base class described as non-virtual shall not be virtual;
- Unless explicitly stated otherwise, types with distinct names shall be distinct types.\(^\text{192}\)

### 17.6.4.11 Restrictions on exception handling [res.on.exception.handling]

Any of the functions defined in the C++ standard library can report a failure by throwing an exception of a type described in its Throws: paragraph or its exception-specification (15.4). An implementation may strengthen the exception-specification for a non-virtual function by removing listed exceptions.\(^\text{193}\)

A function may throw an object of a type not listed in its Throws clause if its type is derived from a type named in the Throws clause and would be caught by an exception handler for the base type.

Functions from the C standard library shall not throw exceptions\(^\text{194}\) except when such a function calls a program-supplied function that throws an exception.\(^\text{195}\)

Destructor operations defined in the C++ standard library shall not throw exceptions. Any other functions defined in the C++ standard library that do not have an exception-specification may throw implementation-

---

\(^{192}\) There is an implicit exception to this rule for types that are described as synonyms for basic integral types, such as `size_t` (18.2) and `streamoff` (27.5.1).

\(^{193}\) 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.

\(^{194}\) That is, the C library functions can all be treated as if they have a `throw()` exception-specification. This allows implementations to make performance optimizations based on the absence of exceptions at runtime.

\(^{195}\) The functions `qsort()` and `bsearch()` (25.5) meet this condition.
defined exceptions unless otherwise specified. An implementation may strengthen this implicit exception-specification by adding an explicit one.

17.6.4.12 Restrictions on storage of pointers

Objects constructed by the standard library that may hold a user-supplied pointer value or an integer of type `std::intptr_t` shall store such values in a traceable pointer location (3.7.4.3). [Note: Other libraries are strongly encouraged to do the same, since not doing so may result in accidental use of pointers that are not safely derived. Libraries that store pointers outside the user’s address space should make it appear that they are stored and retrieved from a traceable pointer location. — end note]

17.6.4.13 Value of error codes

Certain functions in the C++ standard library report errors via a `std::error_code` (19.5.2.1) object. That object’s `category()` member shall return `std::system_category()` for errors originating from the operating system, or a reference to an implementation-defined `error_category` object for errors originating elsewhere. The implementation shall define the possible values of `value()` for each of these error categories. [Example: For operating systems that are based on POSIX, implementations are encouraged to define the `std::system_category()` values as identical to the POSIX `errno` values, with additional values as defined by the operating system’s documentation. Implementations for operating systems that are not based on POSIX are encouraged to define values identical to the operating system’s values. For errors that do not originate from the operating system, the implementation may provide enums for the associated values. — end example]
18 Language support library

18.1 General

This Clause describes the function signatures that are called implicitly, and the types of objects generated implicitly, during the execution of some C++ programs. It also describes the headers that declare these function signatures and define any related types.

The following subclauses describe common type definitions used throughout the library, characteristics of the predefined types, functions supporting start and termination of a C++ program, support for dynamic memory management, support for dynamic type identification, support for exception processing, support for initializer lists, and other runtime support, as summarized in Table 16.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.2 Types</td>
<td><code>&lt;cassert&gt;</code></td>
</tr>
<tr>
<td>18.3 Implementation properties</td>
<td><code>&lt;climits&gt;</code></td>
</tr>
<tr>
<td>18.4 Integer types</td>
<td><code>&lt;cstdint&gt;</code></td>
</tr>
<tr>
<td>18.5 Start and termination</td>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
<tr>
<td>18.6 Dynamic memory management</td>
<td><code>&lt;new&gt;</code></td>
</tr>
<tr>
<td>18.7 Type identification</td>
<td><code>&lt;typeinfo&gt;</code></td>
</tr>
<tr>
<td>18.8 Exception handling</td>
<td><code>&lt;exception&gt;</code></td>
</tr>
<tr>
<td>18.9 Initializer lists</td>
<td><code>&lt;initializer_list&gt;</code></td>
</tr>
<tr>
<td>18.10 Other runtime support</td>
<td><code>&lt;cstdarg&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;csetjmp&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;ctime&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;csignal&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;cstdbool&gt;</code></td>
</tr>
</tbody>
</table>

18.2 Types

Table 17 describes the header `<cassert>`.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros</td>
<td><code>NULL</code></td>
</tr>
<tr>
<td></td>
<td><code>offsetof</code></td>
</tr>
<tr>
<td>Types</td>
<td><code>ptrdiff_t</code></td>
</tr>
<tr>
<td></td>
<td><code>size_t</code></td>
</tr>
<tr>
<td></td>
<td><code>max_align_t</code></td>
</tr>
<tr>
<td></td>
<td><code>nullptr_t</code></td>
</tr>
</tbody>
</table>

The contents are the same as the Standard C library header `<cassert.h>`, with the following changes:
The macro `NULL` is an implementation-defined C++ null pointer constant in this International Standard (4.10).\textsuperscript{198} The macro `offsetof(type, member-designator)` accepts a restricted set of `type` arguments in this International Standard. If `type` is not a standard-layout class (Clause 9), the results are undefined.\textsuperscript{199} 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}\[198]\[199]

See also: Alignment (3.11), Sizeof (5.3.3), Additive operators (5.7), Free store (12.5), and ISO C 7.1.6.

18.3 Implementation properties [support.limits]

The headers `<limits>`, `<climits>`, `<cfloat>`, and `<cinttypes>` supply characteristics of implementation-dependent arithmetic types (3.9.1).

18.3.1 Numeric limits [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>` (26.4.2), shall not have specializations.

Header `<limits>` synopsis

```cpp
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>;
```

\[198]\textsuperscript{198} Possible definitions include 0 and 0L, but not `(void*)0`.
\[199]\textsuperscript{199} Note that `offsetof` is required to work as specified even if unary `operator&` is overloaded for any of the types involved.
template<> class numeric_limits<wchar_t>;

template<> class numeric_limits<short>;
template<> class numeric_limits<int>;
template<> class numeric_limits<long>;
template<> class numeric_limits<long long>;
template<> class numeric_limits<unsigned short>;
template<> class numeric_limits<unsigned int>;
template<> class numeric_limits<unsigned long>;
template<> class numeric_limits<unsigned long long>;

template<> class numeric_limits<float>;
template<> class numeric_limits<double>;
template<> class numeric_limits<long double>;
}

§ 18.3.1.1 434
static constexpr bool traps = false;
static constexpr bool tinyness_before = false;
static constexpr float_round_style round_style = round_toward_zero;
};

template<class T> class numeric_limits<const T>;
template<class T> class numeric_limits<volatile T>;
template<class T> class numeric_limits<const volatile T>;

1 The default numeric_limits<T> template shall have all members, but with 0 or false values.
2 The value of each member of a specialization of numeric_limits on a cv-qualified type cv T shall be equal to the value of the corresponding member of the specialization on the unqualified type T.

18.3.1.2 numeric_limits members

static constexpr T min() throw();
1 Minimum finite value.\textsuperscript{200}
2 For floating types with denormalization, returns the minimum positive normalized value.
3 Meaningful for all specializations in which is_bounded != false, or is_bounded == false &\& is_-signed == false.

static constexpr T max() throw();
4 Maximum finite value.\textsuperscript{201}
5 Meaningful for all specializations in which is_bounded != false.

static constexpr T lowest() throw();
6 A finite value x such that there is no other finite value y where y < x.\textsuperscript{202}
7 Meaningful for all specializations in which is_bounded != false.

static constexpr int digits;
8 Number of radix digits that can be represented without change.
9 For integer types, the number of non-sign bits in the representation.
10 For floating point types, the number of radix digits in the mantissa.\textsuperscript{203}

static constexpr int digits10;
11 Number of base 10 digits that can be represented without change.\textsuperscript{204}
12 Meaningful for all specializations in which is_bounded != false.

static constexpr int max_digits10;

\textsuperscript{200} Equivalent to CHAR_MIN, SHRT_MIN, FLT_MIN, DBL_MIN, etc.
\textsuperscript{201} Equivalent to CHAR_MAX, SHRT_MAX, FLT_MAX, DBL_MAX, etc.
\textsuperscript{202} 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.
\textsuperscript{203} Equivalent to FLT_MANT_DIG, DBL_MANT_DIG, LDBL_MANT_DIG.
\textsuperscript{204} Equivalent to FLT_DIG, DBL_DIG, LDBL_DIG.

§ 18.3.1.2
Number of base 10 digits required to ensure that values which differ are always differentiated.

Meaningful for all floating point types.

```cpp
static constexpr bool is_signed;
```

True if the type is signed.

Meaningful for all specializations.

```cpp
static constexpr bool is_integer;
```

True if the type is integer.

Meaningful for all specializations.

```cpp
static constexpr bool is_exact;
```

True if the type uses an exact representation. All integer types are exact, but not all exact types are integer. For example, rational and fixed-exponent representations are exact but not integer.

Meaningful for all specializations.

```cpp
static constexpr int radix;
```

For floating types, specifies the base or radix of the exponent representation (often 2).

For integer types, specifies the base of the representation.

Meaningful for all specializations.

```cpp
static constexpr T epsilon() throw();
```

Machine epsilon: the difference between 1 and the least value greater than 1 that is representable.

Meaningful for all floating point types.

```cpp
static constexpr T round_error() throw();
```

Measure of the maximum rounding error.

```cpp
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.

Meaningful for all floating point types.

```cpp
static constexpr int min_exponent10;
```

Minimum negative integer such that 10 raised to that power is in the range of normalized floating point numbers.

Meaningful for all floating point types.

```cpp
static constexpr int max_exponent;
```

Meaningful for all floating point types.

---

205) Equivalent to `FLT_RADIX`.

206) Distinguishes types with bases other than 2 (e.g. BCD).

207) Equivalent to `FLT_EPSILON`, `DBL_EPSILON`, `LDBL_EPSILON`.

208) 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.

209) Equivalent to `FLT_MIN_EXP`, `DBL_MIN_EXP`, `LDBL_MIN_EXP`.

210) Equivalent to `FLT_MIN_10_EXP`, `DBL_MIN_10_EXP`, `LDBL_MIN_10_EXP`.

§ 18.3.1.2
Maximum positive integer such that \( \text{radix} \) raised to the power one less than that integer is a representable finite floating point number.\(^{211}\)

Meaningful for all floating point types.

```cpp
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.\(^{212}\)

Meaningful for all floating point types.

```cpp
static constexpr bool has_infinity;
```

True if the type has a representation for positive infinity.

Meaningful for all floating point types.

```cpp
static constexpr bool has_quiet_NaN;
```

True if the type has a representation for a quiet (non-signaling) "Not a Number."\(^{213}\)

Meaningful for all floating point types.

```cpp
static constexpr bool has_signaling_NaN;
```

True if the type has a representation for a signaling "Not a Number."\(^{214}\)

Meaningful for all floating point types.

```cpp
static constexpr float_denorm_style has_denorm;
```

denorm_present if the type allows denormalized values (variable number of exponent bits)\(^{215}\), 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.\(^{216}\)

```cpp
static constexpr T infinity() throw();
```

Representation of positive infinity, if available.\(^{217}\)

Meaningful for all specializations for which has_infinity != false. Required in specializations for which is_iec559 != false.

```cpp
static constexpr T quiet_NaN() throw();
```

Equivalent to FLT_MAX_EXP, DBL_MAX_EXP, LDBL_MAX_EXP.

\(^{211}\) Equivalent to FLT_MAX_EXP, DBL_MAX_EXP, LDBL_MAX_EXP.

\(^{212}\) Equivalent to FLT_MAX_10_EXP, DBL_MAX_10_EXP, LDBL_MAX_10_EXP.

\(^{213}\) Required by LIA-1.

\(^{214}\) Required by LIA-1.

\(^{215}\) Required by LIA-1.

\(^{216}\) See IEC 559.

\(^{217}\) Required by LIA-1.
Representation of a quiet “Not a Number,” if available.\(^\text{218}\)

Meaningful for all specializations for which \texttt{has\_quiet\_NaN} \(!= \texttt{false}\). Required in specializations for which \texttt{is\_iec559} \(!= \texttt{false}\).

\begin{verbatim}
static constexpr T signaling_NaN() throw();
\end{verbatim}

Representation of a signaling “Not a Number,” if available.\(^\text{219}\)

Meaningful for all specializations for which \texttt{has\_signaling\_NaN} \(!= \texttt{false}\). Required in specializations for which \texttt{is\_iec559} \(!= \texttt{false}\).

\begin{verbatim}
static constexpr T denorm_min() throw();
\end{verbatim}

Minimum positive denormalized value.\(^\text{220}\)

Meaningful for all floating point types.

In specializations for which \texttt{has\_denorm} \(!= \texttt{false}\), returns the minimum positive normalized value.

\begin{verbatim}
static constexpr bool is_iec559;
\end{verbatim}

True if and only if the type adheres to IEC 559 standard.\(^\text{221}\)

Meaningful for all floating point types.

\begin{verbatim}
static constexpr bool is_bounded;
\end{verbatim}

True if the set of values representable by the type is finite.\(^\text{222}\) [\textit{Note: All built-in types are bounded. This member would be false for arbitrary precision types. — end note}]\(\text{222}\)

Meaningful for all specializations.

\begin{verbatim}
static constexpr bool is_modulo;
\end{verbatim}

True if the type is modulo.\(^\text{223}\) A type is modulo if, for any operation involving +, −, or \(*\) on values of that type whose result would fall outside the range \([\min(), \max()]\), the value returned differs from the true value by an integer multiple of \(\max() - \min() + 1\).

On most machines, this is \texttt{false} for floating types, \texttt{true} for unsigned integers, and \texttt{true} for signed integers.

Meaningful for all specializations.

\begin{verbatim}
static constexpr bool traps;
\end{verbatim}

\texttt{true} if, at program startup, there exists a value of the type that would cause an arithmetic operation using that value to trap.\(^\text{224}\)

Meaningful for all specializations.

\begin{verbatim}
static constexpr bool tinyness_before;
\end{verbatim}

---

\(^{218}\) Required by LIA-1.

\(^{219}\) Required by LIA-1.

\(^{220}\) International Electrotechnical Commission standard 559 is the same as IEEE 754.

\(^{221}\) Required by LIA-1.

\(^{222}\) Required by LIA-1.

\(^{223}\) Required by LIA-1.

\(^{224}\) Required by LIA-1.
true if tinyness is detected before rounding.\footnote{Refer to IEC 559. Required by LIA-1.}

Meaningful for all floating point types.

```cpp
static constexpr float_round_style round_style;
```

The rounding style for the type.\footnote{Equivalent to FLT_ROUNDS. Required by LIA-1.}

Meaningful for all floating point types. Specializations for integer types shall return `round_toward_zero`.

\section*{18.3.1.3 Type float_round_style [round.style]}

```cpp
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

\section*{18.3.1.4 Type float_denorm_style [denorm.style]}

```cpp
namespace std {
    enum float_denorm_style {
        denorm_indeterminate = -1,
        denorm_absent = 0,
        denorm_present = 1
    };
}
```

The presence or absence of denormalization (variable number of exponent bits) is characterized by the values:

- `denorm_indeterminate` if it cannot be determined whether or not the type allows denormalized values
- `denorm_absent` if the type does not allow denormalized values
- `denorm_present` if the type does allow denormalized values

\footnote{225) Refer to IEC 559. Required by LIA-1.}

\footnote{226) Equivalent to FLT_ROUNDS. Required by LIA-1.}
18.3.1.5 numeric_limits specializations

1 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.

2 [Example:

```cpp
namespace std {
    template<> class numeric_limits<float> {
        public:
            static constexpr bool is_specialized = true;
            inline static constexpr float min() throw() { return 1.17549435E-38F; }
            inline static constexpr float max() throw() { return 3.40282347E+38F; }
            inline static constexpr float lowest() throw() { return -3.40282347E+38F; }

            static constexpr int digits = 24;
            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:

```cpp
namespace std {
    template<> class numeric_limits<bool> {
    public:
        static constexpr bool is_specialized = true;
        static constexpr bool min() throw() { return false; }
        static constexpr bool max() throw() { return true; }
        static constexpr bool lowest() throw() { return false; }

        static constexpr int digits = 1;
        static constexpr int digits10 = 0;
        static constexpr int max_digits10 = 0;

        static constexpr bool is_signed = false;
        static constexpr bool is_integer = true;
        static constexpr bool is_exact = true;
        static constexpr int radix = 2;
        static constexpr bool epsilon() throw() { return 0; }
        static constexpr bool round_error() throw() { return 0; }

        static constexpr int min_exponent = 0;
        static constexpr int min_exponent10 = 0;
        static constexpr int max_exponent = 0;
        static constexpr int max_exponent10 = 0;

        static constexpr bool has_infinity = false;
        static constexpr bool has_quiet_NaN = false;
        static constexpr bool has_signaling_NaN = false;
        static constexpr float_denorm_style has_denorm = denorm_absent;
        static constexpr bool has_denorm_loss = false;
        static constexpr bool infinity() throw() { return 0; }
        static constexpr bool quiet_NaN() throw() { return 0; }
        static constexpr bool signaling_NaN() throw() { return 0; }
        static constexpr bool denorm_min() throw() { return 0; }

        static constexpr bool is_iec559 = false;
        static constexpr bool is_bounded = true;
        static constexpr bool is_modulo = false;

        static constexpr bool traps = false;
        static constexpr bool tinyness_before = false;
        static constexpr float_round_style round_style = round_toward_zero;
    };
}
```

18.3.2 C Library
[c.limits]

1 Table 18 describes the header `<climits>`.

2 The contents are the same as the Standard C library header `<limits.h>`. [Note: The types of the constants defined by macros in `<climits>` are not required to match the types to which the macros refer. — end note]

3 Table 19 describes the header `<cfloat>`.

4 The contents are the same as the Standard C library header `<float.h>`.
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>CHAR_MAX</td>
<td>LLONG_MAX</td>
</tr>
<tr>
<td>CHAR_MIN</td>
<td>LONGLONG_MIN</td>
</tr>
<tr>
<td>INT_MIN</td>
<td></td>
</tr>
</tbody>
</table>

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>DBL_EPSILON</td>
<td>DECIMAL_DIG</td>
</tr>
<tr>
<td>DBL_MANT_DIG</td>
<td>FLT_DIG</td>
</tr>
<tr>
<td>DBL_MAX</td>
<td>FLT_EPSILON</td>
</tr>
<tr>
<td>DBL_MAX_10_EXP</td>
<td>FLT_EVAL_METHOD</td>
</tr>
<tr>
<td>DBL_MAX_EXP</td>
<td>FLT_MANT_DIG</td>
</tr>
<tr>
<td>DBL_MIN</td>
<td>FLT_MAX</td>
</tr>
<tr>
<td>DBL_MIN_10_EXP</td>
<td>FLT_MAX_EXP</td>
</tr>
</tbody>
</table>

See also: ISO C 7.1.5, 5.2.4.2.2, 5.2.4.2.1.

18.4 Integer types

18.4.1 Header `<cstdint>` synopsis

```c
namespace std {
    typedef signed integer type int8_t; // optional
    typedef signed integer type int16_t; // optional
    typedef signed integer type int32_t; // optional
    typedef signed integer type int64_t; // optional
    typedef signed integer type int_fast8_t;
    typedef signed integer type int_fast16_t;
    typedef signed integer type int_fast32_t;
    typedef signed integer type int_fast64_t;
    typedef signed integer type int_least8_t;
    typedef signed integer type int_least16_t;
    typedef signed integer type int_least32_t;
    typedef signed integer type int_least64_t;
    typedef signed integer type intmax_t;
    typedef signed integer type intptr_t; // optional
    typedef unsigned integer type uint8_t; // optional
    typedef unsigned integer type uint16_t; // optional
    typedef unsigned integer type uint32_t; // optional
    typedef unsigned integer type uint64_t; // optional
    typedef unsigned integer type uint_fast8_t;
}
```

§ 18.4.1
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;

// namespace std

1 The header also defines numerous macros of the form:

```
INT_[FAST LEAST]{8 16 32 64}_MIN
[U]INT_[FAST LEAST]{8 16 32 64}_MAX
INT[MAX PTR]_MIN
[U]INT[MAX PTR]_MAX
{PTRDIFF SIG_ATOMIC WCHAR WINT}{_MAX _MIN}
```

plus function macros of the form:

```
[U]INT{8 16 32 64 MAX}_C
```

2 The header defines all functions, types, and macros the same as 7.18 in the C standard. [Note: The macros defined by <cstdint> are provided unconditionally. In particular, the symbols __STDC_LIMIT_MACROS and __STDC_CONSTANT_MACROS (mentioned in footnotes 219, 220, and 222 in the C standard) play no role in C++. — end note]

18.5 Start and termination

1 Table 20 describes some of the contents of the header <cstdlib>.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td>EXIT_FAILURE</td>
</tr>
<tr>
<td></td>
<td>EXIT_SUCCESS</td>
</tr>
<tr>
<td>Functions:</td>
<td>_Exit</td>
</tr>
<tr>
<td></td>
<td>abort</td>
</tr>
<tr>
<td></td>
<td>atexit</td>
</tr>
<tr>
<td></td>
<td>at_quick_exit</td>
</tr>
<tr>
<td></td>
<td>exit</td>
</tr>
<tr>
<td></td>
<td>quick_exit</td>
</tr>
</tbody>
</table>

2 The contents are the same as the Standard C library header <stdlib.h>, with the following changes:

```c
void _Exit [[noreturn]] (int status)
```

3 The function _Exit(int status) has additional behavior in this International Standard:

- The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling functions passed to atexit() (3.6.3).

```c
void abort [[noreturn]] (void)
```

4 The function abort() has additional behavior in this International Standard:

- The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling the functions passed to atexit() (3.6.3).
extern "C" int atexit(void (*)(void))(void))
extern "C++" int atexit(void (*)(void))(void))

Effects: The atexit() functions register the function pointed to by f to be called without arguments at normal program termination. It is unspecified whether a call to atexit() that does not happen before (1.10) a call to exit() will succeed. [Note: the atexit() functions shall not introduce a data race (17.6.4.8). — end note]

Implementation limits: The implementation shall support the registration of at least 32 functions.

Returns: The atexit() function returns zero if the registration succeeds, nozero if it fails.

void exit [[noreturn]] (int status)

The function exit() has additional behavior in this International Standard:
— First, objects with thread storage duration and associated with the current thread are destroyed.
— Next, objects with static storage duration are destroyed and functions registered by calling atexit are called. See 3.6.3 for the order of destructions and calls. (Automatic objects are not destroyed as a result of calling exit().)

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.
— Finally, control is returned to the host environment. If status is zero or EXIT_SUCCESS, an implementation-defined form of the status successful termination is returned. If status is EXIT_FAILURE, an implementation-defined form of the status unsuccessful termination is returned. Otherwise the status returned is implementation-defined.

The function exit() never returns to its caller.

extern "C" int at_quick_exit(void (*)(void))(void));
extern "C++" int at_quick_exit(void (*)(void))(void));

Effects: The at_quick_exit() functions register the function pointed to by f to be called without arguments when quick_exit is called. It is unspecified whether a call to at_quick_exit() that does not happen before (1.10) all calls to quick_exit will succeed. [Note: the at_quick_exit() functions do not introduce a data race (17.6.4.8). — end note] [Note: The at_quick_exit registrations are distinct from the atexit registrations, and applications may need to call both registration functions with the same argument. — end note]

Implementation limits: The implementation shall support the registration of at least 32 functions.

Returns: zero if the registration succeeds, non-zero if it fails.

void quick_exit [[noreturn]] (int status)

227) A function is called for every time it is registered.
228) 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().
229) The macros EXIT_FAILURE and EXIT_SUCCESS are defined in <cstdlib>.


13 Effects: Functions registered by calls to \texttt{at_quick_exit} are called in the reverse order of their registration, except that a function shall be called after any previously registered functions that had already been called at the time it was registered. Objects shall not be destroyed as a result of calling \texttt{quick_exit}. If control leaves a registered function called by \texttt{quick_exit} because the function does not provide a handler for a thrown exception, \texttt{terminate()} shall be called. [\textbf{Note:} \texttt{at_quick_exit} may call a registered function from a different thread than the one that registered it, so registered functions should not rely on the identity of objects with thread storage duration. — end note] After calling registered functions, \texttt{quick_exit} shall call \texttt{_Exit(status)}. [\textbf{Note:} The standard file buffers are not flushed. See: ISO C 7.20.4.4. — end note]

14 The function \texttt{quick_exit()} never returns to its caller.

See also: 3.6, 3.6.3, ISO C 7.10.4.

18.6 Dynamic memory management \textbf{[support.dynamic]}

The header \texttt{<new>} defines several functions that manage the allocation of dynamic storage in a program. It also defines components for reporting storage management errors.

\textbf{Header <new> synopsis}

```cpp
namespace std {
    class bad_alloc;
    class bad_array_new_length;
    struct nothrow_t {};
    extern const nothrow_t nothrow;
    typedef void (*new_handler)();
    new_handler set_new_handler(new_handler new_p) throw();
}
```

```cpp
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, 3.7.4, 5.3.4, 5.3.5, 12.5, 20.9.

18.6.1 Storage allocation and deallocation \textbf{[new.delete]}

Except where otherwise specified, the provisions of (3.7.4) apply to the library versions of \texttt{operator new} and \texttt{operator delete}.

18.6.1.1 Single-object forms \textbf{[new.delete.single]}

```cpp
void* operator new(std::size_t size) throw(std::bad_alloc);
```
Effects: The allocation function (3.7.4.1) called by a new-expression (5.3.4) to allocate size bytes of storage suitably aligned to represent any object of that size.

Replaceable: a C++ program may define a function with this function signature that displaces the default version defined by the C++ standard library.

Required behavior: Return a non-null pointer to suitably aligned storage (3.7.4), or else 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 argument in the most recent call to set_new_handler() (18.6.2.4) was a null pointer, throws bad_alloc.

- Otherwise, the function calls the current new_handler function (18.6.2.3). If the called function returns, the loop repeats.

- The loop terminates when an attempt to allocate the requested storage is successful or when a called new_handler function does not return.

```cpp
void* operator new(std::size_t size, const std::nothrow_t&) throw();
```

Effects: Same as above, except that it is called by a placement version of a new-expression when a C++ program prefers a null pointer result as an error indication, instead of a bad_alloc exception.

Replaceable: a C++ program may define a function with this function signature that displaces the default version defined by the C++ standard library.

Required behavior: Return a non-null pointer to suitably aligned storage (3.7.4), or else return a null pointer. This nothrow version of operator new returns a pointer obtained as if acquired from the (possibly replaced) ordinary version. This requirement is binding on a replacement version of this function.

Default behavior: Calls operator new(size). If the call returns normally, returns the result of that call. Otherwise, returns a null pointer.

```
void operator delete(void* ptr) throw();
```

Effects: The deallocation function (3.7.4.2) called by a delete-expression to render the value of ptr invalid.

Replaceable: a C++ program may define a function with 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*).

Requires: If an implementation has strict pointer safety (3.7.4.3) then ptr shall be a safely-derived pointer.
Default behavior: If `ptr` is null, does nothing. Otherwise, reclaims the storage allocated by the earlier call to `operator new`.

Remarks: It is unspecified under what conditions part or all of such reclaimed storage will be allocated by subsequent calls to `operator new` or any of `calloc`, `malloc`, or `realloc`, declared in `<cstdlib>`.

```cpp
void operator delete(void* ptr, const std::nothrow_t&) throw();
```

Effects: The deallocation function (3.7.4.2) called by the implementation to render the value of `ptr` invalid when the constructor invoked from a noexcept 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.

Required: If an implementation has strict pointer safety (3.7.4.3) then `ptr` shall be a safely-derived pointer.

Default behavior: calls `operator delete(ptr)`.

### 18.6.1.2 Array forms

```cpp
void* operator new[](std::size_t size) throw(std::bad_alloc);
```

Effects: The allocation function (3.7.4.1) called by the array form of a new-expression (5.3.4) to allocate `size` bytes of storage suitably aligned to represent any array object of that size or smaller.

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)`.

```cpp
void* operator new[](std::size_t size, const std::nothrow_t&) throw();
```

Effects: Same as above, except that it is called by a placement version of a new-expression when a C++ program prefers a null pointer result as an error indication, instead of a bad_alloc exception.

Replaceable: a C++ program can define a function with this function signature that displaces the default version defined by the C++ standard library.

Required behavior: Return a non-null pointer to suitably aligned storage (3.7.4), or return a null pointer. This requirement is binding on a replacement version of this function.

Default behavior: Calls `operator new[](size)`. If the call returns normally, returns the result of that call. Otherwise, returns a null pointer.

```cpp
void operator delete[](void* ptr) throw();
```

Effects: The deallocation function (3.7.4.2) called by the array form of a delete-expression to render the value of `ptr` invalid.

---

230) 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.
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*)`.

Requires: If an implementation has strict pointer safety (3.7.4.3) then `ptr` shall be a safely-derived pointer.

Default behavior: Calls `operator delete(ptr)`.

```cpp
void operator delete[](void* ptr, const std::nothrow_t&) throw();
```

Effects: The deallocation function (3.7.4.2) called by the implementation to render the value of `ptr` invalid when the constructor invoked from a nothrow placement version of the array `new-expression` throws an exception.

Replaceable: a C++ program may define a function with this function signature that displaces the default version defined by the C++ standard library.

Requires: If an implementation has strict pointer safety (3.7.4.3) then `ptr` shall be a safely-derived pointer.

Default behavior: calls `operator delete[](ptr)`.

### 18.6.1.3 Placement forms

These functions are reserved, a C++ program may not define functions that displace the versions in the Standard C++ library (17.6.3). The provisions of (3.7.4) do not apply to these reserved placement forms of `operator new` and `operator delete`.

```cpp
void* operator new(std::size_t size, void* ptr) 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.

Requires: If an implementation has strict pointer safety (3.7.4.3) then `ptr` shall be a safely-derived pointer.

Remarks: Default function called when any part of the initialization in a placement new expression that invokes the library’s non-array placement operator new terminates by throwing an exception (5.3.4).
void operator delete[](void* ptr, void*) throw();

Effects: Intentionally performs no action.

Requires: If an implementation has strict pointer safety (3.7.4.3) then ptr shall be a safely-derived pointer.

Remarks: Default function called when any part of the initialization in a placement new expression that invokes the library’s array placement operator new terminates by throwing an exception (5.3.4).

18.6.1.4 Data races

The library versions of operator new and operator delete, user replacement versions of global operator new and operator delete, and the C standard library functions calloc, malloc, realloc, and free shall not introduce data races (1.10) as a result of concurrent calls from different threads. Calls to these functions that allocate or deallocate a particular unit of storage shall occur in a single total order, and each such deallocation call shall happen before the next allocation (if any) in this order.

18.6.2 Storage allocation errors

18.6.2.1 Class bad_alloc

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();
        }
    }

The class bad_alloc defines the type of objects thrown as exceptions by the implementation to report a failure to allocate storage.

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.

bad_alloc(const bad_alloc&) throw();
bad_alloc& operator=(const bad_alloc&) throw();

Effects: Copies an object of class bad_alloc.

virtual const char* what() const throw();

Returns: An implementation-defined ntbs.

18.6.2.2 Class bad_array_new_length

namespace std {
    class bad_array_new_length : public bad_alloc {
        public:
            bad_array_new_length() throw();
        }
    }

§ 18.6.2.2
The class `bad_array_new_length` defines the type of objects thrown as exceptions by the implementation to report an attempt to allocate an array of size greater than an implementation-defined limit (5.3.4).

```cpp
bad_array_new_length() throw();
```

**Effects:** constructs an object of class `bad_array_new_length`.

**Remarks:** the result of calling `what()` on the newly constructed object is implementation-defined.

### 18.6.2.3 Type new_handler

```cpp
typedef void (*new_handler)();
```

The type of a *handler function* to be called by `operator new()` or `operator new[]()` (18.6.1) when they cannot satisfy a request for additional storage.

**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`;
- terminate execution of the program without returning to the caller;

### 18.6.2.4 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.

### 18.7 Type identification

The header `<typeinfo>` defines a type associated with type information generated by the implementation. It also defines two types for reporting dynamic type identification errors.

**Header `<typeinfo>` synopsis**

```cpp
namespace std {
    class type_info;
    class bad_cast;
    class bad_typeid;
}
```

**See also:** 5.2.7, 5.2.8.

### 18.7.1 Class type_info

```cpp
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;
            size_t hash_code() const throw();
            const char* name() const;
    }
}
```

§ 18.7.1
The class `type_info` describes type information generated by the implementation. Objects of this class effectively store a pointer to a name for the type, and an encoded value suitable for comparing two types for equality or collating order. The names, encoding rule, and collating sequence for types are all unspecified and may differ between programs.

```cpp
bool operator==(const type_info& rhs) const;  // Effects: Compares the current object with rhs.
Returns: true if the two values describe the same type.

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.

size_t hash_code() const throw();  // Returns: an unspecified value, except that within a single execution of the program, it shall return the same value for any two type_info objects which compare equal.
Remark: an implementation should return different values for two type_info objects which do not compare equal.

const char* name() const;  // Returns: an implementation-defined ntbs.
Remarks: The message may be a null-terminated multibyte string (17.5.2.1.4.2), suitable for conversion and display as a wstring (21.3, 22.4.1.4)
```

### 18.7.2 Class `bad_cast` [bad.cast]

```cpp
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.

```cpp
bad_cast(const bad_cast&) throw();
bad_cast& operator=(const bad_cast&) throw();
```

Effects: Copies an object of class `bad_cast`.

```cpp
virtual const char* what() const throw();
```

Returns: An implementation-defined `ntbs`.

Remarks: The message may be a null-terminated multibyte string (17.5.2.1.4.2), suitable for conversion and display as a `wstring` (21.3, 22.4.1.4)

### 18.7.3 Class `bad_typeid`

```cpp
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).

```cpp
bad_typeid() throw();
```

Effects: Constructs an object of class `bad_typeid`.

Remarks: The result of calling `what()` on the newly constructed object is implementation-defined.

```cpp
bad_typeid(const bad_typeid&) throw();
bad_typeid& operator=(const bad_typeid&) throw();
```

Effects: Copies an object of class `bad_typeid`.

```cpp
virtual const char* what() const throw();
```

Returns: An implementation-defined `ntbs`.

Remarks: The message may be a null-terminated multibyte string (17.5.2.1.4.2), suitable for conversion and display as a `wstring` (21.3, 22.4.1.4)

### 18.8 Exception handling

The header `<exception>` defines several types and functions related to the handling of exceptions in a C++ program.

**Header `<exception>` synopsis**

```cpp
namespace std {
    class exception;
    class bad_exception;
}
```
class nested_exception;

typedef void (*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler f) throw();
void unexpected [[noreturn]] ()

typedef void (*terminate_handler)();
terminate_handler set_terminate(terminate_handler f) throw();
void terminate [[noreturn]] ();

bool uncaught_exception() throw();

typedef unspecified exception_ptr;

exception_ptr current_exception();
void rethrow_exception [[noreturn]] (exception_ptr p);
template<class E> exception_ptr make_exception_ptr(E e);

template <class T> void throw_with_nested [[noreturn]] (T&& t);
template <class E> void rethrow_if_nested(const E& e);
}

See also: 15.5.

18.8.1 Class exception
[exception]

namespace std {

class exception {
    public:
        exception() throw();
        exception(const exception&) throw();
        exception& operator=(const exception&) throw();
        virtual ~exception() throw();
        virtual const char* what() const throw();
    }
}

1 The class exception defines the base class for the types of objects thrown as exceptions by C++ standard library components, and certain expressions, to report errors detected during program execution.

2 Each standard library class T that derives from class exception shall have a publicly accessible copy constructor and a publicly accessible copy assignment operator that do not exit with an exception. These member functions shall meet the following postcondition: If two objects lhs and rhs both have dynamic type T and lhs is a copy of rhs, then strcmp(lhs.what(), rhs.what()) shall equal 0.

exception() throw();

Effects: Constructs an object of class exception.

Remarks: Does not throw any exceptions.

exception(const exception& rhs) throw();
exception& operator=(const exception& rhs) throw();

Effects: Copies an exception object.
Postcondition: If this and rhs both have dynamic type exception then strcmp(what(), rhs.what()) shall equal 0.

virtual ~exception() throw();
Effects: Destroys an object of class exception.
Remarks: Does not throw any exceptions.

virtual const char* what() const throw();
Returns: An implementation-defined NTBS.
Remarks: The message may be a null-terminated multibyte string (17.5.2.1.4.2), suitable for conversion and display as a wstring (21.3, 22.4.1.4). The return value remains valid until the exception object from which it is obtained is destroyed or a non-const member function of the exception object is called.

18.8.2 Violating exception-specifications

18.8.2.1 Class bad_exception

namespace std {
class bad_exception : public exception {
public:
    bad_exception() throw();
    bad_exception(const bad_exception&) throw();
    bad_exception& operator=(const bad_exception&) throw();
    virtual const char* what() const throw();
};
}

The class bad_exception defines the type of objects thrown as described in (15.5.2).

bad_exception() throw();
Effects: Constructs an object of class bad_exception.
Remarks: The result of calling what() on the newly constructed object is implementation-defined.

bad_exception(const bad_exception&) throw();
bad_exception& operator=(const bad_exception&) throw();
Effects: Copies an object of class bad_exception.

virtual const char* what() const throw();
Returns: An implementation-defined NTBS.
Remarks: The message may be a null-terminated multibyte string (17.5.2.1.4.2), suitable for conversion and display as a wstring (21.3, 22.4.1.4).

18.8.2.2 Type unexpected_handler

typedef void (*unexpected_handler)();
The type of a handler function to be called by unexpected() when a function attempts to throw an exception not listed in its dynamic-exception-specification.

Required behavior: An unexpected_handler shall not return. See also 15.5.2.
Default behavior: The implementation’s default \texttt{unexpected\_handler} calls \texttt{terminate\()\).
18.8.4 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() returns true, throwing an exception can result in a call of terminate() (15.5.1).

18.8.5 Exception Propagation

typedef unspecified exception_ptr;

The type exception_ptr can be used to refer to an exception object.

exception_ptr shall satisfy the requirements of NullablePointer (20.2.3).

Two non-null values of type exception_ptr are equivalent and compare equal if and only if they refer to the same exception.

The default constructor of exception_ptr produces the null value of the type.

exception_ptr shall not be implicitly convertible to any arithmetic, enumeration, or pointer type.

[Note: An implementation might use a reference-counted smart pointer as exception_ptr. — end note]

exception_ptr current_exception();

Returns: An exception_ptr object that refers to the currently handled exception (15.3) or a copy of the currently handled exception, or a null exception_ptr object if no exception is being handled. The referenced object shall remain valid at least as long as there is an exception_ptr object that refers to it. If the function needs to allocate memory and the attempt fails, it returns an exception_ptr object that refers to an instance of bad_alloc. It is unspecified whether the return values of two successive calls to current_exception refer to the same exception object. [Note: That is, it is unspecified whether current_exception creates a new copy each time it is called. — end note] If the attempt to copy the current exception object throws an exception, the function returns an exception_ptr object that refers to the thrown exception or, if this is not possible, to an instance of bad_exception. [Note: The copy constructor of the thrown exception may also fail, so the implementation is allowed to substitute a bad_exception object to avoid infinite recursion. — end note]

Throws: nothing.

void rethrow_exception [[noreturn]] (exception_ptr p);

Requires: p shall not be a null pointer.

Throws: the exception object to which p refers.

template<class E> exception_ptr make_exception_ptr(E e);

Effects: Creates an exception_ptr object that refers to a copy of e, as if

try {
  throw e;
} catch(...) {

§ 18.8.5
return current_exception();
}

[Note: this function is provided for convenience and efficiency reasons. — end note]

18.8.6 nested_exception

namespace std {
    class nested_exception {
    public:
        nested_exception() throw();
        nested_exception(const nested_exception&) throw() = default;
        nested_exception& operator=(const nested_exception&) throw() = default;
        virtual ~nested_exception() = default;

    // access functions
    void rethrow_nested [[noreturn]] () const;
    exception_ptr nested_ptr() const;
    
    template<class T> void throw_with_nested [[noreturn]] (T&& t);
    template <class E> void rethrow_if_nested(const E& e);
    }
}

1 The class nested_exception is designed for use as a mixin through multiple inheritance. It captures the currently handled exception and stores it for later use.

2 [Note: nested_exception has a virtual destructor to make it a polymorphic class. Its presence can be tested for with dynamic_cast. — end note]

nested_exception() throw();

3 Effects: The constructor calls current_exception() and stores the returned value.

void rethrow_nested [[noreturn]] () const;

4 Effects: If nested_ptr() returns a null pointer, the function calls terminate(). Otherwise, it throws the stored exception captured by *this.

exception_ptr nested_ptr() const;

5 Returns: the stored exception captured by this nested_exception object.

template <class T> void throw_with_nested [[noreturn]] (T&& t);

6 Requires: T shall be CopyConstructible.

7 Throws: If T is a non-union class type not derived from nested_exception, an exception of unspecified type that is publicly derived from both T and nested_exception, otherwise t.

template <class E> void rethrow_if_nested(const E& e);

8 Effects: If the dynamic type of e is publicly and unambiguously derived from nested_exception, calls dynamic_cast<const nested_exception&>(e).rethrow_nested().

§ 18.8.6
18.9  Initializer lists  

The header `<initializer_list>` defines one type.

Header `<initializer_list>` synopsis

```cpp
namespace std {
    template<class E> class initializer_list {
        public:
            typedef E value_type;
            typedef const E& reference;
            typedef const E& const_reference;
            typedef size_t size_type;

            typedef const E* iterator;
            typedef const E* const_iterator;

        initializer_list();

        size_t size() const;  // number of elements
        const E* begin() const;  // first element
        const E* end() const;  // one past the last element
    };

    // 18.9.3 initializer list range access
    template<class E> const E* begin(initializer_list<E> il);
    template<class E> const E* end(initializer_list<E> il);
}
```

An object of type `initializer_list<E>` provides access to an array of objects of type `const E`.  

[Note: A pair of pointers or a pointer plus a length would be obvious representations for `initializer_list`. `initializer_list` is used to implement initializer lists as specified in §8.5.4. Copying an initializer list does not copy the underlying elements. — end note]

18.9.1  Initializer list constructors

```cpp
initializer_list();
```

Effects: constructs an empty `initializer_list` object.

Postcondition: `size() == 0`

Throws: nothing.

18.9.2  Initializer list access

```cpp
const E* begin() const;
```

Returns: a pointer to the beginning of the array. If `size() == 0` the values of `begin()` and `end()` are unspecified but they shall be identical.

Throws: nothing.

```cpp
const E* end() const;
```

Returns: `begin() + size()`

Throws: nothing.
size_t size() const;

   Returns: the number of elements in the array.

   Throws: nothing.

   Complexity: constant time.

18.9.3 Initializer list range access  

template<class E> const E* begin(initializer_list<E> il);

   Returns: il.begin().

template<class E> const E* end(initializer_list<E> il);

   Returns: il.end().

18.10 Other runtime support

1 Headers <cstdarg> (variable arguments), <csetjmp> (nonlocal jumps), <ctime> (system clock clock(), time()), <csignal> (signal handling), <cstdlib> (runtime environment getenv(), system()), and <cstdbool> (_ _ bool_true_false_are_defined).

2 The contents of these headers are the same as the Standard C library headers <stdarg.h>, <setjmp.h>, <time.h>, <signal.h>, and <stdlib.h> respectively, with the following changes:

3 The restrictions that ISO C places on the second parameter to the va_start() macro in header <stdarg.h> are different in this International Standard. The parameter parmN is the identifier of the rightmost parameter in the variable parameter list of the function definition (the one just before the ...). 231 If the parameter

231) Note that va_start is required to work as specified even if unary operator& is overloaded for the type of parmN.
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 4.8.1.1.

4 The function signature `longjmp(jmp_buf jbuf, int val)` has more restricted behavior in this International Standard. A `setjmp/longjmp` call pair has undefined behavior if replacing the `setjmp` and `longjmp` by `catch` and `throw` would invoke any non-trivial destructors for any automatic objects.

See also: ISO C 7.10.4, 7.8, 7.6, 7.12.

5 Calls to the function `getenv` shall not introduce a data race (17.6.4.8) provided that nothing modifies the environment. [Note: Calls to the POSIX functions `setenv` and `putenv` modify the environment. — end note]

6 The header `<cstdlib>` and the header `<stdbool.h>` shall not define macros named `bool`, `true`, and `false`.

7 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.232

---

232) In particular, a signal handler using exception handling is very likely to have problems. Also, invoking `std::exit` may cause destruction of objects, including those of the standard library implementation, which, in general, yields undefined behavior in a signal handler (see 1.9).
19 Diagnostics library

19.1 General

This Clause describes components that C++ programs may use to detect and report error conditions.

The following subclauses describe components for reporting several kinds of exceptional conditions, documenting program assertions, and a global variable for error number codes, as summarized in Table 27.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.2 Exception classes</td>
<td>&lt;stdexcept&gt;</td>
</tr>
<tr>
<td>19.3 Assertions</td>
<td>&lt;cassert&gt;</td>
</tr>
<tr>
<td>19.4 Error numbers</td>
<td>&lt;cerrno&gt;</td>
</tr>
<tr>
<td>19.5 System error support</td>
<td>&lt;system_error&gt;</td>
</tr>
</tbody>
</table>

19.2 Exception classes

The Standard C++ library provides classes to be used to report certain errors (17.6.4.11) in C++ programs. In the error model reflected in these classes, errors are divided into two broad categories: logic errors and runtime errors.

The distinguishing characteristic of logic errors is that they are due to errors in the internal logic of the program. In theory, they are preventable.

By contrast, runtime errors are due to events beyond the scope of the program. They cannot be easily predicted in advance. The header <stdexcept> defines several types of predefined exceptions for reporting errors in a C++ program. These exceptions are related by inheritance.

**Header <stdexcept> synopsis**

```c++
namespace std {
    class logic_error;
    class domain_error;
    class invalid_argument;
    class length_error;
    class out_of_range;
    class runtime_error;
    class range_error;
    class overflow_error;
    class underflow_error;
}
```

19.2.1 Class logic_error

```c++
namespace std {
    class logic_error : public exception {
        public:
            explicit logic_error(const string& what_arg);
    }
}
```
The class `logic_error` defines the type of objects thrown as exceptions to report errors presumably detectable before the program executes, such as violations of logical preconditions or class invariants.

```cpp
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.

```cpp
logic_error(const string& what_arg);
```

2. *Effects:* Constructs an object of class `logic_error`.

```cpp
Postcondition: strcmp(what(), what_arg.c_str()) == 0.
```

```cpp
logic_error(const char* what_arg);
```

4. *Effects:* Constructs an object of class `logic_error`.

```cpp
Postcondition: strcmp(what(), what_arg) == 0.
```

### 19.2.2 Class `domain_error`

```cpp
namespace std {
    class domain_error : public logic_error {
        public:
            explicit domain_error(const string& what_arg);
            explicit domain_error(const char* what_arg);
    };
}
```

1. The class `domain_error` defines the type of objects thrown as exceptions by the implementation to report domain errors.

```cpp
domain_error(const string& what_arg);
```

2. *Effects:* Constructs an object of class `domain_error`.

```cpp
Postcondition: strcmp(what(), what_arg.c_str()) == 0.
```

```cpp
domain_error(const char* what_arg);
```


```cpp
Postcondition: strcmp(what(), what_arg) == 0.
```

### 19.2.3 Class `invalid_argument`

```cpp
namespace std {
    class invalid_argument : public logic_error {
        public:
            explicit invalid_argument(const string& what_arg);
            explicit invalid_argument(const char* what_arg);
    };
}
```

1. The class `invalid_argument` defines the type of objects thrown as exceptions to report an invalid argument.

```cpp
invalid_argument(const string& what_arg);
```
19.2.4 Class length_error

```cpp
namespace std {
    class length_error : public logic_error {
    public:
        explicit length_error(const string& what_arg);
        explicit length_error(const char* what_arg);
    }
}
```

The class length_error defines the type of objects thrown as exceptions to report an attempt to produce an object whose length exceeds its maximum allowable size.

```cpp
length_error(const string& what_arg);
```

Effects: Constructs an object of class length_error.

```cpp
Postcondition: strcmp(what(), what_arg.c_str()) == 0.
```

19.2.5 Class out_of_range

```cpp
namespace std {
    class out_of_range : public logic_error {
    public:
        explicit out_of_range(const string& what_arg);
        explicit out_of_range(const char* what_arg);
    }
}
```

The class out_of_range defines the type of objects thrown as exceptions to report an argument value not in its expected range.

```cpp
out_of_range(const string& what_arg);
```

Effects: Constructs an object of class out_of_range.

```cpp
Postcondition: strcmp(what(), what_arg.c_str()) == 0.
```
19.2.6 Class runtime_error [runtime.error]

namespace std {
    class runtime_error : public exception {
        public:
            explicit runtime_error(const string& what_arg);
            explicit runtime_error(const char* what_arg);
        };
    }
}

1 The class runtime_error defines the type of objects thrown as exceptions to report errors presumably detectable only when the program executes.

runtime_error(const string& what_arg);
2 Effects: Constructs an object of class runtime_error.
3 Postcondition: strcmp(what(), what_arg.c_str()) == 0.

runtime_error(const char* what_arg);
4 Effects: Constructs an object of class runtime_error.
5 Postcondition: strcmp(what(), what_arg) == 0.

19.2.7 Class range_error [range.error]

namespace std {
    class range_error : public runtime_error {
        public:
            explicit range_error(const string& what_arg);
            explicit range_error(const char* what_arg);
        };
    }
}

1 The class range_error defines the type of objects thrown as exceptions to report range errors in internal computations.

range_error(const string& what_arg);
2 Effects: Constructs an object of class range_error.
3 Postcondition: strcmp(what(), what_arg.c_str()) == 0.

range_error(const char* what_arg);
4 Effects: Constructs an object of class range_error.
5 Postcondition: strcmp(what(), what_arg) == 0.

19.2.8 Class overflow_error [overflow.error]

namespace std {
    class overflow_error : public runtime_error {
        public:
            explicit overflow_error(const string& what_arg);
            explicit overflow_error(const char* what_arg);
        };
    }
}

§ 19.2.8
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`.

### 19.2.9 Class `underflow_error`  [underflow.error]

```cpp
namespace std {
    class underflow_error : public runtime_error {
    public:
        explicit underflow_error(const string& what_arg);
        explicit underflow_error(const char* what_arg);
    };
}
```

The class `underflow_error` defines the type of objects thrown as exceptions to report an arithmetic underflow error.

```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`.

### 19.3 Assertions  [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>Macro</td>
<td>assert</td>
</tr>
</tbody>
</table>

The contents are the same as the Standard C library header `<assert.h>`.

See also: ISO C 7.2.
19.4 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] A separate `errno` value shall be provided for each thread.

Table 29 — Header `<cerrno>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td>ECONNREFUSED</td>
</tr>
<tr>
<td></td>
<td>E2BIG</td>
</tr>
<tr>
<td></td>
<td>EACCESS</td>
</tr>
<tr>
<td></td>
<td>EADDRINUSE</td>
</tr>
<tr>
<td></td>
<td>EADDRNOTAVAIL</td>
</tr>
<tr>
<td></td>
<td>EAFNOSUPPORT</td>
</tr>
<tr>
<td></td>
<td>EAGAIN</td>
</tr>
<tr>
<td></td>
<td>EALREADY</td>
</tr>
<tr>
<td></td>
<td>EBADF</td>
</tr>
<tr>
<td></td>
<td>EBADMSG</td>
</tr>
<tr>
<td></td>
<td>EBUSY</td>
</tr>
<tr>
<td></td>
<td>ECANCELED</td>
</tr>
<tr>
<td></td>
<td>ECHILD</td>
</tr>
<tr>
<td></td>
<td>ECONNABorted</td>
</tr>
</tbody>
</table>

19.5 System error support

This subclause describes components that the standard library and C++ programs may use to report error conditions originating from the operating system or other low-level application program interfaces.

Components described in this subclause shall not change the value of `errno` (19.4). Implementations should leave the error states provided by other libraries unchanged.

Header `<system_error>` synopsis

```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 {};

    enum class errc {
        address_family_not_supported, // EAFNOSUPPORT
        address_in_use,               // EADDRINUSE
        address_not_available,        // EADDRNOTAVAIL
        already_connected,            // EISCONN
        argument_list_too_long,       // E2BIG
        argument_out_of_domain,       // EDOM
        bad_address,                  // EFAULT
    
    };
```

§ 19.5
bad_file_descriptor,  // EBADF
bad_message,  // EBADMSG
broken_pipe,  // EPIPE
connection_aborted,  // ECONNABORTED
connection_already_in_progress,  // EALREADY
connection_refused,  // ECONNREFUSED
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,  // EILSEQ
inappropriate_io_control_operation,  // ENOTTY
interrupted,  // EINTR
invalid_argument,  // EINVAL
invalid_seek,  // ESPIPE
io_error,  // EINVAL
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,  // ENOMSG
no_message,  // ENOMSG
no_protocol_option,  // ENOPROTOOPT
no_space_on_device,  // ENOSPC
no_stream_resources,  // ENOSR
no_such_device_or_address,  // ENXIO
no_such_device,  // ENODEV
no_such_file_or_directory,  // ENOENT
no_such_process,  // ESRCH
not_a_directory,  // ENOTDIR
not_a_socket,  // ENOTSOCK
not_a_stream,  // ENOTSTR
not_connected,  // ENOTCONN
not_enough_memory,  // ENOMEM
not_supported,  // ENOTSUP
operation_canceled,  // ECANCELED
operation_in_progress,  // EINPROGRESS
operation_not_permitted,  // EPERM
operation_not_supported,  // ENOTSUPP
operation_would_block,  // EWOULDBLOCK
owner_dead,  // EOWNERDEAD
permission_denied,  // EACCES

§ 19.5 467
The value of each `enum errc` constant shall be the same as the value of the `<cerrno>` macro shown in the above synopsis. Whether or not the `<system_error>` implementation exposes the `<cerrno>` macros is unspecified.

19.5.1 Class error_category

19.5.1.1 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]
19.5.1.2 Class error_category virtual members

virtual const char* name() const = 0;

Returns: A string naming the error category.

virtual error_condition default_error_condition(int ev) const;

Returns: error_condition(ev, *this).

virtual bool equivalent(int code, const error_condition& condition) const;

Returns: default_error_condition(code) == condition.

virtual bool equivalent(const error_code& code, int condition) const;

Returns: *this == code.category() && code.value() == condition.

virtual string message(int ev) const = 0;

Returns: A string that describes the error condition denoted by ev.

19.5.1.3 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<const error_category*>(this, &rhs).

[Note: less (20.8.6) provides a total ordering for pointers. — end note]

Throws: Nothing.

### 19.5.1.4 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.

### 19.5.1.5 Error category objects

const error_category & generic_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 "generic".

const error_category & 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, generic_category()). Otherwise, the function shall return error_condition(ev, system_category()). What constitutes correspondence for any given operating system is unspecified. [Note: The number of potential system error codes is large and unbounded, and some may not correspond to any POSIX errno value. Thus implementations are given latitude in determining correspondence. — end note]
19.5.2 Class error_code

19.5.2.1 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:
            // 19.5.2.2 constructors:
            error_code();
            error_code(int val, const error_category& cat);
            template <class ErrorCodeEnum>
                error_code(ErrorCodeEnum e);

            // 19.5.2.3 modifiers:
            void assign(int val, const error_category& cat);
            template <class ErrorCodeEnum>
                errorcode& operator=(ErrorCodeEnum e);
            void clear();

            // 19.5.2.4 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
        };

        // 19.5.2.5 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

19.5.2.2 Class error_code constructors

error_code();
1    Effects: Constructs an object of type error_code.
2    Postconditions: val_ == 0 and cat_ == &system_category().
3    Throws: Nothing.

error_code(int val, const error_category& cat);
4    Effects: Constructs an object of type error_code.
5    Postconditions: val_ == val and cat_ == &cat.
template <class ErrorCodeEnum>
error_code(ErrorCodeEnum e);

Effects: Constructs an object of type error_code.

Postconditions: *this == make_error_code(e).

Throws: Nothing.

Remarks: This constructor shall not participate in overload resolution unless is_error_code_enum<ErrorCodeEnum>::value is true.

19.5.2.3 Class error_code modifiers

void assign(int val, const error_category& cat);

Postconditions: val_ == val and cat_ == &cat.

Throws: Nothing.

template <class ErrorCodeEnum>
error_code& operator=(ErrorCodeEnum e);

Postconditions: *this == make_error_code(e).

Returns: *this.

Throws: Nothing.

Remarks: This operator shall not participate in overload resolution unless is_error_code_enum<ErrorCodeEnum>::value is true.

void clear();

Postconditions: value() == 0 and category() == system_category().

19.5.2.4 Class error_code observers

int value() const;

Returns: val_.

Throws: Nothing.

const error_category& category() const;

Returns: *cat_.

Throws: Nothing.

error_condition default_error_condition() const;

Returns: category().default_error_condition(value()).

Throws: Nothing.

string message() const;

Returns: category().message(value()).

§ 19.5.2.4
explicit operator bool() const;

8    \textit{Returns:} value() \neq 0.

9    \textit{Throws:} Nothing.

\section*{19.5.2.5 Class error\_code non-member functions \hspace{1em} [syserr.errcode.nonmembers]}

error\_code make\_error\_code(errc e);

\textit{Returns:} error\_code(static\_cast<int>(e), generic\_category()).

bool operator<(const error\_code& lhs, const error\_code& rhs);

1    \textit{Returns:} lhs\_category() < rhs\_category() || lhs\_category() == rhs\_category() && lhs\_value() < rhs\_value().

2    \textit{Throws:} Nothing.

template <class charT, class traits>
    basic\_ostream<charT, traits>&
    operator<<(basic\_ostream<charT, traits>& os, const error\_code& ec);

3    \textit{Effects:} os \ll ec\_category().name() \ll ':' \ll ec\_value().

\section*{19.5.3 Class error\_condition \hspace{1em} [syserr.errcondition]}

\subsection*{19.5.3.1 Class error\_condition overview \hspace{1em} [syserr.errcondition.overview]}

1 The class \texttt{error\_condition} describes an object used to hold values identifying error conditions. [\textit{Note: error\_condition} values are portable abstractions, while \texttt{error\_code} values (19.5.2) are implementation specific. -- end note]

\begin{verbatim}
namespace std {
    class error\_condition {
        public:
            // 19.5.3.2 constructors:
            error\_condition();
            error\_condition(int val, const error\_category& cat);
            template <class ErrorConditionEnum>
                error\_condition(ErrorConditionEnum e);

            // 19.5.3.3 modifiers:
            void assign(int val, const error\_category& cat);
            template<class ErrorConditionEnum>
                error\_condition& operator=(ErrorConditionEnum e);
            void clear();

            // 19.5.3.4 observers:
            int value() const;
            const error\_category& category() const;
            string message() const;
            explicit operator bool() const;

        private:
            int val_;          // exposition only
            const error\_category* cat_; // exposition only
        };
\end{verbatim}

\section*{§ 19.5.3.1}
19.5.3.2 Class `error_condition` constructors

```cpp
error_condition();
Effects: Constructs an object of type `error_condition`.
Postconditions: `val_` == 0 and `cat_` == &generic_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);
Effects: Constructs an object of type `error_condition`.
Postcondition: `*this` == make_error_condition(e).
Throws: Nothing.
Remarks: This constructor shall not participate in overload resolution unless `is_error_condition_enum<ErrorConditionEnum>::value` is true.
```

19.5.3.3 Class `error_condition` modifiers

```cpp
void assign(int val, const error_category& cat);
Postconditions: `val_` == `val` and `cat_` == &`cat`.
Throws: Nothing.
```

```cpp
template <class ErrorConditionEnum>
error_condition& operator=(ErrorConditionEnum e);
Postcondition: `*this` == make_error_condition(e).
Returns: `*this`.
Throws: Nothing.
Remarks: This operator shall not participate in overload resolution unless `is_error_condition_enum<ErrorConditionEnum>::value` is true.
```

```cpp
void clear();
Postconditions: `value()` == 0 and `category()` == `generic_category()`.
```
19.5.3.4 Class error_condition observers

```cpp
int value() const;
1  Returns: val_.
2  Throws: Nothing.

const error_category& category() const;
3  Returns: *cat_.
4  Throws: Nothing.

string message() const;
5  Returns: category().message(value()).

explicit operator bool() const;
6  Returns: value() != 0.
7  Throws: Nothing.
```

19.5.3.5 Class error_condition non-member functions

```cpp
error_condition make_error_condition(errc e);
1  Returns: error_condition(static_cast<int>(e), generic_category()).

bool operator<(const error_condition& lhs, const error_condition& rhs);
1  Returns: lhs.category() < rhs.category() || lhs.category() == rhs.category() && lhs.value() < rhs.value().
2  Throws: Nothing.
```

19.5.4 Comparison operators

```cpp
bool operator==(const error_code& lhs, const error_code& rhs);
1  Returns: lhs.category() == rhs.category() && lhs.value() == rhs.value().
2  Throws: Nothing.

bool operator==(const error_code& lhs, const error_condition& rhs);
3  Returns: lhs.category().equivalent(lhs.value(), rhs) || rhs.category().equivalent(lhs, rhs.value()).
4  Throws: Nothing.

bool operator==(const error_condition& lhs, const error_code& rhs);
5  Returns: rhs.category().equivalent(rhs.value(), lhs) || lhs.category.equivalent(rhs, lhs.value()).
6  Throws: Nothing.
```

§ 19.5.4
7    Returns: lhs.category() == rhs.category() && lhs.value() == rhs.value().
8    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: !(lhs == rhs).
Throws: Nothing.

19.5.5 Hash support

template <> struct hash<error_code>;

Requires: the template specialization shall meet the requirements of class template hash (20.8.15).

19.5.6 Class system_error

19.5.6.1 Class system_error overview

The class system_error describes an exception object used to report error conditions that have an associated error code. Such error conditions typically originate from the operating system or other low-level application program interfaces.

[Note: If an error represents an out-of-memory condition, implementations are encouraged to throw an exception object of type bad_alloc 18.6.2.1 rather than system_error. — end note]

namespace std {
    class system_error : public runtime_error {
    public:
        system_error(error_code ec, const string& what_arg);
        system_error(error_code ec, const char* what_arg);
        system_error(error_code ec);
        system_error(int ev, const error_category& ecat,
                        const string& what_arg);
        system_error(int ev, const error_category& ecat,
                        const char* what_arg);
        system_error(int ev, const error_category& ecat);
        error_code& code() const throw();
        const char* what() const throw();
    };
}   // namespace std

19.5.6.2 Class system_error members

system_error(error_code ec, const string& what_arg);

Effects: Constructs an object of class system_error.

Postconditions: code() == ec.
    string(what()).find(what_arg) != string::npos.

system_error(error_code ec, const char* what_arg);
3  **Effects:** Constructs an object of class `system_error`.

4  **Postconditions:** `code() == ec`.

5  ```cpp
system_error(error_code ec);
```

6  **Effects:** Constructs an object of class `system_error`.

7  **Postconditions:** `code() == ec`.

8  ```cpp
system_error(int ev, const error_category& ecat,
             const string& what_arg);
```

9  **Effects:** Constructs an object of class `system_error`.

10  **Postconditions:** `code() == error_code(ev, ecat)`.

11  ```cpp
string(what()).find(what_arg) != string::npos.
```

12  ```cpp
system_error(int ev, const error_category& ecat,
             const string& what_arg);
```

13  **Effects:** Constructs an object of class `system_error`.

14  **Postconditions:** `code() == error_code(ev, ecat)`.

15  ```cpp
const error_code& code() const throw();
```

16  **Returns:** `ec` or `error_code(ev, ecat)`, from the constructor, as appropriate.

17  ```cpp
const char* what() const throw();
```

18  **Returns:** An `ntbs` incorporating and `code().message()` the arguments supplied in the constructor.

19  ```cpp
[ Note: The returned `ntbs` might be the contents of `what_arg + ": " + code.message(). — end note ]
```
20 General utilities library

20.1 General

This Clause describes components used by other elements of the C++ standard library. These components may also be used by C++ programs.

The following subclauses describe utility and allocator requirements, utility components, compile-time rational arithmetic, tuples, type traits templates, function objects, dynamic memory management utilities, and date/time utilities, as summarized in Table 30.

Table 30 — General utilities library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.2</td>
<td>Requirements</td>
</tr>
<tr>
<td>20.3</td>
<td>Utility components       &lt;utility&gt;</td>
</tr>
<tr>
<td>20.5</td>
<td>Fixed-size sequences of bits &lt;bitset&gt;</td>
</tr>
<tr>
<td>20.6</td>
<td>Compile-time rational arithmetic &lt;ratio&gt;</td>
</tr>
<tr>
<td>20.4</td>
<td>Tuples &lt;tuple&gt;</td>
</tr>
<tr>
<td>20.7</td>
<td>Type traits &lt;type_traits&gt;</td>
</tr>
<tr>
<td>20.8</td>
<td>Function objects &lt;functional&gt;</td>
</tr>
<tr>
<td>20.9</td>
<td>Memory &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>20.10</td>
<td>Time utilities &lt;chrono&gt;</td>
</tr>
<tr>
<td>20.11</td>
<td>Date and time functions &lt;ctime&gt;</td>
</tr>
<tr>
<td>20.12</td>
<td>Type indexes &lt;typeindex&gt;</td>
</tr>
</tbody>
</table>

20.2 Requirements

20.2.1 describes requirements on types and expressions used to instantiate templates defined in the C++ standard library. 20.2.2 describes the requirements on swappable types and swappable expressions. 20.2.4 describes the requirements on hash function objects. 20.2.5 describes the requirements on storage allocators.

20.2.1 Template argument requirements

The template definitions in the C++ standard library refer to various named requirements whose details are set out in tables 31–38. In these tables, T is an object or reference type to be supplied by a C++ program instantiating a template; a, b, and c are values of type (possibly const) T; s and t are modifiable lvalues of type T; u denotes an identifier; rv is an rvalue of type T; and v is a lvalue of type (possibly const) T or an rvalue of type const T.

In general, a default constructor is not required. Certain container class member function signatures specify the default constructor as a default argument. T() shall be a well-defined expression (8.5) if one of those signatures is called using the default argument (8.3.6).
Table 31 — EqualityComparable requirements [equalitycomparable]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Requirement</th>
</tr>
</thead>
</table>
| a == b     | convertible to bool | == is an equivalence relation, that is, it has the following properties:  
- For all a, a == a.  
- If a == b, then b == a.  
- If a == b and b == c, then a == c. |

Table 32 — LessThanComparable requirements [lessthancomparable]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt; b</td>
<td>convertible to bool</td>
<td>&lt; is a strict weak ordering relation (25.4)</td>
</tr>
</tbody>
</table>

Table 33 — DefaultConstructible requirements [defaultconstructible]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>T t;</td>
<td>object t is default-initialized</td>
</tr>
<tr>
<td>T u();</td>
<td>object u is value-initialized</td>
</tr>
<tr>
<td>T()</td>
<td>a temporary object of type T is value-initialized</td>
</tr>
</tbody>
</table>

Table 34 — MoveConstructible requirements [moveconstructible]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>T u(rv);</td>
<td>u is equivalent to the value of rv before the construction</td>
</tr>
<tr>
<td>T(rv)</td>
<td>T(rv) is equivalent to the value of rv before the construction</td>
</tr>
</tbody>
</table>

[Note: rv remains a valid object. Its state is unspecified — end note]

Table 35 — CopyConstructible requirements (in addition to MoveConstructible) [copyconstructible]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>T u(v);</td>
<td>the value of v is unchanged and is equivalent to u</td>
</tr>
<tr>
<td>T(v)</td>
<td>the value of v is unchanged and is equivalent to T(v)</td>
</tr>
</tbody>
</table>

Table 36 — MoveAssignable requirements [moveassignable]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Return value</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = rv</td>
<td>T&amp;</td>
<td>t</td>
<td>t is equivalent to the value of rv before the assignment</td>
</tr>
</tbody>
</table>

[Note: rv remains a valid object. Its state is unspecified. — end note]
Table 37 — **CopyAssignable** requirements (in addition to MoveAssignable) [copyassignable]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Return value</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>t = v</code></td>
<td><code>T&amp;</code></td>
<td><code>t</code></td>
<td><code>t</code> is equivalent to <code>v</code>, the value of <code>v</code> is unchanged</td>
</tr>
</tbody>
</table>

Table 38 — **Destructible** requirements [destructible]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>u.~T()</code></td>
<td>All resources owned by <code>u</code> are reclaimed, no exception is propagated.</td>
</tr>
</tbody>
</table>
20.2.2 Swappable requirements

1 This subclause provides definitions for swappable types and expressions. In these definitions, let \( t \) denote an expression of type \( T \), and let \( u \) denote an expression of type \( U \).

2 An object \( t \) is swappable with an object \( u \) if and only if:
   - the expressions \( \text{swap}(t, u) \) and \( \text{swap}(u, t) \) are valid when evaluated in the context described below, and
   - these expressions have the following effects:
     - the object referred to by \( t \) has the value originally held by \( u \) and
     - the object referred to by \( u \) has the value originally held by \( t \).

3 The context in which \( \text{swap}(t, u) \) and \( \text{swap}(u, t) \) are evaluated shall ensure that a binary non-member function named “swap” is selected via overload resolution (13.3) on a candidate set that includes:
   - the two \( \text{swap} \) function templates defined in \(<\text{utility}>\) (20.3) and
   - the lookup set produced by argument-dependent lookup (3.4.2).

[Note: If \( T \) and \( U \) are both fundamental types or arrays of fundamental types and the declarations from the header \(<\text{utility}>\) are in scope, the overall lookup set described above is equivalent to that of the qualified name lookup applied to the expression \( \text{std::swap}(t, u) \) or \( \text{std::swap}(u, t) \) as appropriate. — end note]

[Note: It is unspecified whether a library component that has a swappable requirement includes the header \(<\text{utility}>\) to ensure an appropriate evaluation context. — end note]

4 An rvalue or lvalue \( t \) is swappable if and only if \( t \) is swappable with any rvalue or lvalue, respectively, of type \( T \).

5 A type \( X \) satisfying any of the iterator requirements (24.2) is ValueSwappable if, for any dereferenceable object \( x \) of type \( X \), \( *x \) is swappable.

[Example: User code can ensure that the evaluation of \( \text{swap} \) calls is performed in an appropriate context under the various conditions as follows:

```cpp
#include <utility>

// Requires: \( \text{std::forward<T>(t)} \) shall be swappable with \( \text{std::forward<U>(u)} \).
template <class T, class U>
void value_swap(T&& t, U&& u) {
    using std::swap;
    swap(std::forward<T>(t), std::forward<U>(u)); // OK: uses “swappable with” conditions
    // for rvalues and lvalues
}

// Requires: lvalues of T shall be swappable.
template <class T>
void lv_swap(T& t1 T& t2) {
    using std::swap;
    swap(t1, t2); // OK: uses swappable conditions for
    // lvalues of type T
}

namespace N {
    struct A { int m; };
    struct Proxy { A* a; };
    Proxy proxy(A* a) { return Proxy{ a }; }
}
```
```cpp
void swap(A & x, Proxy p) {
    std::swap(x.m, p.a->m);  // OK: uses context equivalent to swappable
                        // conditions for fundamental types
}
void swap(Proxy p, A & x) { swap(x, p); }  // satisfy symmetry constraint

int main() {
    int i = 1, j = 2;
    lv_swap(i, j);
    assert(i == 2 && j == 1);

    N::A a1 = { 5 }, a2 = { -5 };
    value_swap(a1, proxy(a2));
    assert(a1.m == -5 && a2.m == 5);
}
— end example]

20.2.3 NullablePointer requirements [nullablepointer.requirements]

1 A NullablePointer type is a pointer-like type that supports null values. A type P meets the requirements of NullablePointer if:
   — P satisfies the requirements of EqualityComparable, DefaultConstructible, CopyConstructible, CopyAssignable, and Destructible,
   — lvalues of type P are swappable (20.2.2),
   — the expressions shown in Table 39 are valid and have the indicated semantics, and
   — P satisfies all the other requirements of this subclause.

2 A value-initialized object of type P produces the null value of the type. The null value shall be equivalent only to itself. A default-initialized object of type P may have an indeterminate value. [Note: Operations involving indeterminate values may cause undefined behavior. — end note]

3 An object p of type P can be contextually converted to bool (Clause 4). The effect shall be as if p != nullptr had been evaluated in place of p.

4 No operation which is part of the NullablePointer requirements shall exit via an exception.

5 In Table 39, u denotes an identifier, t denotes a non-const lvalue of type P, a and b denote values of type (possibly const) P, and np denotes a value of type (possibly const) std::nullptr_t.

20.2.4 Hash requirements [hash.requirements]

1 A type H meets the Hash requirements if:
   — it is a function object type (20.8),
   — it satisfies the requirements of CopyConstructible and Destructible (20.2.1),
   — the expressions shown in the following table are valid and have the indicated semantics, and
   — it satisfies all other requirements in this subclause.
```
Table 39 — NullablePointer requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>P u(np);</code></td>
<td></td>
<td>post: <code>u == nullptr</code></td>
</tr>
<tr>
<td><code>P u = np;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>P(np)</code></td>
<td><code>P&amp;</code></td>
<td>post: <code>P(np) == nullptr</code></td>
</tr>
<tr>
<td><code>t = np</code></td>
<td></td>
<td>post: <code>t == nullptr</code></td>
</tr>
<tr>
<td><code>a != b</code></td>
<td>contextually convertible to <code>bool</code></td>
<td>!(a == b)</td>
</tr>
<tr>
<td><code>a == np</code></td>
<td>contextually convertible to <code>bool</code></td>
<td><code>a == P()</code></td>
</tr>
<tr>
<td><code>np == a</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a != np</code></td>
<td>contextually convertible to <code>bool</code></td>
<td>!(a == np)</td>
</tr>
<tr>
<td><code>np != a</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Given `Key` is an argument type for function objects of type `H`, in Table 40 `h` is a value of type (possibly `const`) `H`, `u` is an lvalue of type `Key`, and `k` is a value of a type convertible to (possibly `const`) `Key`.

Table 40 — Hash requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>h(k)</code></td>
<td><code>size_t</code></td>
<td>Shall not throw exceptions. The value returned shall depend only on the argument <code>k</code>. [Note: thus all evaluations of the expression <code>h(k)</code> with the same value for <code>k</code> yield the same result. — end note] [Note: for two different values <code>t1</code> and <code>t2</code>, the probability that <code>h(t1)</code> and <code>h(t2)</code> compare equal should be very small, approaching <code>1.0 / numeric_limits&lt;size_t&gt;::max()</code>. — end note]</td>
</tr>
<tr>
<td><code>h(u)</code></td>
<td><code>size_t</code></td>
<td>Shall not modify <code>u</code>.</td>
</tr>
</tbody>
</table>

## 20.2.5 Allocator requirements

1. The library describes a standard set of requirements for *allocators*, which are class-type objects that encapsulate the information about an allocation model. This information includes the knowledge of pointer types, the type of their difference, the type of the size of objects in this allocation model, as well as the memory allocation and deallocation primitives for it. All of the string types (Clause 21), containers (Clause 23) (except array (Clause 23)), string buffers and string streams (Clause 27), and `match_results` (Clause 28) are parameterized in terms of allocators.

2. The template struct `allocator_traits` (20.9.4) supplies a uniform interface to all allocator types. Table 41 describes the types manipulated through allocators. Table 42 describes the requirements on allocator types and thus on types used to instantiate `allocator_traits`. A requirement is optional if the last column of Table 42 specifies a default for a given expression. Within the standard library `allocator_traits` template, an optional requirement that is not supplied by an allocator is replaced by the specified default expression. A user specialization of `allocator_traits` may provide different defaults and may provide defaults for different requirements than the primary template. Within Tables 41 and 42, the use of `move` and `forward` always refers to `std::move` and `std::forward`, respectively.

Table 41 — Descriptive variable definitions

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T, U, C</code></td>
<td>any non-const, non-reference object type</td>
</tr>
</tbody>
</table>
Table 41 — Descriptive variable definitions (continued)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>a type convertible to T</td>
</tr>
<tr>
<td>X</td>
<td>an Allocator class for type T</td>
</tr>
<tr>
<td>Y</td>
<td>the corresponding Allocator class for type U</td>
</tr>
<tr>
<td>XX</td>
<td>the type allocator_traits&lt;X&gt;</td>
</tr>
<tr>
<td>YY</td>
<td>the type allocator_traits&lt;Y&gt;</td>
</tr>
<tr>
<td>t</td>
<td>a value of type const T&amp;</td>
</tr>
<tr>
<td>a, a1, a2</td>
<td>values of type X&amp;</td>
</tr>
<tr>
<td>a3</td>
<td>an rvalue of type X</td>
</tr>
<tr>
<td>b</td>
<td>a value of type Y</td>
</tr>
<tr>
<td>c</td>
<td>a dereferenceable pointer of type C*</td>
</tr>
<tr>
<td>p</td>
<td>a value of type XX::pointer, obtained by calling a1.allocate, where a1 == a</td>
</tr>
<tr>
<td>q</td>
<td>a value of type XX::const_pointer obtained by conversion from a value p.</td>
</tr>
<tr>
<td>w</td>
<td>a value of type XX::void_pointer obtained by conversion from a value p.</td>
</tr>
<tr>
<td>z</td>
<td>a value of type XX::const_void_pointer obtained by conversion from a value q or a value w</td>
</tr>
<tr>
<td>r</td>
<td>a value of type T&amp; obtained by the expression *p.</td>
</tr>
<tr>
<td>s</td>
<td>a value of type const T&amp; obtained by the expression *q or by conversion from a value r.</td>
</tr>
<tr>
<td>u</td>
<td>a value of type YY::const_pointer obtained by calling YY::allocate, or else nullptr.</td>
</tr>
<tr>
<td>v</td>
<td>a value of type V</td>
</tr>
<tr>
<td>n</td>
<td>a value of type XX::size_type.</td>
</tr>
<tr>
<td>Args</td>
<td>a template parameter pack</td>
</tr>
<tr>
<td>args</td>
<td>a function parameter pack with the pattern Args&amp;&amp;</td>
</tr>
</tbody>
</table>
### Table 42 — Allocator requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X::pointer</code></td>
<td><code>T*</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>X::const_pointer</code></td>
<td><code>X::pointer</code> is convertible to <code>X::const_pointer</code></td>
<td>pointer-traits&lt;T&gt;::rebind&lt;const T&gt;</td>
<td></td>
</tr>
<tr>
<td><code>X::void_pointer</code></td>
<td><code>Y::void_pointer</code></td>
<td><code>X::pointer</code> is convertible to <code>X::const_pointer</code>, <code>X::void_pointer</code>, and <code>Y::void_pointer</code> are the same type.</td>
<td>pointer-traits&lt;X&gt;::rebind&lt;void&gt;</td>
</tr>
<tr>
<td><code>X::const_void_pointer</code></td>
<td><code>Y::const_void_pointer</code></td>
<td><code>X::pointer</code> is convertible to <code>X::const_void_pointer</code>. <code>X::const_void_pointer</code> and <code>Y::const_void_pointer</code> are the same type.</td>
<td>pointer-traits&lt;X&gt;::rebind&lt;const void&gt;</td>
</tr>
<tr>
<td><code>X::value_type</code></td>
<td><code>Identical to T</code></td>
<td>Identical to <code>T</code></td>
<td></td>
</tr>
<tr>
<td><code>X::size_type</code></td>
<td><code>unsigned integral type</code></td>
<td>a type that can represent the size of the largest object in the allocation model.</td>
<td><code>size_t</code></td>
</tr>
<tr>
<td><code>X::difference_type</code></td>
<td><code>signed integral type</code></td>
<td>a type that can represent the difference between any two pointers in the allocation model.</td>
<td><code>ptrdiff_t</code></td>
</tr>
<tr>
<td><code>typename</code></td>
<td><code>Y</code></td>
<td>For all <code>U</code> (including <code>T</code>), <code>Y::template rebind&lt;T&gt;::other</code> is <code>X</code>.</td>
<td>See Note A, below.</td>
</tr>
<tr>
<td><code>*p</code></td>
<td><code>T&amp;</code></td>
<td><code>*q</code> refers to the same object as <code>*p</code></td>
<td></td>
</tr>
<tr>
<td><code>*q</code></td>
<td><code>const T&amp;</code></td>
<td><code>*q</code> refers to the same object as <code>*p</code></td>
<td></td>
</tr>
<tr>
<td><code>p-&gt;m</code></td>
<td><code>type of T::m</code></td>
<td><code>pre: (*p).m</code> is well-defined. equivalent to <code>(*p).m</code></td>
<td></td>
</tr>
<tr>
<td><code>q-&gt;m</code></td>
<td><code>type of T::m</code></td>
<td><code>pre: (*q).m</code> is well-defined. equivalent to <code>(*q).m</code></td>
<td></td>
</tr>
<tr>
<td><code>static_cast&lt;X::pointer&gt;(w)</code></td>
<td><code>X::pointer</code></td>
<td>static_cast&lt;X::pointer&gt;(w) == <code>p</code></td>
<td></td>
</tr>
<tr>
<td><code>static_cast&lt;X::const_pointer&gt;(z)</code></td>
<td><code>X::const_pointer</code></td>
<td>static_cast&lt;X::const_pointer&gt;(z) == <code>q</code></td>
<td></td>
</tr>
</tbody>
</table>
## Table 42 — Allocator requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.allocate(n)</td>
<td>X::pointer</td>
<td>Memory is allocated for n objects of type T but objects are not constructed. allocate may raise an appropriate exception. [Note: If n == 0, the return value is unspecified. — end note]</td>
<td>a.allocate(n)</td>
</tr>
<tr>
<td>a.allocate(n, u)</td>
<td>X::pointer</td>
<td>Same as a.allocate(n). The use of u is unspecified, but it is intended as an aid to locality.</td>
<td>a.allocate(n)</td>
</tr>
<tr>
<td>a.deallocate(p, n)</td>
<td>(not used)</td>
<td>All n T objects in the area pointed to by p shall be destroyed prior to this call. n shall match the value passed to allocate to obtain this memory. Does not throw exceptions. [Note:p shall not be singular. — end note]</td>
<td></td>
</tr>
<tr>
<td>a.max_size()</td>
<td>X::size_type</td>
<td>the largest value that can meaningfully be passed to X::allocate() numeric_limits&lt;package&gt;::max()</td>
<td></td>
</tr>
<tr>
<td>a1 == a2</td>
<td>bool</td>
<td>returns true only if storage allocated from each can be deallocated via the other. operator== shall be reflexive, symmetric, and transitive, and shall not exit via an exception.</td>
<td></td>
</tr>
<tr>
<td>a1 != a2</td>
<td>bool</td>
<td>same as !(a1 == a2)</td>
<td></td>
</tr>
<tr>
<td>a == b</td>
<td>bool</td>
<td>same as a == Y::rebind&lt;T&gt;::other(b)</td>
<td></td>
</tr>
<tr>
<td>a != b</td>
<td>bool</td>
<td>same as !(a == b)</td>
<td></td>
</tr>
<tr>
<td>X a1(a);</td>
<td></td>
<td>Shall not exit via an exception. post: a1 == a</td>
<td></td>
</tr>
<tr>
<td>X a(b);</td>
<td></td>
<td>Shall not exit via an exception. post: Y(a) == b, a == X(b)</td>
<td></td>
</tr>
<tr>
<td>X a1(move(a));</td>
<td></td>
<td>Shall not exit via an exception. post: a1 equals the prior value of a.</td>
<td></td>
</tr>
<tr>
<td>X a(move(b));</td>
<td></td>
<td>Shall not exit via an exception. post: a equals the prior value of X(b).</td>
<td></td>
</tr>
<tr>
<td>a.construct(c, args)</td>
<td>(not used)</td>
<td>Effect: Constructs an object of type C at c ::new ((void*)c) C(forward&lt;Args&gt;(args)...)}</td>
<td></td>
</tr>
<tr>
<td>a.destroy(c)</td>
<td>(not used)</td>
<td>Effect: Destroys the object at c c-&gt;~C()</td>
<td></td>
</tr>
</tbody>
</table>
Table 42 — Allocator requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.select_on_container_copy_construction()</code></td>
<td>X</td>
<td>Typically returns either <code>a</code> or <code>X()</code></td>
<td>return <code>a</code>;</td>
</tr>
<tr>
<td><code>X::propagate_on_container_copy_assignment</code></td>
<td>Identical to or derived from <code>true_type</code> or <code>false_type</code></td>
<td><code>true_type</code> only if an allocator of type <code>X</code> should be copied when the client container is copy-assigned.</td>
<td><code>false_type</code></td>
</tr>
<tr>
<td><code>X::propagate_on_container_move_assignment</code></td>
<td>Identical to or derived from <code>true_type</code> or <code>false_type</code></td>
<td><code>true_type</code> only if an allocator of type <code>X</code> should be copied when the client container is move-assigned.</td>
<td><code>false_type</code></td>
</tr>
<tr>
<td><code>X::propagate_on_container_swap</code></td>
<td>Identical to or derived from <code>true_type</code> or <code>false_type</code></td>
<td><code>true_type</code> only if an allocator of type <code>X</code> should be swapped when the client container is swapped.</td>
<td><code>false_type</code></td>
</tr>
</tbody>
</table>

3 Note A: The member class template `rebind` in the table above is effectively a typedef template. [Note: in general, if the name `Allocator` is bound to `SomeAllocator<T>`, then `Allocator::rebind<U>::other` is the same type as `SomeAllocator<U>` where `SomeAllocator<T>::value_type` is `T` and `SomeAllocator<U>::value_type` is `U`. — end note] If Allocator is a class template instantiation of the form `SomeAllocator<T, Args>`, where `Args` is zero or more type arguments, and Allocator does not supply a `rebind` member template, the standard allocator_traits template uses `SomeAllocator<U, Args>` in place of `Allocator::rebind<U>::other` by default. For allocator types that are not template instantiations of the above form, no default is provided.

4 The `X::pointer`, `X::const_pointer`, `X::void_pointer`, and `X::const_void_pointer` types shall satisfy the requirements of NullablePointer (20.2.3). No constructor, comparison operator, copy operation, move operation, or swap operation on these types shall exit via an exception. `X::pointer` and `X::const_pointer` shall also satisfy the requirements for a random access iterator (24.2).

5 An allocator may constrain the types on which it can be instantiated and the arguments for which its `construct` member may be called. If a type cannot be used with a particular allocator, the allocator class or the call to `construct` may fail to instantiate.

[Example: the following is an allocator class template supporting the minimal interface that satisfies the requirements of Table 42:]

```cpp
template <class Tp>
struct SimpleAllocator {
    typedef Tp value_type;
    SimpleAllocator(ctor args);

    template <class T> SimpleAllocator(const SimpleAllocator<T>& other);

    Tp *allocate(std::size_t n);
    void deallocate(Tp *p, std::size_t n);
};
```

233) 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.

§ 20.2.5
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]

20.3 Utility components [utility]

This subclause contains some basic function and class templates that are used throughout the rest of the library.

Header <utility> synopsis

```cpp
#include <initializer_list>

namespace std {
    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&);
    }

    template<class T> void swap(T& a, T& b);
    template <class T, size_t N> void swap(T (&a)[N], T (&b)[N]);

    template <class T, class U> T& forward(U&);
    template <class T> typename remove_reference<T>::type&& move(T&&);
    template <class T> typename conditional<
        !has_nothrow_move_constructor<T>::value && has_copy_constructor<T>::value,
        const T&, T&&>::type move_if_noexcept(T& x);

    template <class T> typename add_rvalue_reference<T>::type declval() noexcept;
    // as unevaluated operand

    template <class T1, class T2> struct pair;
    template <class T1, class T2> bool operator==(const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2> bool operator< (const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2> bool operator!= (const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2> bool operator> (const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2> bool operator>=(const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2> void swap(pair<T1,T2>&, pair<T1,T2>&);
```

§ 20.3 488
struct piecewise_construct_t { };
constexpr piecewise_construct_t piecewise_construct = piecewise_construct_t();
template <class... Types> class tuple;  // defined in <tuple>

// 20.3.5, tuple-like access to pair:
template <class T> class tuple_size;
template <size_t I, class T> class tuple_element;

template <class T1, class T2> struct tuple_size<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> >;

typename tuple_element<0, std::pair<T1, T2> >::type& get(std::pair<T1, T2>&);
template <size_t I, class T1, class T2> const
type const tuple_element<I, std::pair<T1, T2> >::type& get(const std::pair<T1, T2>&);

// 20.3.5.4, pair range access:
template <class InputIterator>
InputIterator begin(const std::pair<InputIterator, InputIterator>& p);
template <class InputIterator>
InputIterator end(const std::pair<InputIterator, InputIterator>& p);

20.3.1 Operators [operators]

To avoid redundant definitions of operator! = out of operator== and operators >, <=, and >= out of operator<, the library provides the following:

template <class T> bool operator!=(const T& x, const T& y);

- Requires: Type T is EqualityComparable (31).
- Returns: !(x == y).

template <class T> bool operator>(const T& x, const T& y);

- Requires: Type T is LessThanComparable (32).
- Returns: y < x.

template <class T> bool operator<=(const T& x, const T& y);

- Requires: Type T is LessThanComparable (32).
- Returns: !(y < x).

template <class T> bool operator>=(const T& x, const T& y);

- Requires: Type T is LessThanComparable (32).
- Returns: !(x < y).

In this library, whenever a declaration is provided for an operator! =, operator>, operator> =, or operator< =, and requirements and semantics are not explicitly provided, the requirements and semantics are as specified in this Clause.
20.3.2 swap

template<class T> void swap(T& a, T& b);

Requires: Type T shall be MoveConstructible (34) and MoveAssignable (36).

Effects: Exchanges values stored in two locations.

template<class T, size_t N>
void swap(T (&a)[N], T (&b)[N]);

Requires: a[i] shall be swappable with (20.2.2) b[i] for all i in the range [0,N).

Effects: swap_ranges(a, a + N, b)

20.3.3 forward/move helpers

The library provides templated helper functions to simplify applying move semantics to an lvalue and to simplify the implementation of forwarding functions.

template <class T, class U> T&& forward(U&& u);

Returns: static_cast<T&&>(u).

Remarks: if the following constraints are not met, this signature shall not participate in overload resolution:

— the type formed by remove_reference<U>::type* is implicitly convertible to the type remove_reference<T>::type*; and

— if T is an lvalue reference type, then U is also an lvalue reference type.

Example:

template <class T, class A1, class A2>
shared_ptr<T> factory(A1&& a1, A2&& a2) {
    return shared_ptr<T>(new T(std::forward<A1>(a1), std::forward<A2>(a2)));
}

struct A {
    A(int&, const double&);
};

void g() {
    shared_ptr<A> sp1 = factory<A>(2, 1.414); // error: 2 will not bind to int&
    int i = 2;
    shared_ptr<A> sp2 = factory<A>(i, 1.414); // OK
}

In the first call to factory, A1 is deduced as int, so 2 is forwarded to A’s constructor as an rvalue. In the second call to factory, A1 is deduced as int&, so i is forwarded to A’s constructor as an lvalue. In both cases, A2 is deduced as double, so 1.414 is forwarded to A’s constructor as an rvalue.

— end example

template <class T> typename remove_reference<T>::type& move(T&& t);

Returns: t.

§ 20.3.3
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 ]

template <class T> typename conditional< 
    !has_nothrow_move_constructor<T>::value && has_copy_constructor<T>::value, 
    const T&, T&&>::type move_if_noexcept(T& x);

Returns: std::move(t)

20.3.4 Function template declval [declval]

1 The library provides the function template declval to simplify the definition of expressions which occur as unevaluated operands (Clause 5).

template <class T> 
typename add_rvalue_reference<T>::type declval() noexcept; // as unevaluated operand

2 Remarks: If this function is used (as defined by (3.2)), the program is ill-formed.

3 Remarks: The template parameter T of declval may be an incomplete type.

[Example: 
    template <class To, class From> 
    decltype(static_cast<To>(declval<From>())) convert(From&); 
]

declares a function template convert which only participates in overloading if the type From can be explicitly converted to type To. For another example see class template common_type (20.7.6.6). — end example ]

20.3.5 Pairs [pairs]

20.3.5.1 In general [pairs.general]

1 The library provides a template for heterogeneous pairs of values. The library also provides a matching
function template to simplify their construction and several templates that provide access to `pair` objects as if they were `tuple` objects (see 20.4.2.5 and 20.4.2.6).

### 20.3.5.2 Class template pair

```cpp
namespace std {
  template <class T1, class T2>
  struct pair {
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;

    pair(const pair&) = default;
    constexpr pair();
    pair(const T1& x, const T2& y);
    template<class U, class V> pair(U&& x, V&& y);
    template<class U, class V> pair(const pair<U, V>& p);
    template<class U, class V> pair(pair<U, V>&& p);
    template <class... Args1, class... Args2>
    pair(piecewise_construct_t,
         tuple<Args1...> first_args, tuple<Args2...> second_args);

    template<class U, class V> pair& operator=(const pair<U, V>& p);
    pair& operator=(pair&& p);
    template<class U, class V> pair& operator=(pair<U, V>&& p);
    void swap(pair& p);
  }
}
```

```cpp
constexpr pair();
```

1. **Effects:** Initializes its members as if implemented: `pair() : first(), second() { }`

```cpp
pair(const T1& x, const T2& y);
```

2. **Effects:** The constructor initializes `first` with `x` and `second` with `y`.

```cpp
template<class U, class V> pair(U&& x, V&& y);
```

3. **Effects:** The constructor initializes `first` with `std::forward<U>(x)` and `second` with `std::forward<V>(y)`.

4. **Remarks:** If `U` is not implicitly convertible to `first_type` or `V` is not implicitly convertible to `second_type` this constructor shall not participate in overload resolution.

```cpp
template<class U, class V> pair(const pair<U, V>& p);
```

5. **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);
```

6. **Effects:** The constructor initializes `first` with `std::move(p.first)` and `second` with `std::move(p.second)`.

```cpp
template<class... Args1, class... Args2>
pair(piecewise_construct_t,
     tuple<Args1...> first_args, tuple<Args2...> second_args);
```

§ 20.3.5.2


Requires: All the types in Args1 and Args2 shall be CopyConstructible (Table 35). T1 shall be constructible from Args1. T2 shall be constructible from Args2.

Effects: The constructor initializes first with arguments of types Args1... obtained by forwarding the elements of first_args and initializes second with arguments of types Args2... obtained by forwarding the elements of second_args. (Here, forwarding an element x of type U within a tuple object means calling std::forward<U>(x).) This form of construction, whereby constructor arguments for first and second are each provided in a separate tuple object, is called piecewise construction.

\[
\text{template<class U, class V> pair& operator=(const pair<U, V>& p);}\
\]

Requires: T1 shall satisfy the requirements of CopyAssignable from U. T2 shall satisfy the requirements of CopyAssignable from V.

Effects: Assigns p.first to first and p.second to second.

Returns: *this.

\[
\text{pair& operator=(pair&& p);}\
\]

Effects: Assigns to first with std::move(p.first) and to second with std::move(p.second).

Returns: *this.

\[
\text{template<class U, class V> pair& operator=(pair<U, V>&& p);}\
\]

Effects: Assigns to first with std::move(p.first) and to second with std::move(p.second).

Returns: *this.

\[
\text{void swap(pair& p);}\
\]

Requires: first shall be swappable with (20.2.2) p.first and second shall be swappable with p.second.

Effects: Swaps first with p.first and second with p.second.

\[
\text{template <class T1, class T2> bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y);}\
\]

Returns: x.first == y.first && x.second == y.second.

\[
\text{template <class T1, class T2> bool operator<(const pair<T1, T2>& x, const pair<T1, T2>& y);}\
\]

Returns: x.first < y.first || (!(y.first < x.first) && x.second < y.second).

\[
\text{template <class T1, class T2> bool operator!=(const pair<T1, T2>& x, const pair<T1, T2>& y);}\
\]

Returns: !(x == y)

\[
\text{template <class T1, class T2> bool operator>(const pair<T1, T2>& x, const pair<T1, T2>& y);}\
\]

Returns: y < x

\[
\text{template <class T1, class T2> bool operator>=(const pair<T1, T2>& x, const pair<T1, T2>& y);}\
\]

Returns: !(x < y)

\[\text{§ 20.3.5.2}\]
template <class T1, class T2>
bool operator<=(const pair<T1, T2>& x, const pair<T1, T2>& y);
23  
Returns: !(y < x)

template <class T1, class T2> void swap(pair<T1, T2>& x, pair<T1, T2>& y);
24  
Effects: x.swap(y)

template <class T1, class T2>
pair<V1, V2> make_pair(T1&&, T2&&);
25  
Returns:
pair<V1, V2>(std::forward<T1>(x), std::forward<T2>(y));

where V1 and V2 are determined as follows: Let Ui be decay<Ti>::type for each Ti. Then each Vi is X& if Ui equals reference_wrapper<X>, otherwise Vi is Ui.

[Example: In place of:
return pair<int, double>(5, 3.1415926); // explicit types

a C++ program may contain:
return make_pair(5, 3.1415926); // types are deduced

— end example]

20.3.5.3 Tuple-like access to pair  [pair.astuple]
tuple_size<pair<T1, T2>>>::value
1  
Returns: integral constant expression.
2  
Value: 2.

tuple_element<0, pair<T1, T2>>::type
3  
Value: the type T1.

tuple_element<1, pair<T1, T2>>::type
4  
Value: the type T2.

template<
size_t I, class T1, class T2>
typename tuple_element<I, std::pair<T1, T2>>::type&
get(pair<T1, T2>&);

template<
size_t I, class T1, class T2>
const typename tuple_element<I, std::pair<T1, T2>>::type&
get(const pair<T1, T2>&);

5  
Returns: If I == 0 returns p.first; if I == 1 returns p.second; otherwise the program is ill-formed.
6  
Throws: nothing.

20.3.5.4 pair range access  [pair.range]
template <class InputIterator>
InputIterator begin(const std::pair<InputIterator, InputIterator>& p);

§ 20.3.5.4
template <class InputIterator>
    InputIterator end(const std::pair<InputIterator, InputIterator>& p);

Returns: p.second.

20.3.5.5 Piecewise construction

struct piecewise_construct_t { };  
constexpr piecewise_construct_t piecewise_construct = piecewise_construct_t();

The struct piecewise_construct_t is an empty structure type used as a unique type to disambiguate
constructor and function overloading. Specifically, pair has a constructor with piecewise_construct_t as
the first argument, immediately followed by two tuple (20.4) arguments used for piecewise construction
of the elements of the pair object.

20.4 Tuples

20.4.1 In general

This subclause describes the tuple library that provides a tuple type as the class template tuple that can
be instantiated with any number of arguments. Each template argument specifies the type of an element
in the tuple. Consequently, tuples are heterogeneous, fixed-size collections of values. An instantiation of
tuple with two arguments is similar to an instantiation of pair with the same two arguments. See 20.3.5.

Header <tuple> synopsis

namespace std {
    // 20.4.2, class template tuple:
    template <class... Types> class tuple;

    // 20.4.2.4, tuple creation functions:
    const unspecified ignore;
    template <class... Types>
        tuple<Types...> make_tuple(Types&&...);
    template <class... Types>
        tuple<Types...> pack_arguments(Types&&...);
    template<class... Types>
        tuple<Types&...> tie(Types&...);
    template <class... TTypes, class... UTypes>
        tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>&, const tuple<UTypes...>&);
    template <class... TTypes, class... UTypes>
        tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&& , const tuple<UTypes...>&);
    template <class... TTypes, class... UTypes>
        tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&, tuple<UTypes...>&&);
    template <class... TTypes, class... UTypes>
        tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&& , tuple<UTypes...>&&);

    // 20.4.2.5, tuple helper classes:
    template <class T> class tuple_size; // undefined
    template <class... Types> class tuple_size<tuple<Types...> >;
template <size_t I, class T> class tuple_element; // undefined
template <size_t I, class... Types> class tuple_element<I, tuple<Types...>>;

// 20.4.2.6, element access:
template <size_t I, class... Types>
  typename tuple_element<I, tuple<Types...>>::type& get(tuple<Types...>&);
template <size_t I, class... types>
  typename tuple_element<I, tuple<Types...>>::type const& get(const tuple<Types...>&);

// 20.4.2.7, relational operators:
template<class... TTypes, class... UTypes>
  bool operator==(const tuple<TTypes...>&, const tuple<UTypes...>&);
template<class... TTypes, class... UTypes>
  bool operator<(const tuple<TTypes...>&, const tuple<UTypes...>&);

// 20.4.2.8, allocator-related traits
template <class... Types, class Alloc>
  struct uses_allocator<tuple<Types...>, Alloc>;

// 20.4.2.9, specialized algorithms:
template <class... Types>
  void swap(tuple<Types...>& x, tuple<Types...>& y);

// 20.4.2.10, tuple range access:
template <class InputIterator>
  InputIterator begin(const std::tuple<InputIterator, InputIterator>& t);
template <class InputIterator>
  InputIterator end(const std::tuple<InputIterator, InputIterator>& t);

20.4.2 Class template tuple

namespace std {
  template <class... Types>
  class tuple {
    public:

      // 20.4.2.1, tuple construction
      constexpr tuple();
      explicit tuple(const Types&...);
      template <class... UTypes>
        explicit tuple(UTypes&&...);

      tuple(const tuple&) = default;
      tuple(tuple&&);

      template <class... UTypes>
For each `tuple` constructor, an exception is thrown only if the construction of one of the types in `Types` throws an exception.

```
class tuple;
```

```cpp
tuple(const tuple<UTypes...>&);
tuple(tuple<UTypes...>&&);
```

template <class U1, class U2>
tuple(const pair<U1, U2>&);  // iff sizeof...(Types) == 2
tuple<class U1, class U2>
tuple(pair<U1, U2>&&);  // iff sizeof...(Types) == 2

```cpp
// allocator-extended constructors
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&);
tuple(allocator_arg_t, const Alloc& a, tuple&&);
tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);
tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);
tuple<class Alloc, class U1, class U2>
tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);
tuple<class Alloc, class U1, class U2>
tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);
```

// 20.4.2.2, tuple assignment
tuple& operator=(const tuple&);
tuple& operator=(tuple&&);

template <class... UTypes>
tuple& operator=(const tuple<UTypes...>&);
tuple& operator=(tuple<UTypes...>&&);

template <class U1, class U2>
tuple& operator=(const pair<U1, U2>&);  // iff sizeof...(Types) == 2
tuple<class U1, class U2>
tuple& operator=(pair<U1, U2>&&);  // iff sizeof...(Types) == 2

// 20.4.2.3, tuple swap
void swap(tuple&);
```

20.4.2.1 Construction

1. For each `tuple` constructor, an exception is thrown only if the construction of one of the types in `Types` throws an exception.

   ```cpp
class tuple;
```

2. `constexpr tuple();`

   ```cpp
tuple<class... UTypes>
tuple(tuple<UTypes...>&&);
```

   ```cpp
template <class U1, class U2>
tuple(const pair<U1, U2>&);  // iff sizeof...(Types) == 2
tuple<class U1, class U2>
tuple(pair<U1, U2>&&);  // iff sizeof...(Types) == 2
```

   ```cpp
   // allocator-extended constructors
   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&>
tuple<allocator_arg_t, const Alloc& a, tuple&&>
tuple<allocator_arg_t, const Alloc& a, tuple<UTypes...>&&>
tuple<allocator_arg_t, const Alloc& a, tuple<UTypes...>&&>
tuple<class Alloc, class U1, class U2>
tuple<allocator_arg_t, const Alloc& a, const pair<U1, U2>&>
tuple<class Alloc, class U1, class U2>
tuple<allocator_arg_t, const Alloc& a, pair<U1, U2>&&>
```

20.4.2.2, tuple assignment
tuple& operator=(const tuple&);
tuple& operator=(tuple&&);

template <class... UTypes>
tuple& operator=(const tuple<UTypes...>&);
tuple& operator=(tuple<UTypes...>&&);

template <class U1, class U2>
tuple& operator=(const pair<U1, U2>&);  // iff sizeof...(Types) == 2
tuple<class U1, class U2>
tuple& operator=(pair<U1, U2>&&);  // iff sizeof...(Types) == 2

20.4.2.3, tuple swap
void swap(tuple&);
```
**Effects:** Value initializes each element.

```cpp
explicit tuple(const Types&...);
```

**Requires:** Each type in `Types` shall be copy constructible.

**Effects:** Copy initializes each element with the value of the corresponding parameter.

```cpp
template <class... UTypes>
explicit tuple(UTypes&&... u);
```

**Requires:** Each type in `Types` shall satisfy the requirements of `MoveConstructible` (Table 34) from the corresponding type in `UTypes`. `sizeof...(Types) == sizeof...(UTypes)`.

**Effects:** Initializes the elements in the tuple with the corresponding value in `std::forward<UTypes>(u)`.

```cpp
tuple(const tuple& u) = default;
```

**Requires:** Each type in `Types` shall satisfy the requirements of `CopyConstructible` (Table 35).

**Effects:** Copy constructs each element of `*this` with the corresponding element of `u`.

```cpp
tuple(tuple&& u);
```

**Requires:** Each type in `Types` shall satisfy the requirements of `MoveConstructible` (Table 34).

**Effects:** Move-constructs each element of `*this` with the corresponding element of `u`.

```cpp
template <class... UTypes> tuple(const tuple<UTypes...>& u);
```

**Requires:** Each type in `Types` shall be constructible from the corresponding type in `UTypes`. `sizeof...(Types) == sizeof...(UTypes)`.

**Effects:** Constructs each element of `*this` with the corresponding element of `u`.

```cpp
 noting enable_if can be used to make the converting constructor and assignment operator exist only in the cases where the source and target have the same number of elements. — end note
```

```cpp
template <class... UTypes> tuple(tuple<UTypes...>&& u);
```

**Requires:** Each type in `Types` shall satisfy the requirements of `MoveConstructible` (Table 34) from the corresponding type in `UTypes`. `sizeof...(Types) == sizeof...(UTypes)`.

**Effects:** Move-constructs each element of `*this` with the corresponding element of `u`.

```cpp
 noting enable_if can be used to make the converting constructor and assignment operator exist only in the cases where the source and target have the same number of elements. — end note
```

```cpp
template <class U1, class U2> tuple(const pair<U1, U2>& u);
```

**Requires:** The first type in `Types` shall be constructible from `U1` and the second type in `Types` shall be constructible from `U2`. `sizeof...(Types) == 2`.

**Effects:** Constructs the first element with `u.first` and the second element with `u.second`.

```cpp
template <class U1, class U2> tuple(pair<U1, U2>&& u);
```

**Requires:** The first type in `Types` shall satisfy the requirements of `MoveConstructible` (Table 34) 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)`.

§ 20.4.2.1 498
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&&...);
template <class Alloc>
    tuple(allocator_arg_t, const Alloc& a, const tuple&);
template <class Alloc>
    tuple(allocator_arg_t, const Alloc& a, tuple&&);
template <class Alloc, class... UTypes>
    tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
template <class Alloc, class... UTypes>
    tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);
template <class Alloc, class U1, class U2>
    tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);
template <class Alloc, class U1, class U2>
    tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);

21
    Requires: Alloc shall meet the requirements for an Allocator (20.2.5).
22
    Effects: Equivalent to the preceding constructors except that each element is constructed with uses-
alloc tor construction (20.9.2.2).

20.4.2.2 Assignment

1  For each tuple assignment operator, an exception is thrown only if the assignment of one of the types in
Types throws an exception.

tuple& operator=(const tuple& u);
2
    Requires: Each type in Types shall be CopyAssignable (Table 37).
3
    Effects: Assigns each element of u to the corresponding element of *this.
4
    Returns: *this

tuple& operator=(tuple&& u);
5
    Requires: Each type in Types shall shall satisfy the requirements of MoveAssignable (Table 36).
6
    Effects: Move-assigns each element of u to the corresponding element of *this.
7
    Returns: *this.

template <class... UTypes>
    tuple& operator=(const tuple<UTypes...>& u);
8
    Requires: Each type in Types shall be Assignable from the corresponding type in UTypes.
9
    Effects: Assigns each element of u to the corresponding element of *this.
10
    Returns: *this

template <class... UTypes>
    tuple& operator=(tuple<UTypes...>&& u);
11
    Requires: Each type in Types shall satisfy the requirements of MoveAssignable (Table 36) from the
    corresponding type in UTypes. sizeof...(Types) == sizeof...(UTypes).

§ 20.4.2.2
Effects: Move-assigns each element of \( u \) to the corresponding element of \(*\text{this}\).

Returns: \(*\text{this}\).

template <class U1, class U2> tuple& operator=(const pair<U1, U2>&& u);

Requires: The first type in \( \text{Types} \) shall satisfy the requirements of \textit{MoveAssignable} (Table 36) from \( U1 \) and the second type in \( \text{Types} \) shall satisfy the requirements of \textit{MoveAssignable} (Table 36) from \( U2. \ sizeof\ldots(\text{Types}) == 2 \).

Effects: Assigns \( u.\text{first} \) to the first element of \(*\text{this}\) and \( u.\text{second} \) to the second element of \(*\text{this}\).

Returns: \(*\text{this}\).

[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 likely the same as the effect of the element-wise construction would have been. However, 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 \texttt{enable_if} template or other tools for constrained templates. — end note]

template <class U1, class U2> tuple& operator=(pair<U1, U2>&& u);

Requires: The first type in \( \text{Types} \) shall be \textit{Assignable} from \( U1 \) and the second type in \( \text{Types} \) shall be \textit{Assignable} from \( U2. \ sizeof\ldots(\text{Types}) == 2 \).

Effects: Assigns \( \text{std::move}(u.\text{first}) \) to the first element of \(*\text{this}\) and \( \text{std::move}(u.\text{second}) \) to the second element of \(*\text{this}\).

Returns: \(*\text{this}\).

### 20.4.2.3 swap

[tuple.swap]

```cpp
void swap(tuple& rhs);
```

Requires: Each element in \(*\text{this}\) shall be swappable with (20.2.2) the corresponding element in \( \text{rhs} \).

Effects: Calls \textit{swap} for each element in \(*\text{this}\) and its corresponding element in \( \text{rhs} \).

Throws: Nothing unless one of the element-wise \textit{swap} calls throws an exception.

### 20.4.2.4 Tuple creation functions

[tuple.creation]

```cpp
template<class... Types>
tuple<VTypes...> make_tuple(Types&&... t);
```

Let \( Ui \) be \texttt{decay<Ti>::type} for each \( Ti \) in \( \text{Types} \). Then each \( Vi \) in \( \text{VTypes} \) is \( X\& \) if \( Ui \) equals \texttt{reference_wrapper<X>}, otherwise \( Vi \) is \( Ui \).

Returns: \( \text{tuple}<\text{VTypes...}>(\text{std::forward<Types>}(t)...) \).

[Example:

```cpp
int i; float j;
make_tuple(1, ref(i), cref(j))
```
creates a tuple of type
tuple<int, int&, const float&>

— end example]

template<class... Types>
tuple<ATypes...> pack_arguments(Types&&... t);

4 Types: Let \( T_i \) be each type in \( \text{Types} \). Then each corresponding \( A_i \) in \( \text{Atypes} \) is \( T_i\& \) if \( T_i \) is an array type and \( \text{std::add_rvalue_reference<Ti>::type} \) otherwise.

5 Effects: Constructs a tuple of references to the arguments in \( t \) suitable for forwarding as arguments to a function. Because the result may contain references to temporary variables, a program shall ensure that the return value of this function does not outlive any of its arguments. (e.g., the program should typically not store the result in a named variable).

6 Returns: \( \text{tuple<ATypes...>(std::forward<Types>(t)...)} \)

template<class... Types>
tuple<Types&...> tie(Types&... t);

7 Returns: \( \text{tuple<Types&>(t...)} \). When an argument in \( t \) is \( \text{ignore} \), assigning any value to the corresponding tuple element has no effect.

8 [ Example: tie functions allow one to create tuples that unpack tuples into variables. ignore can be used for elements that are not needed:

\[
\text{int i; std::string s;}
\text{tie(i, ignore, s) = make_tuple(42, 3.14, "C++");}
\text{// i == 42, s == "C++"}
\]

— end example]

template <class... TTypes, class... UTypes>
tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>& t, const tuple<UTypes...>& u);

9 Requires: All the types in \( \text{TTypes} \) shall be CopyConstructible (Table 35). All the types in \( \text{UTypes} \) shall be CopyConstructible (Table 35).

10 Returns: A \( \text{tuple} \) object constructed by copy constructing its first \( \text{sizeof...}(...\text{TTypes}) \) elements from the corresponding elements of \( t \) and copy constructing its last \( \text{sizeof...}(...\text{UTypes}) \) elements from the corresponding elements of \( u \).

template <class... TTypes, class... UTypes>
tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&& t, const tuple<UTypes...>& u);

11 Requires: All the types in \( \text{TTypes} \) shall be MoveConstructible (Table 34). All the types in \( \text{UTypes} \) shall be CopyConstructible (Table 35).

12 Returns: A \( \text{tuple} \) object constructed by move constructing its first \( \text{sizeof...}(...\text{TTypes}) \) elements from the corresponding elements of \( t \) and copy constructing its last \( \text{sizeof...}(...\text{UTypes}) \) elements from the corresponding elements of \( u \).

template <class... TTypes, class... UTypes>
tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>& t, tuple<UTypes...>&& u);

13 Requires: All the types in \( \text{TTypes} \) shall be CopyConstructible (Table 35). All the types in \( \text{UTypes} \) shall be MoveConstructible (Table 34).
Returns: A tuple object constructed by copy constructing its first \texttt{sizeof...(TTypes)} elements from the corresponding elements of \texttt{t} and move constructing its last \texttt{sizeof...(UTypes)} elements from the corresponding elements of \texttt{u}.

\begin{verbatim}
template <class... TTypes, class... UTypes>
tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&& t, tuple<UTypes...>&& u);
\end{verbatim}

Requires: All the types in \texttt{TTypes} shall be MoveConstructible (Table 34). All the types in \texttt{UTypes} shall be MoveConstructible (Table 34).

Returns: A tuple object constructed by move constructing its first \texttt{sizeof...(TTypes)} elements from the corresponding elements of \texttt{t} and move constructing its last \texttt{sizeof...(UTypes)} elements from the corresponding elements of \texttt{u}.

\section{20.4.2.5 Tuple helper classes} \hfill [tuple.helper]

template <class... Types>
class tuple_size<tuple<Types...> >
  : public integral_constant<size_t, sizeof...(Types)> { };

template <size_t I, class... Types>
class tuple_element<I, tuple<Types...> > { 
public:
  typedef TI type;
};

Requires: \texttt{I < sizeof...(Types)}. The program is ill-formed if \texttt{I} is out of bounds.

Type: \texttt{TI} is the type of the \texttt{I}th element of \texttt{Types}, where indexing is zero-based.

\section{20.4.2.6 Element access} \hfill [tuple.elem]

template <size_t I, class... Types>
typename tuple_element<I, tuple<Types...> >::type& get(tuple<Types...>& t);

Requires: \texttt{I < sizeof...(Types)}. The program is ill-formed if \texttt{I} is out of bounds.

Returns: A reference to the \texttt{I}th element of \texttt{t}, where indexing is zero-based.

Throws: nothing.

\begin{verbatim}
typename tuple_element<I, tuple<Types...> >::type const& get(const tuple<Types...>& t);
\end{verbatim}

Requires: \texttt{I < sizeof...(Types)}. The program is ill-formed if \texttt{I} is out of bounds.

Returns: A const reference to the \texttt{I}th element of \texttt{t}, where indexing is zero-based.

Throws: nothing.

[\textit{Note:} Constness is shallow. If a \texttt{T} in \texttt{Types} is some reference type \texttt{X&}, the return type is \texttt{X&}, not \texttt{const X&}. However, if the element type is non-reference type \texttt{T}, the return type is \texttt{const T&}. This is consistent with how constness is defined to work for member variables of reference type. — end note]

[\textit{Note:} The reason \texttt{get} is a nonmember function is that if this functionality had been provided as a member function, code where the type depended on a template parameter would have required using the \texttt{template} keyword. — end note]
20.4.2.7 Relational operators

\[
\text{template<} \text{class... TTypes, class... UTypes>}
\]

\[
\text{bool operator==(const tuple<TTypes...>& t, const tuple<UTypes...>& u);}\]

\text{1 Requires: For all } i, \text{ where } 0 \leq i \text{ and } i < \text{sizeof... (Types)}, \text{ get<}i\text{>(}t\text{)} = \text{ get<}i\text{>(}u\text{)} \text{ is a valid expression returning a type that is convertible to bool. sizeof... (TTypes) = sizeof... (UTypes).}

\text{2 Returns: true iff get<}i\text{>(}t\text{)} = \text{ get<}i\text{>(}u\text{)} \text{ for all } i. \text{ For any two zero-length tuples } e \text{ and } f, e == f \text{ returns true.}

\text{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.}

\[
\text{template<} \text{class... TTypes, class... UTypes>}
\]

\[
\text{bool operator<(const tuple<TTypes...>& t, const tuple<UTypes...>& u);}\]

\text{4 Requires: For all } i, \text{ where } 0 \leq i \text{ and } i < \text{sizeof... (Types)}, \text{ get<}i\text{>(}t\text{)} < \text{ get<}i\text{>(}u\text{)} \text{ is a valid expression returning a type that is convertible to bool. sizeof... (TTypes) = sizeof... (UTypes).}

\text{5 Returns: The result of a lexicographical comparison between } t \text{ and } u. \text{ The result is defined as: (bool)(get<0>(}t\text{)} < get<0>(}u\text{)}) || (! (bool)(get<0>(}u\text{)} < get<0>(}t\text{))) \& \& t_{\text{tail}} < u_{\text{tail}}, \text{ where } r_{\text{tail}} \text{ for some tuple } r \text{ is a tuple containing all but the first element of } r. \text{ For any two zero-length tuples } e \text{ and } f, e < f \text{ returns false.}

\[
\text{template<} \text{class... TTypes, class... UTypes>}
\]

\[
\text{bool operator!=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);}\]

\text{6 Returns: !(t == u).}

\[
\text{template<} \text{class... TTypes, class... UTypes>}
\]

\[
\text{bool operator>(const tuple<TTypes...>& t, const tuple<UTypes...>& u);}\]

\text{7 Returns: u < t.}

\[
\text{template<} \text{class... TTypes, class... UTypes>}
\]

\[
\text{bool operator<=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);}\]

\text{8 Returns: !(u < t).}

\[
\text{template<} \text{class... TTypes, class... UTypes>}
\]

\[
\text{bool operator>=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);}\]

\text{9 Returns: !(t < u).}

\text{[Note: The above definitions for comparison operators do not require } t_{\text{tail}} \text{ (or } u_{\text{tail}} \text{) to be constructed. It may not even be possible, as } t \text{ and } u \text{ are not required to be copy constructible. Also, all comparison operators are short circuited; they do not perform element accesses beyond what is required to determine the result of the comparison. — end note]}

20.4.2.8 Tuple traits

\text{template <class... Types, class Alloc>}

\[
\text{struct uses_allocator<tuple<Types...>, Alloc> : true_type { };}\]

\text{Requires: Alloc shall be an Allocator (20.2.5).}

§ 20.4.2.8
[Note: Specialization of this trait informs other library components that tuple can be constructed with an allocator, even though it does not have a nested allocator_type. — end note]

20.4.2.9 Tuple specialized algorithms [tuple.special]

```cpp
template <class... Types>
void swap(tuple<Types...>& x, tuple<Types...>& y);
```

Effects: x.swap(y)

20.4.2.10 tuple range access [tuple.range]

```cpp
template <class InputIterator>
InputIterator begin(const tuple<InputIterator, InputIterator>& t);
```

Returns: std::get<0>(t).

```cpp
template <class InputIterator>
InputIterator end(const tuple<InputIterator, InputIterator>& t);
```

Returns: std::get<1>(t).

20.5 Class template bitset [template.bitset]

Header <bitset> synopsis

```cpp
#include <string>
#include <iosfwd>
// for istream, ostream
namespace std {
    template <size_t N> class bitset;

    // 20.5.4 bitset operators:
    template <size_t N>
    bitset<N> operator&(const bitset<N>&, const bitset<N>&);
    template <size_t N>
    bitset<N> operator|(const bitset<N>&, const bitset<N>&);
    template <size_t N>
    bitset<N> operator^(const bitset<N>&, const bitset<N>&);
    template <class charT, class traits, size_t N>
    basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>& is, bitset<N>& x);
    template <class charT, class traits, size_t N>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);
}
```

The header <bitset> defines a class template and several related functions for representing and manipulating fixed-size sequences of bits.

```cpp
namespace std {
    template<size_t N> class bitset {
        public:
        // bit reference:
        class reference {
            friend class bitset;
            reference();
        }
    }
```
public:
  ~reference();
  reference& operator=(bool x); // for b[i] = x;
  reference& operator=(const reference&); // for b[i] = b[j];
  bool operator~() const; // flips the bit
  operator bool() const; // for x = b[i];
  reference& flip(); // for b[i].flip();
};

// 20.5.1 constructors:
constexpr bitset();
constexpr bitset(unsigned long long val);
template<class charT, class traits, class Allocator>
  explicit bitset(
      const basic_string<charT, traits, Allocator>& str,
      typename basic_string<charT, traits, Allocator>::size_type pos = 0,
      typename basic_string<charT, traits, Allocator>::size_type n =
      basic_string<charT, traits, Allocator>::npos,
      charT zero = charT('0'), charT one = charT('1'));
explicit bitset(const char *str);

// 20.5.2 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];

unsigned long to_ulong() const;
unsigned long long to_ullong() const;
template <class charT = char,
          class traits = char_traits<charT>,
          class Allocator = allocator<charT>>
  basic_string<charT, traits, Allocator>
to_string(charT zero = charT('0'), charT one = charT('1')) const;
size_t count() const;
constexpr size_t size();
bool operator==(const bitset<N>& rhs) const;
bool operator!=(const bitset<N>& rhs) const;
bool test(size_t pos);
bool all() const;
bool any() const;
bool none() const;
bitset<N> operator<<(size_t pos) const;

§ 20.5
The class template `bitset<N>` describes an object that can store a sequence consisting of a fixed number of bits, `N`.

Each bit represents either the value zero (reset) or one (set). To toggle a bit is to change the value zero to one, or the value one to zero. Each bit has a non-negative position `pos`. When converting between an object of class `bitset<N>` and a value of some integral type, bit position `pos` corresponds to the bit value `1 <<pos`. The integral value corresponding to two or more bits is the sum of their bit values.

The functions described in this subclause can report three kinds of errors, each associated with a distinct exception:

- an *invalid-argument* error is associated with exceptions of type `invalid_argument` (19.2.3);
- an *out-of-range* error is associated with exceptions of type `out_of_range` (19.2.5);
- an *overflow* error is associated with exceptions of type `overflow_error` (19.2.8).

### 20.5.1 `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 `M` bit positions to the corresponding bit values in `val`. `M` is the smaller of `N` and the number of bits in the value representation (section 3.9) of `unsigned long long`. If `M<N`, the remaining bit positions are initialized to zero.

```cpp
template <class charT, class traits, class Allocator>
explicit
bitset(const basic_string<charT, traits, Allocator>& str,
       typename basic_string<charT, traits, Allocator>::size_type pos = 0,
       typename basic_string<charT, traits, Allocator>::size_type n =
       basic_string<charT, traits, Allocator>::npos,
       charT zero = charT('0'), charT one = charT('1'));
```

*Requires:* `pos <= str.size()`.

*Throws:* `out_of_range` if `pos > str.size()`.

*Effects:* Determines the effective length `rlen` of the initializing string as the smaller of `n` and `str.size()` - `pos`.

The function then throws `invalid_argument` if any of the `rlen` characters in `str` beginning at position `pos` is other than `zero` or `one`. The function uses `traits::eq()` to compare the character values.

Otherwise, the function constructs an object of class `bitset<N>`, initializing the first `M` bit positions to values determined from the corresponding characters in the string `str`. `M` is the smaller of `N` and `rlen`.
An element of the constructed string has value zero if the corresponding character in \texttt{str}, beginning at position \texttt{pos}, is 0 \texttt{zero}. Otherwise, the element has the value 1. Character position \texttt{pos + M - 1} corresponds to bit position zero. Subsequent decreasing character positions correspond to increasing bit positions.

If \texttt{M < N}, remaining bit positions are initialized to zero.

\begin{verbatim}
explicit bitset(const char *str);

Effects: Constructs an object of class \texttt{bitset<N>} as if by \texttt{bitset(string(str))}.
\end{verbatim}

\section*{20.5.2 \texttt{bitset} members} 

\begin{verbatim}
bitset<N>& operator&=(const bitset<N>& rhs);

Effects: Clears each bit in \texttt{*this} for which the corresponding bit in \texttt{rhs} is clear, and leaves all other bits unchanged.

Returns: \texttt{*this}.

diset<N>& operator|=(const bitset<N>& rhs);

Effects: Sets each bit in \texttt{*this} for which the corresponding bit in \texttt{rhs} is set, and leaves all other bits unchanged.

Returns: \texttt{*this}.

bitset<N>& operator^=(const bitset<N>& rhs);

Effects: Toggles each bit in \texttt{*this} for which the corresponding bit in \texttt{rhs} is set, and leaves all other bits unchanged.

Returns: \texttt{*this}.

bitset<N>& operator<<=(size_t pos);

Effects: Replaces each bit at position \texttt{I} in \texttt{*this} with a value determined as follows:

\begin{itemize}
\item If \texttt{I < pos}, the new value is zero;
\item If \texttt{I >= pos}, the new value is the previous value of the bit at position \texttt{I - pos}.
\end{itemize}

Returns: \texttt{*this}.

bitset<N>& operator>>=(size_t pos);

Effects: Replaces each bit at position \texttt{I} in \texttt{*this} with a value determined as follows:

\begin{itemize}
\item If \texttt{pos >= N - I}, the new value is zero;
\item If \texttt{pos < N - I}, the new value is the previous value of the bit at position \texttt{I + pos}.
\end{itemize}

Returns: \texttt{*this}.

bitset<N>& set();

Effects: Sets all bits in \texttt{*this}.

Returns: \texttt{*this}.

bitset<N>& set(size_t pos, bool val = true);
\end{verbatim}
Requires: pos is valid

Throws: out_of_range if pos does not correspond to a valid bit position.

Effects: Stores a new value in the bit at position pos in *this. If val is nonzero, the stored value is one, otherwise it is zero.

Returns: *this.

bitset<N>& reset();

Effects: Resets all bits in *this.

Returns: *this.

bitset<N>& reset(size_t pos);

Requires: pos is valid

Throws: out_of_range if pos does not correspond to a valid bit position.

Effects: Resets the bit at position pos in *this.

Returns: *this.

bitset<N> operator~() const;

Effects: Constructs an object x of class bitset<N> and initializes it with *this.

Returns: x.flip().

bitset<N>& flip();

Effects: Toggles all bits in *this.

Returns: *this.

bitset<N>& flip(size_t pos);

Requires: pos is valid

Throws: out_of_range if pos does not correspond to a valid bit position.

Effects: Toggles the bit at position pos in *this.

Returns: *this.

unsigned long to_ulong() const;

Throws: overflow_error if the integral value x corresponding to the bits in *this cannot be represented as type unsigned long.

Returns: x.

unsigned long long to_ullong() const;

Throws: overflow_error if the integral value x corresponding to the bits in *this cannot be represented as type unsigned long long.

Returns: x.

template <class charT = char,
   class traits = char_traits<charT>,
   class Allocator = allocator<charT> >
basic_string<charT, traits, Allocator> 
to_string(charT zero = charT('0'), charT one = charT('1')) const;

Effects: Constructs a string object of the appropriate type and initializes it to a string of length \( N \) characters. Each character is determined by the value of its corresponding bit position in *this. Character position \( N - 1 \) corresponds to bit position zero. Subsequent decreasing character positions correspond to increasing bit positions. Bit value zero becomes the character zero, bit value one becomes the character one.

Returns: The created object.

size_t count() const;

Returns: A count of the number of bits set in *this.

constexpr size_t size();

Returns: \( N \).

bool operator==(const bitset<N>& rhs) const;

Returns: A nonzero value if the value of each bit in *this equals the value of the corresponding bit in rhs.

bool operator!=(const bitset<N>& rhs) const;

Returns: A nonzero value if !(*this == rhs).

bool test(size_t pos);

Requires: pos is valid

Throws: out_of_range if pos does not correspond to a valid bit position.

Returns: true if the bit at position pos in *this has the value one.

bool all() const;

Returns: count() == size()

bool any() const;

Returns: count() != 0

bool none() const;

Returns: count() == 0

bitset<N> operator<<(size_t pos) const;

Returns: bitset<N>(*this) <<= pos.

bitset<N> operator>>(size_t pos) const;

Returns: bitset<N>(*this) >>= pos.

constexpr bool operator[](size_t pos);

Requires: pos shall be valid.

Throws: nothing.

Returns: true if the bit at position pos in *this has the value one, otherwise false.
bitset<
>::reference operator[](size_t pos);

Requires: pos shall be valid.

Throws: nothing.

Returns: An object of type bitset<
>::reference such that (*this)[pos] == this->test(pos), and such that (*this)[pos] = val is equivalent to this->set(pos, val).

Remark: For the purpose of determining the presence of a data race (1.10), any access or update through the resulting reference potentially accesses or modifies, respectively, the entire underlying bitset.

20.5.3 Hash support

template <size_t N> struct hash<bitset<N>>;

Requires: the template specialization shall meet the requirements of class template hash (20.8.15).

20.5.4 bitset operators

bitset<
> operator&(const bitset<&
>& lhs, const bitset<&
>& rhs);

Returns: bitset<
>(lhs) &= rhs.

bitset<
> operator|(const bitset<&
>& lhs, const bitset<&
>& rhs);

Returns: bitset<
>(lhs) |= rhs.

bitset<
> operator^(const bitset<&
>& lhs, const bitset<&
>& rhs);

Returns: bitset<
>(lhs) ^= rhs.

template <class charT, class traits, size_t N>
basic_istream<charT, traits>& operator>>(
basic_istream<charT, traits>& is, bitset<N>& x);

Returns: is.

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_base::failbit) (which may throw ios_base::failure (27.5.4.3)).

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:
This subclause describes the ratio library. It provides a class template ratio which exactly represents any finite rational number with a numerator and denominator representable by compile-time constants of type intmax_t.

Throughout this subclause, if the template argument types R1 and R2 are not specializations of the ratio template, the program is ill-formed.

Header <ratio> synopsis

namespace std {
    template <intmax_t N, intmax_t D = 1> class ratio;

    // ratio arithmetic
    template <class R1, class R2> using ratio_add = see below;
    template <class R1, class R2> using ratio_subtract = see below;
    template <class R1, class R2> using ratio_multiply = see below;
    template <class R1, class R2> using ratio_divide = see below;

    // ratio comparison
    template <class R1, class R2> struct ratio_equal;
    template <class R1, class R2> struct ratio_not_equal;
    template <class R1, class R2> struct ratio_less;
    template <class R1, class R2> struct ratio_less_equal;
    template <class R1, class R2> struct ratio_greater;
    template <class R1, class R2> struct ratio_greater_equal;

    // convenience SI typedefs
    typedef ratio<1, 1000000000000000000000000> yocto; // see 20.6.4
    typedef ratio<1, 1000000000000000000000> zepto; // see 20.6.4
    typedef ratio<1, 1000000000000000000> atto;
    typedef ratio<1, 1000000000000000> femto;
    typedef ratio<1, 1000000000000> pico;
    typedef ratio<1, 1000000> nano;
    typedef ratio<1, 1000> micro;
    typedef ratio<1, 100> milli;
    typedef ratio<1, 10> centi;
    typedef ratio<10, 1> deci;
    typedef ratio<100, 1> hecto;
    typedef ratio<1000, 1> kilo;
    typedef ratio<10000, 1> mega;
    typedef ratio<100000, 1> giga;
    typedef ratio<1000000, 1> tera;
    typedef ratio<10000000, 1> peta;
    typedef ratio<100000000, 1> exa;
    typedef ratio<1000000000, 1> zetta; // see 20.6.4
    typedef ratio<10000000000, 1> yotta; // see 20.6.4
}

§ 20.6
20.6.1 Class template ratio

```cpp
namespace std {
    template <intmax_t N, intmax_t D = 1>
    class ratio {
        public:
            typedef ratio type;
            static constexpr intmax_t num;
            static constexpr intmax_t den;
    };
}
```

1. If the template argument \( D \) is zero or the absolute values of either of the template arguments \( N \) and \( D \) is not representable by type `intmax_t`, the program is ill-formed. [Note: These rules ensure that infinite ratios are avoided and that for any negative input, there exists a representable value of its absolute value which is positive. In a two's complement representation, this excludes the most negative value. — end note]

2. The static data members `num` and `den` shall have the following values, where `gcd` represents the greatest common divisor of the absolute values of \( N \) and \( D \):

   - `num` shall have the value `sign(N) * sign(D) * abs(N) / gcd`.
   - `den` shall have the value `abs(D) / gcd`.

20.6.2 Arithmetic on ratio types

1. Implementations may use other algorithms to compute these values. If overflow occurs, the program is ill-formed.

```cpp
template <class R1, class R2> using ratio_add = see below;
```

2. The type `ratio_add<R1, R2>` shall be a synonym for `ratio<T1, T2>` where \( T1 \) has the value \( R1::num * R2::den + R2::num * R1::den \) and \( T2 \) has the value \( R1::den * R2::den \).

```cpp
template <class R1, class R2> using ratio_subtract = see below;
```

3. The type `ratio_subtract<R1, R2>` shall be a synonym for `ratio<T1, T2>` where \( T1 \) has the value \( R1::num * R2::den - R2::num * R1::den \) and \( T2 \) has the value \( R1::den * R2::den \).

```cpp
template <class R1, class R2> using ratio_multiply = see below;
```

4. The type `ratio_multiply<R1, R2>` shall be a synonym for `ratio<T1, T2>` where \( T1 \) has the value \( R1::num * R2::num \) and \( T2 \) has the value \( R1::den * R2::den \).

```cpp
template <class R1, class R2> using ratio_divide = see below;
```

5. The type `ratio_divide<R1, R2>` shall be a synonym for `ratio<T1, T2>` where \( T1 \) has the value \( R1::num * R2::den \) and \( T2 \) has the value \( R1::den * R2::num \).

20.6.3 Comparison of ratio types

```cpp
template <class R1, class R2> struct ratio_equal
    : integral_constant<bool, see below> { };
```

1. If `R1::num == R2::num` and `R1::den == R2::den`, `ratio_equal<R1, R2>` shall be derived from `integral_constant<bool, true>`; otherwise it shall be derived from `integral_constant<bool, false>`.

§ 20.6.3
template <class R1, class R2> struct ratio_not_equal:
   integral_constant<bool, !ratio_equal<R1, R2>::value> { };  

template <class R1, class R2> struct ratio_less:
   integral_constant<bool, see below> { };  

2 If R1::num * R2::den < R2::num * R1::den, ratio_less<R1, R2> shall be derived from integral_constant<bool, true>; otherwise it shall be derived from integral_constant<bool, false>. Implementations may use other algorithms to compute this relationship to avoid overflow. If overflow occurs, the program is ill-formed.

template <class R1, class R2> struct ratio_less_equal:
   integral_constant<bool, !ratio_less<R2, R1>::value> { };  

template <class R1, class R2> struct ratio_greater:
   integral_constant<bool, ratio_less<R2, R1>::value> { };  

template <class R1, class R2> struct ratio_greater_equal:
   integral_constant<bool, !ratio_less<R1, R2>::value> { };  

20.6.4 SI types for ratio [ratio.si]  
1 For each of the typedefs yocto, zepto, zetta, and yotta, if both of the constants used in its specification are representable by intmax_t, the typedef shall be defined; if either of the constants is not representable by intmax_t, the typedef shall not be defined.

20.7 Metaprogramming and type traits [meta]  
1 This subclause describes components used by C++ programs, particularly in templates, to support the widest possible range of types, optimise template code usage, detect type related user errors, and perform type inference and transformation at compile time. It includes type classification traits, type property inspection traits, and type transformations. The type classification traits describe a complete taxonomy of all possible C++ types, and state where in that taxonomy a given type belongs. The type property inspection traits allow important characteristics of types or of combinations of types to be inspected. The type transformations allow certain properties of types to be manipulated.

20.7.1 Requirements [meta.rqmts]  
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 and unambiguously derived, directly or indirectly, from its BaseCharacteristic, which is a specialization of the template integral_constant (20.7.3), with the arguments to the template integral_constant determined by the requirements for the particular property being described. The member names of the BaseCharacteristic shall not be hidden and shall be unambiguously available in the UnaryTypeTrait.

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 and unambiguously derived, directly or indirectly, from its BaseCharacteristic, which is a specialization of the template integral_constant (20.7.3), with the arguments to the template integral_constant determined by the requirements for the particular relationship being described. The member names of the BaseCharacteristic shall not be hidden and shall be unambiguously available in the BinaryTypeTrait.
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.

### 20.7.2 Header `<type_traits>` synopsis

```cpp
namespace std {
    // 20.7.3, helper class:
    template <class T, T v> struct integral_constant;
    typedef integral_constant<bool, true> true_type;
    typedef integral_constant<bool, false> false_type;

    // 20.7.4.1, 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;

    // 20.7.4.2, composite type categories:
    template <class T> struct is_reference;
    template <class T> struct is_arithmetic;
    template <class T> struct is_fundamental;
    template <class T> struct is_object;
    template <class T> struct is_scalar;
    template <class T> struct is_compound;
    template <class T> struct is_member_pointer;

    // 20.7.4.3, type properties:
    template <class T> struct is_const;
    template <class T> struct is_volatile;
    template <class T> struct is_trivial;
    template <class T> struct is_trivially_copyable;
    template <class T> struct is_standard_layout;
    template <class T> struct is_pod;
    template <class T> struct is_literal_type;
    template <class T> struct is_empty;
    template <class T> struct is_polymorphic;
    template <class T> struct is_abstract;
    template <class T, class... Args> struct is_constructible;
    template <class T, class... Args> struct is_nothrow_constructible;
    template <class T> struct has_default_constructor;
    template <class T> struct has_copy_constructor;
    template <class T> struct has_copy_assign;
    template <class T> struct has_move_constructor;
    template <class T> struct has_move_assign;
    template <class T> struct has_trivial_default_constructor;
}
```
template <class T> struct has_trivial_copy_constructor;
template <class T> struct has_trivial_move_constructor;
template <class T> struct has_trivial_copy_assign;
template <class T> struct has_trivial_move_assign;
template <class T> struct has_trivialDestructor;
template <class T> struct has_nothrow_default_constructor;
template <class T> struct has_nothrow_move_constructor;
template <class T> struct has_nothrow_copy_constructor;
template <class T> struct has_nothrow_move_assign;
template <class T> struct has_virtualDestructor;
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;

// 20.7.5, 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;
template <class From, class To> struct is_explicitly_convertible;

// 20.7.6.1, 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;

// 20.7.6.2, reference modifications:
template <class T> struct remove_reference;
template <class T> struct add_lvalue_reference;
template <class T> struct add_rvalue_reference;

// 20.7.6.3, sign modifications:
template <class T> struct make_signed;
template <class T> struct make_unsigned;

// 20.7.6.4, array modifications:
template <class T> struct remove_extent;
template <class T> struct remove_all_extents;

// 20.7.6.5, pointer modifications:
template <class T> struct remove_pointer;
template <class T> struct add_pointer;

// 20.7.6.6, other transformations:
template <std::size_t Len, std::size_t Align> struct aligned_storage;
template <std::size_t Len, std::... 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;
template <class... T> struct common_type;
The behavior of a program that adds specializations for any of the class templates defined in this subclause is undefined unless otherwise specified.

### 20.7.3 Helper classes

```cpp
namespace std {
    template <class T, T v>
    struct integral_constant {
        static constexpr T value = v;
        typedef T value_type;
        typedef integral_constant<T, v> type;
        constexpr operator value_type() { return value; }
    };
    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.

### 20.7.4 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 (20.7.1) with a BaseCharacteristic of `true_type` if the corresponding condition is true, otherwise `false_type`.

### 20.7.4.1 Primary Type Categories

The primary type categories correspond to the descriptions given in section 3.9 of the C++ standard.

For any given type `T`, the result of applying one of these templates to `T` and to `cv-qualified T` shall yield the same result.

[Note: For any given type `T`, exactly one of the primary type categories has a `value` member that evaluates to `true`. — end note]

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt; struct is_void;</td>
<td><code>T</code> is void</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_integral;</td>
<td><code>T</code> is an integral type (3.9.1)</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_floating_point;</td>
<td><code>T</code> is a floating point type (3.9.1)</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_array;</td>
<td><code>T</code> is an array type (3.9.2) of known or unknown extent</td>
<td>Class template <code>array</code> (23.3.1) is not an array type.</td>
</tr>
</tbody>
</table>
### Table 43 — Primary type category predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt;</td>
<td>struct is_pointer;</td>
<td>T is a pointer type (<a href="#3.9.2">3.9.2</a>)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Includes pointers to functions but not pointers to non-static members.</td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>struct is_lvalue_reference;</td>
<td>T is an lvalue reference type (<a href="#8.3.2">8.3.2</a>)</td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>struct is_rvalue_reference;</td>
<td>T is an rvalue reference type (<a href="#8.3.2">8.3.2</a>)</td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>struct is_member_object_pointer;</td>
<td>T is a pointer to non-static data member</td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>struct is_member_function_pointer;</td>
<td>T is a pointer to non-static member function</td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>struct is_enum;</td>
<td>T is an enumeration type (<a href="#3.9.2">3.9.2</a>)</td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>struct is_union;</td>
<td>T is a union type (<a href="#3.9.2">3.9.2</a>)</td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>struct is_class;</td>
<td>T is a class type but not a union type (<a href="#3.9.2">3.9.2</a>)</td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>struct is_function;</td>
<td>T is a function type (<a href="#3.9.2">3.9.2</a>)</td>
</tr>
</tbody>
</table>

**20.7.4.2 Composite type traits**

1. These templates provide convenient compositions of the primary type categories, corresponding to the descriptions given in section [3.9](#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 44 — Composite type category predicates

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt;</td>
<td>struct is_reference;</td>
<td>T is an lvalue reference or an rvalue reference</td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>struct is_arithmetic;</td>
<td>T is an arithmetic type (<a href="#3.9.1">3.9.1</a>)</td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>struct is_fundamental;</td>
<td>T is a fundamental type (<a href="#3.9.1">3.9.1</a>)</td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>struct is_object;</td>
<td>T is an object type (<a href="#3.9">3.9</a>)</td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>struct is_scalar;</td>
<td>T is a scalar type (<a href="#3.9">3.9</a>)</td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>struct is_compound;</td>
<td>T is a compound type (<a href="#3.9.2">3.9.2</a>)</td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>struct is_member_pointer;</td>
<td>T is a pointer to non-static data member or non-static member function</td>
</tr>
</tbody>
</table>
20.7.4.3 Type properties [meta.unary.prop]

1. These templates provide access to some of the more important properties of types.
2. It is unspecified whether the library defines any full or partial specialisations of any of these templates.
3. For all of the class templates $X$ declared in this Clause, instantiating that template with a template-argument that is a class template specialization may result in the implicit instantiation of the template argument if and only if the semantics of $X$ require that the argument must be a complete type.

Table 45 — Type property predicates

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>template $&lt;$class $T$ $&gt;$ struct is_const;</td>
<td>$T$ is const-qualified (3.9.3)</td>
<td></td>
</tr>
<tr>
<td>template $&lt;$class $T$ $&gt;$ struct is_volatile;</td>
<td>$T$ is volatile-qualified (3.9.3)</td>
<td></td>
</tr>
<tr>
<td>template $&lt;$class $T$ $&gt;$ struct is_trivial;</td>
<td>$T$ is a trivial type (3.9)</td>
<td>$T$ shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template $&lt;$class $T$ $&gt;$ struct is_trivially_copyable;</td>
<td>$T$ is a trivially copyable type (3.9)</td>
<td>$T$ shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template $&lt;$class $T$ $&gt;$ struct is_standard_layout;</td>
<td>$T$ is a standard-layout type (3.9)</td>
<td>$T$ shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template $&lt;$class $T$ $&gt;$ struct is_pod;</td>
<td>$T$ is a POD type (3.9)</td>
<td>$T$ shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template $&lt;$class $T$ $&gt;$ struct is_literal_type;</td>
<td>$T$ is a literal type (3.9)</td>
<td>$T$ shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template $&lt;$class $T$ $&gt;$ struct is_empty;</td>
<td>$T$ is a class type, but not a union type, with no non-static data members other than bit-fields of length 0, no virtual member functions, no virtual base classes, and no base class $B$ for which is_empty$&lt;$B$&gt;::value$ is false.</td>
<td>$T$ shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template $&lt;$class $T$ $&gt;$ struct is_polymorphic;</td>
<td>$T$ is a polymorphic class (10.3)</td>
<td>$T$ shall be a complete type, type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
</tbody>
</table>
Table 45 — Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>template <code>&lt;class T&gt;</code> struct is_abstract;</td>
<td>T is an abstract class (10.4)</td>
<td>T shall be a complete type, type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template <code>&lt;class T, class... Args&gt;</code> struct is_constructible;</td>
<td>see below</td>
<td>T and all types in the parameter pack Args shall be complete types, (possibly cv-qualified) void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template <code>&lt;class T, class... Args&gt;</code> struct is_nothrow_constructible;</td>
<td>is_constructible&lt;T, Args...&gt;::value is true and the expression noexcept(CE) is true, where CE is defined below.</td>
<td>T and all types in the parameter pack Args shall be complete types, (possibly cv-qualified) void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template <code>&lt;class T&gt;</code> struct has_default_constructor;</td>
<td>is_constructible&lt;U&gt;::value is true, where U is remove_all_extents&lt;T&gt;::type.</td>
<td>T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template <code>&lt;class T&gt;</code> struct has_copy_constructor;</td>
<td>is_constructible&lt;U, const U&amp;&gt;::value is true, where U is remove_all_extents&lt;T&gt;::type.</td>
<td>T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template <code>&lt;class T&gt;</code> struct has_move_constructor;</td>
<td>T is neither const nor a reference type, and T is a trivial type (3.9) or the expression <em>(U</em>)0 = declval&lt;const U&amp;&gt;() is well-formed when treated as an unevaluated operand (Clause 5), where U is remove_all_extents&lt;T&gt;::type.</td>
<td>T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template <code>&lt;class T&gt;</code> struct has_move_assign;</td>
<td>is_constructible&lt;U, U&amp;&amp;&gt;::value is true, where U is remove_all_extents&lt;T&gt;::type.</td>
<td>T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
</tbody>
</table>
Table 45 — Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;class T&gt; struct has_move_assign;</code></td>
<td>(T) is neither <code>const</code> nor a reference type, and (T) is a trivial type (3.9) or the expression (\ast (U*)0 = \text{declval}&lt;U&gt;()) is well-formed when treated as an unevaluated operand (Clause 5), where (U) is <code>remove_all_extents&lt;T&gt;::type</code>.</td>
<td>(T) shall be a complete type, (possibly cv-qualified) <code>void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct has_trivial_default_constructor;</code></td>
<td>(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.</td>
<td>(T) shall be a complete type, (possibly cv-qualified) <code>void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct has_trivial_copy_constructor;</code></td>
<td>(T) is a trivial type (3.9) or a reference type or a class type whose copy constructors (12.8) are all trivial.</td>
<td>(T) shall be a complete type, (possibly cv-qualified) <code>void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct has_trivial_move_constructor;</code></td>
<td>(T) is a trivial type (3.9) or a reference type or a class type whose move constructors (12.8) are all trivial.</td>
<td>(T) shall be a complete type, (possibly cv-qualified) <code>void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct has_trivial_copy_assign;</code></td>
<td>(T) is neither <code>const</code> nor a reference type, and (T) is a trivial type (3.9) or a class type whose copy assignment operators (12.8) are all trivial.</td>
<td>(T) shall be a complete type, (possibly cv-qualified) <code>void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct has_trivial_move_assign;</code></td>
<td>(T) is neither <code>const</code> nor a reference type, and (T) is a trivial type (3.9) or a class type whose move assignment operators (12.8) are all trivial.</td>
<td>(T) shall be a complete type, (possibly cv-qualified) <code>void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td><code>template &lt;class T&gt; struct has_trivial_destructor;</code></td>
<td>(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.</td>
<td>(T) shall be a complete type, (possibly cv-qualified) <code>void</code>, or an array of unknown bound.</td>
</tr>
</tbody>
</table>
Table 45 — Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt; struct has_nothrow_default_constructor;</td>
<td>has_trivial_default_constructor&lt;T&gt;::value is true or is_nothrow_constructible&lt;U&gt;::value is true, where U is remove_all_extents&lt;T&gt;::type.</td>
<td>T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct has_nothrow_copy_constructor;</td>
<td>has_trivial_copy_constructor&lt;T&gt;::value is true or is_nothrow_constructible&lt;U, const U&amp;&gt;::value is true, where U is remove_all_extents&lt;T&gt;::type.</td>
<td>T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct has_nothrow_move_constructor;</td>
<td>has_trivial_move_constructor&lt;T&gt;::value is true or is_nothrow_constructible&lt;U, U&amp;&amp;&gt;::value is true, where U is remove_all_extents&lt;T&gt;::type.</td>
<td>T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct has_nothrow_copy_assign;</td>
<td>has_trivial_copy_assign&lt;T&gt;::value is true or the expression noexcept(<em>(U</em>)0 = declval&lt;const U&amp;&gt;()) is well-formed and true, where U is remove_all_extents&lt;T&gt;::type.</td>
<td>T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct has_nothrow_move_assign;</td>
<td>has_trivial_move_assign&lt;T&gt;::value is true and T is a trivial type (3.9) or the expression noexcept(<em>(U</em>)0 = declval&lt;U&amp;&gt;()) is well-formed and true, where U is remove_all_extents&lt;T&gt;::type.</td>
<td>T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct has_virtualDestructor;</td>
<td>T has a virtual destructor (12.4)</td>
<td>T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct is_signed;</td>
<td>is_arithmetic&lt;T&gt;::value &amp;&amp; T(-1) &lt; T(0)</td>
<td></td>
</tr>
</tbody>
</table>
Table 45 — Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt; struct is_unsigned;</td>
<td>is_-</td>
<td>arithmetic&lt;T&gt;::value &amp;&amp; T(0) &lt; T(-1)</td>
</tr>
</tbody>
</table>

4  [Example:

is_const<const volatile int>::value // true
is_const<const int*>::value    // false
is_const<const int&>::value    // false
is_const<int[3]>::value        // false
is_const<const int[3]>::value  // true

— end example]

5  [Example:

remove_const<const volatile int>::type // volatile int
remove_const<const int* const>::type   // const int*
remove_const<const int&>::type         // const int&

— end example]

6  Given the following function prototype:

```cpp
template <class T>
typename add_rvalue_reference<T>::type create();
```

the predicate condition for a template specialization `is_constructible<T, Args...>` shall be satisfied if and only if the following expression `CE` would be well-formed:

— if `sizeof...(Args) == 1`, the expression:

```
static_cast<T>(create<Args>()...)
```

— otherwise, the expression:

```
T(create<Args>()...)
```

Table 46 — Type property queries

<table>
<thead>
<tr>
<th>Template</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt; struct alignment_of;</td>
<td><code>alignof(T)</code>.</td>
</tr>
<tr>
<td>template &lt;class T&gt; struct rank;</td>
<td>If T names an array type, an integer value representing the number of dimensions of T; otherwise, 0.</td>
</tr>
<tr>
<td>template &lt;class T, unsigned I = 0&gt; struct extent;</td>
<td>If T is not an array type, or if it has rank less than or equal to I, or if I is 0 and T has type “array of unknown bound of U”, then 0; otherwise, the bound (8.3.4) of the I’th dimension of T, where indexing of I is zero-based</td>
</tr>
</tbody>
</table>

§ 20.7.4.3
Example:
// the following assertions hold:
assert(rank<int>::value == 0);
assert(rank<int[2]>::value == 1);
assert(rank<int[] [4]>::value == 2);
— end example]

Example:
// the following assertions hold:
assert(extent<int>::value == 0);
assert(extent<int[2]>::value == 2);
assert(extent<int[2][4]>::value == 2);
assert((extent<int[], 1>::value) == 0);
assert((extent<int[2], 1>::value) == 0);
assert((extent<int[2][4], 1>::value) == 4);
assert((extent<int[], 1>::value) == 4);
— end example]

20.7.5 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 (20.7.1) with a BaseCharacteristic of true_type if the corresponding condition is true, otherwise false_type.

### Table 47 — Type relationship predicates

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T, class U&gt; struct is_same;</td>
<td>T and U name the same type with the same cv-qualifications</td>
<td></td>
</tr>
<tr>
<td>template &lt;class Base, class Derived&gt; struct is_base_of;</td>
<td>Base is a base class of Derived (10) without regard to cv-qualifiers or Base and Derived are not unions and name the same class type without regard to cv-qualifiers</td>
<td>If Base and Derived are class types and are different types (ignoring possible cv-qualifiers) then Derived shall be a complete type. [Note: Base classes that are private, protected, or ambiguous are, nonetheless, base classes. — end note]</td>
</tr>
<tr>
<td>template &lt;class From, class To&gt; struct is_convertible;</td>
<td>see below</td>
<td>From and To shall be complete types, arrays of unknown bound, or (possibly cv-qualified) void types.</td>
</tr>
<tr>
<td>template &lt;class From, class To&gt; struct is_explicitly_convertible;</td>
<td>is_constructible&lt; To, From&gt;::value</td>
<td>is_explicitly_convertible is a synonym for a two-argument version of is_constructible. An implementation may define it as a template alias.</td>
</tr>
</tbody>
</table>
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<const B, const D>::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 

Given the following function prototype:

```cpp
template <class T>
typename add_rvalue_reference<T>::type create();
```

the predicate condition for a template specialization `is_convertible<From, To>` shall be satisfied if and only if the return expression in the following code would be well-formed, including any implicit conversions to the return type of the function:

```cpp
To test() {
    return create<From>();
}
```

[Note: This requirement gives well defined results for reference types, void types, array types, and function types. — end note]

### 20.7.6 Transformations between types

This sub-clause contains templates that may be used to transform one type to another following some predefined rule.

Each of the templates in this subclause shall be a `TransformationTrait` (20.7.1).

#### 20.7.6.1 Const-volatile modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt; struct remove_const;</td>
<td>The member typedef <code>type</code> shall name the same type as <code>T</code> except that any top-level const-qualifier has been removed.</td>
</tr>
<tr>
<td>example: <code>remove_const&lt;const volatile int&gt;::type</code> evaluates to <code>volatile int</code>, whereas <code>remove_const&lt;const int*&gt;::type</code> evaluates to <code>const int*</code>. — end example</td>
<td></td>
</tr>
</tbody>
</table>

| 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*>::type` evaluates to `volatile int*`. — end example |

§ 20.7.6.1
Table 48 — Const-volatile modifications (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt;</td>
<td>The member typedef type shall be the same as T except that any top-level cv-qualifier has been removed. [Example: remove_cv&lt;const volatile int&gt;::type evaluates to int, whereas remove_cv&lt;const volatile int*&gt;::type evaluates to const volatile int*. — end example]</td>
</tr>
<tr>
<td>struct remove_cv;</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>If T is a reference, function, or top-level const-qualified type, then type shall name the same type as T, otherwise T const.</td>
</tr>
<tr>
<td>struct add_const;</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>If T is a reference, function, or top-level volatile-qualified type, then type shall name the same type as T, otherwise T volatile.</td>
</tr>
<tr>
<td>struct add_volatile;</td>
<td></td>
</tr>
</tbody>
</table>
| template <class T>     | The member typedef type shall name the same type as add_const<
type add_volatile<T>::type>::type. |
| struct add_cv;         |                                                                           |

20.7.6.2 Reference modifications

Table 49 — Reference modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt;</td>
<td>If T has type “reference to T1” then the member typedef type shall name T1; otherwise, type shall name T.</td>
</tr>
<tr>
<td>struct remove_reference;</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>If T names an object or function type then the member typedef type shall name T1&amp;; otherwise, if T names a type “rvalue reference to T1” then the member typedef type shall name T1&amp;&amp;; otherwise, type shall name T.</td>
</tr>
<tr>
<td>struct add_lvalue_reference;</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>If T names an object or function type then the member typedef type shall name T&amp;&amp;; otherwise, type shall name T. [Note: This rule reflects the semantics of reference collapsing. For example, when a type T names a type T1&amp;, the type add_rvalue_reference&lt;T&gt;::type is not an rvalue reference. — end note]</td>
</tr>
<tr>
<td>struct add_rvalue_reference;</td>
<td></td>
</tr>
</tbody>
</table>

20.7.6.3 Sign modifications

Table 50 — Sign modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt;</td>
<td>If T names a (possibly cv-qualified) signed integral type (3.9.1) then the member typedef type shall name the type T; otherwise, if T names a (possibly cv-qualified) unsigned integral type then type shall name the corresponding signed integral type, with the same cv-qualifiers as T; otherwise, type shall name the signed integral type with smallest rank (4.13) for which sizeof(T) == sizeof(type), with the same cv-qualifiers as T. Requires: T shall be a (possibly cv-qualified) integral type or enumeration but not a bool type.</td>
</tr>
<tr>
<td>struct make_signed;</td>
<td></td>
</tr>
</tbody>
</table>
Table 50 — Sign modifications (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt;</td>
<td>If T names a (possibly cv-qualified) unsigned integral type (3.9.1) then the member typedef type shall name the type T; otherwise, if T names a (possibly cv-qualified) signed integral type then type shall name the corresponding unsigned integral type, with the same cv-qualifiers as T; otherwise, type shall name the unsigned integral type with smallest rank (4.13) for which sizeof(T) == sizeof(type), with the same cv-qualifiers as T. Requires: T shall be a (possibly cv-qualified) integral type or enumeration but not a bool type.</td>
</tr>
<tr>
<td>struct make_unsigned;</td>
<td></td>
</tr>
</tbody>
</table>

20.7.6.4 Array modifications

Table 51 — Array modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt;</td>
<td>If T names a type “array of U”, the member typedef type shall be U, otherwise T. [ Note: For multidimensional arrays, only the first array dimension is removed. For a type “array of const U”, the resulting type is const U. — end note ]</td>
</tr>
<tr>
<td>struct remove_extent;</td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>If T is “multi-dimensional array of U”, the resulting member typedef type is U, otherwise T.</td>
</tr>
<tr>
<td>struct remove_all_extents;</td>
<td></td>
</tr>
</tbody>
</table>

1 [Example

// 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));
— end example]

20.7.6.5 Pointer modifications [meta.trans.ptr]
Table 52 — Pointer modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt;</td>
<td>If T has type “(possibly cv-qualified) pointer to T1” then the member</td>
</tr>
<tr>
<td>struct remove_pointer;</td>
<td>typedef type shall name T1; otherwise, it shall name T.</td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td>The member typedef type shall name the same type as</td>
</tr>
<tr>
<td>struct add_pointer;</td>
<td>remove_reference&lt;T&gt;::type*.</td>
</tr>
</tbody>
</table>

20.7.6.6 Other transformations

Table 53 — 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,</td>
<td>Len shall not be zero. Align shall be equal to</td>
<td>The value of default-alignment shall be the most stringent alignment requirement for any C++ object type whose size is no greater than Len (3.9). The member typedef type shall be a POD type suitable for use as uninitialized storage for any object whose size is at most Len and whose alignment is a divisor of Align.</td>
</tr>
<tr>
<td>std::size_t Align =</td>
<td>alignof(T) for some type T or to default-alignment.</td>
<td></td>
</tr>
<tr>
<td>default-alignment&gt; struct</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aligned_storage;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template &lt;class T&gt; struct decay;</td>
<td></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>
</tr>
<tr>
<td>template &lt;bool B, class T =</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>
<tr>
<td>void&gt; struct enable_if;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template &lt;bool B, class T,</td>
<td>If B is true, the member typedef type shall equal T. If B is false, the member typedef type shall equal F.</td>
<td></td>
</tr>
<tr>
<td>class F&gt; struct conditional;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template &lt;class... T&gt; struct</td>
<td></td>
<td>The member typedef type shall be defined as set out below. All types in the parameter pack T shall be complete or (possibly cv) void. A program may specialize this trait if at least one template parameter in the specialization is a user-defined type. [Note: Such specializations are needed when only explicit conversions are desired among the template arguments. — end note]</td>
</tr>
<tr>
<td>common_type;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 53 — Other transformations (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt; struct underlying_type;</td>
<td>T shall be an enumeration type (7.2)</td>
<td>The member typedef type shall name the underlying type of T.</td>
</tr>
<tr>
<td>template &lt;class Fn, class... ArgTypes&gt; struct result_of&lt;Fn(ArgTypes...)&gt;;</td>
<td>Fn shall be a function object type (20.8), reference to function object type. The expression decltype(declval&lt;Fn&gt;()(declval&lt;ArgTypes&gt;()...)) shall be well formed.</td>
<td>The member typedef type shall name the type decltype(declval&lt;Fn&gt;()(declval&lt;ArgTypes&gt;()...))</td>
</tr>
</tbody>
</table>

1 [Note: A typical implementation would define \texttt{aligned} as:

```cpp
template <std::size_t Len, std::size_t Alignment>
struct aligned_storage {
    typedef struct {
        unsigned char __data [[ align(Alignment) ]] [Len];
    } type;
};
```

— end note]

2 It is implementation-defined whether any extended alignment is supported (3.11).

3 The nested typedef \texttt{common}::type shall be defined as follows:

```cpp
template <class ...T> struct common_type;
```

```cpp
template <class T> struct common_type<T> {
    typedef T type;
};
```

```cpp
template <class T, class U> struct common_type<T, U> {
    typedef decltype(true ? declval<T>() : declval<U>()) type;
};
```

```cpp
template <class T, class U, class... V> struct common_type<T, U, V...> {
    typedef typename common_type<typename common_type<T, U>::type, V...>::type type;
};
```

4 [Example: Given these definitions:

```cpp
typedef bool (&PF1)();
typedef short (*PF2)(long);

struct S {
    operator PF2() const;
    double operator()(char, int&);
};
```

§ 20.7.6.6
the following assertions will hold:

```cpp
static_assert(std::is_same<std::result_of<S(int)>::type, short>::value, "Error!");
static_assert(std::is_same<std::result_of<S&(unsigned char, int&)>::type, double>::value, "Error!");
static_assert(std::is_same<std::result_of<PF1()>::type, bool>::value, "Error!");
```

— end example]

### 20.8 Function objects

A *function object type* is an object type (3.9) that can be the type of the *postfix-expression* in a function call (5.2.2, 13.3.1.1). A *function object* is an object of a function object type. In the places where one would expect to pass a pointer to a function to an algorithmic template (Clause 25), the interface is specified to accept a function object. This not only makes algorithmic templates work with pointers to functions, but also enables them to work with arbitrary function objects.

#### 2 Header `<functional>` synopsis

```cpp
namespace std {
  // 20.8.3, base:
  template <class Arg, class Result> struct unary_function;
  template <class Arg1, class Arg2, class Result> struct binary_function;

  // 20.8.4, 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> void ref(const T&&) = delete;
  template <class T> void cref(const T&&) = delete;

  template <class T> reference_wrapper<T> ref(reference_wrapper<T>);
  template <class T> reference_wrapper<const T> cref(reference_wrapper<T>);

  // 20.8.5, 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;

  // 20.8.6, 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;

  // 20.8.7, logical operations:
  template <class T> struct logical_and;
  template <class T> struct logical_or;
  template <class T> struct logical_not;
```

234 Such a type is a function pointer or a class type which has a member `operator()` or a class type which has a conversion to a pointer to function.

§ 20.8 529
// 20.8.8, bitwise operations:
template <class T> struct bit_and;
template <class T> struct bit_or;
template <class T> struct bit_xor;

// 20.8.9, 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&);

// 20.8.10, bind:
template<class T> struct is_bind_expression;
template<class T> struct is_placeholder;

template<class F, class... BoundArgs>
  unspecified bind(F&, BoundArgs&&...);
template<class R, class F, class... BoundArgs>
  unspecified bind(F&&, BoundArgs&&...);

namespace placeholders {
  // M is the implementation-defined number of placeholders
  extern unspecified _1;
  extern unspecified _2;

  extern unspecified _M;
}

// D.9, 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&);

// 20.8.11, adaptors:
template <class Arg, class Result> class pointer_to_unary_function;
template <class Arg, class Result>
  pointer_to_unary_function<Arg,Result> ptr_fun(Result (*)(Arg));
template <class Arg1, class Arg2, class Result>
  class pointer_to_binary_function;
template <class Arg1, class Arg2, class Result>
  pointer_to_binary_function<Arg1,Arg2,Result>
    ptr_fun(Result (*)(Arg1,Arg2));

// 20.8.12, adaptors:
template<class S, class T> class mem_fun_t;
template<class S, class T, class A> class mem_fun1_t;
template<class S, class T>
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 A>
mem_fun1_ref_t<S,T,A> mem_fun_ref(S (T::*f)(A));

template<class S, class T, class A>
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 A>
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 A>
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);

// 20.8.13, member function adaptors:
template<class R, class T> unspecified mem_fn(R T::*);
template<class R, class T, class ... Arg>
unspecified mem_fn(R T::*)(Args...);
template<class R, class T, class ... Arg>
unspecified mem_fn(R T::*)(Args...) const;
template<class R, class T, class ... Arg>
unspecified mem_fn(R T::*)(Args...) volatile;
template<class R, class T, class ... Arg>
unspecified mem_fn(R T::*)(Args...) const volatile;
template<class R, class T, class ... Arg>
unspecified mem_fn(R T::*)(Args...) &;
template<class R, class T, class ... Arg>
unspecified mem_fn(R T::*)(Args...) const &;
template<class R, class T, class ... Arg>
unspecified mem_fn(R T::*)(Args...) volatile &;
template<class R, class T, class ... Arg>
unspecified mem_fn(R T::*)(Args...) const volatile &;
template<class R, class T, class ... Arg>
unspecified mem_fn(R T::*)(Args...) & &;
template<class R, class T, class ... Arg>
unspecified mem_fn(R T::*)(Args...) const & &;
template<class R, class T, class ... Arg>
unspecified mem_fn(R T::*)(Args...) volatile & &;
template<class R, class T, class ... Arg>
unspecified mem_fn(R T::*)(Args...) const volatile & &;

// 20.8.14 polymorphic function wrappers:
class bad_function_call;

template<class R, class ... ArgTypes> class function<R(ArgTypes...)>;

template<class R, class ... ArgTypes>
void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&);

template<class R, class ... ArgTypes>
bool operator==(const function<R(ArgTypes...)>&, nullptr_t);
template<class R, class ... ArgTypes>
bool operator==(nullptr_t, const function<R(ArgTypes...)>&);
template<class R, class ... ArgTypes>
bool operator!=(const function<R(ArgTypes...)>&, nullptr_t);
template<class R, class ... ArgTypes>
bool operator!=(nullptr_t, const function<R(ArgTypes...)>&);
// 20.8.15, hash function base template:
template <class T> struct hash;

// Hash function specializations
template <> struct hash<bool>;
template <> struct hash<char>;
template <> struct hash<signed char>;
template <> struct hash<unsigned char>;
template <> struct hash<char16_t>;
template <> struct hash<char32_t>;
template <> struct hash<wchar_t>;
template <> struct hash<short>;
template <> struct hash<unsigned short>;
template <> struct hash<int>;
template <> struct hash<unsigned int>;
template <> struct hash<long>;
template <> struct hash<long long>;
template <> struct hash<unsigned long>;
template <> struct hash<unsigned long long>;

template <> struct hash<float>;
template <> struct hash<double>;
template <> struct hash<long double>;

template<class T> struct hash<T*>;
}

3 [Example: If a C++ program wants to have a by-element addition of two vectors a and b containing double and put the result into a, it can do:

transform(a.begin(), a.end(), b.begin(), a.begin(), plus<double>());

— end example]

4 [Example: To negate every element of a:

transform(a.begin(), a.end(), a.begin(), negate<double>());

— end example]

5 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.

20.8.1 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 callable type is a function object type (20.8) or a pointer to member.

4 A callable object is an object of a callable type.
A call wrapper type is a type that holds a callable object and supports a call operation that forwards to that object.

A call wrapper is an object of a call wrapper type.

A target object is the callable object held by a call wrapper.

### 20.8.2 Requirements

1. Define \( \text{INVOKE}(f, t_1, t_2, \ldots, t_N) \) as follows:
   - \((t_1.*f)(t_2, \ldots, t_N)\) when \( f \) is a pointer to a member function of a class \( T \) and \( t_1 \) is an object of type \( T \) or a reference to an object of type \( T \) or a reference to an object of a type derived from \( T \);
   - \(((*t_1).*f)(t_2, \ldots, t_N)\) when \( f \) is a pointer to a member function of a class \( T \) and \( t_1 \) is not one of the types described in the previous item;
   - \( t_1.*f \) when \( f \) is a pointer to member data of a class \( T \) and \( t_1 \) is an object of type \( T \) or a reference to an object of type \( T \) or a reference to an object of a type derived from \( T \);
   - \((*t_1).*f \) when \( f \) is a pointer to member data of a class \( T \) and \( t_1 \) is not one of the types described in the previous item;
   - \( f(t_1, t_2, \ldots, t_N) \) in all other cases.

2. Define \( \text{INVOKE}(f, t_1, t_2, \ldots, t_N, R) \) as \( \text{INVOKE}(f, t_1, t_2, \ldots, t_N) \) implicitly converted to \( R \).

3. If a call wrapper (20.8.1) has a weak result type the type of its member type \( \text{result_type} \) is based on the type \( T \) of the wrapper’s target object (20.8.1):
   - if \( T \) is a function, reference to function, or pointer to function type, \( \text{result_type} \) shall be a synonym for the return type of \( T \);
   - if \( T \) is a pointer to member function, \( \text{result_type} \) shall be a synonym for the return type of \( T \);
   - if \( T \) is a class type with a member type \( \text{result_type} \), then \( \text{result_type} \) shall be a synonym for \( T::\text{result_type} \);
   - otherwise \( \text{result_type} \) shall not be defined.

4. Every call wrapper (20.8.1) shall be MoveConstructible. A simple call wrapper is a call wrapper that is CopyConstructible and CopyAssignable and whose copy constructor, move constructor, and assignment operator do not throw exceptions. A forwarding call wrapper is a call wrapper that can be called with an argument list. [ Note: in a typical implementation forwarding call wrappers have an overloaded function call operator of the form
   \[
   \text{template<class... UnBoundArgs>}
   \quad \text{R operator()}(\text{UnBoundArgs}\&\&\ldots \text{unbound_args}) \text{ cv-qual};
   \]
   — end note ]

### 20.8.3 Base

1. The following classes are provided to simplify the typedefs of the argument and result types:

   ```cpp
   namespace std {
      template <class Arg, class Result>
      struct unary_function {
         typedef Arg argument_type;
         typedef Result result_type;
   }
   ```

§ 20.8.3
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;
    };
}

20.8.4 Class template reference_wrapper

namespace 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 std::result_type;           // not always defined

        // construct/copy/destroy
        reference_wrapper(T&);
        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 reference_wrapper<T> is a CopyConstructible and CopyAssignable wrapper around a reference to an object or function of type T.

2 reference_wrapper has a weak result type (20.8.2).

3 The template instantiation reference_wrapper<T> shall be derived from std::unary_function<T1, R> only if the type T is any of the following:
   — a function type or a pointer to function type taking one argument of type T1 and returning R
   — a pointer to member function R T0::f cv (where cv represents the member function's cv-qualifiers);
     the type T1 is cv T0*
   — a class type that is derived from std::unary_function<T1, R>
The template instantiation \texttt{reference\_wrapper<T>} shall be derived from \texttt{std::binary\_function<T1, T2, R>} only if the type \texttt{T} is any of the following:

- a function type or a pointer to function type taking two arguments of types \texttt{T1} and \texttt{T2} and returning \texttt{R}
- a pointer to member function \texttt{R T0::f(T2) cv} (where \texttt{cv} represents the member function’s cv-qualifiers);
  - the type \texttt{T1} is \texttt{cv T0*}
- a class type that is derived from \texttt{std::binary\_function<T1, T2, R>}

20.8.4.1 \texttt{reference\_wrapper} construct/copy/destroy \quad \texttt{[refwrap.const]}

\texttt{reference\_wrapper(T& t);}  
\texttt{1 Effects: Constructs a reference\_wrapper object that stores a reference to t.}
\texttt{2 Throws: nothing.}

\texttt{reference\_wrapper(const reference\_wrapper<T>& x);}  
\texttt{3 Effects: Constructs a reference\_wrapper object that stores a reference to x.get().}
\texttt{4 Throws: nothing.}

20.8.4.2 \texttt{reference\_wrapper} assignment \quad \texttt{[refwrap.assign]}

\texttt{reference\_wrapper& operator=(const reference\_wrapper<T>& x);}  
\texttt{1 Postconditions: *this stores a reference to x.get().}
\texttt{2 Throws: nothing.}

20.8.4.3 \texttt{reference\_wrapper} access \quad \texttt{[refwrap.access]}

\texttt{operator T& () const;}  
\texttt{1 Returns: The stored reference.}
\texttt{2 Throws: nothing.}

\texttt{T& get() const;}  
\texttt{3 Returns: The stored reference.}
\texttt{4 Throws: nothing.}

20.8.4.4 \texttt{reference\_wrapper} invocation \quad \texttt{[refwrap.invoke]}

\texttt{template <class... ArgTypes>}
\texttt{typename result\_of<T(ArgTypes... )>::type}
\texttt{operator()(ArgTypes&&... args) const;}  
\texttt{1 Returns: INVOLVE(get(), std::forward<ArgTypes>(args)...). (20.8.2)}
\texttt{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.}

§ 20.8.4.4
20.8.4.5 reference_wrapper helper functions

```cpp
template <class T> reference_wrapper<T> ref(T& t);

Returns: reference_wrapper<T>(t)
Throws: nothing.
```

```cpp
template <class T> reference_wrapper<T> ref(reference_wrapper<T>t);

Returns: ref(t.get())
Throws: nothing.
```

```cpp
template <class T> reference_wrapper<const T> cref(const T& t);

Returns: reference_wrapper<const T>(t)
Throws: nothing.
```

```cpp
template <class T> reference_wrapper<const T> cref(reference_wrapper<T> t);

Returns: cref(t.get())
Throws: nothing.
```

20.8.5 Arithmetic operations

The library provides basic function object classes for all of the arithmetic operators in the language (5.6, 5.7).

```cpp
template <class T> struct plus : binary_function<T,T,T> {
    T operator()(const T& x, const T& y) const;
};

operator() returns x + y.
```

```cpp
template <class T> struct minus : binary_function<T,T,T> {
    T operator()(const T& x, const T& y) const;
};

operator() returns x - y.
```

```cpp
template <class T> struct multiplies : binary_function<T,T,T> {
    T operator()(const T& x, const T& y) const;
};

operator() returns x * y.
```

```cpp
template <class T> struct divides : binary_function<T,T,T> {
    T operator()(const T& x, const T& y) const;
};

operator() returns x / y.
```

```cpp
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 \(-x\).

20.8.6 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> {
    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 \(x \geq y\).

template <class T> struct less_equal : binary_function<T,T,bool> {
    bool operator()(const T& x, const T& y) const;
};
operator() returns \(x \leq y\).

For templates \texttt{greater}, \texttt{less}, \texttt{greater\_equal}, and \texttt{less\_equal}, the specializations for any pointer type yield a total order, even if the built-in operators \(<\), \(>, \leq, \geq\) do not.

20.8.7 Logical operations

The library provides basic function object classes for all of the logical operators in the language (5.14, 5.15, 5.3.1).

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.

20.8.8 Bitwise operations

The library provides basic function object classes for all of the bitwise operators in the language (5.11, 5.13, 5.12).

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 | y.

template <class T> struct bit_xor : binary_function<T,T,T> {
    T operator()(const T& x, const T& y) const;
};
operator() returns x ^ y.

20.8.9 Negators

Negators not1 and not2 take a unary and a binary predicate, respectively, and return their complements (5.3.1).

template <class Predicate>
class unary_negate
    : public 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).

template <class Predicate>
unary_negate<Predicate> not1(const Predicate& pred);

Returns: unary_negate<Predicate>(pred).
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).

template <class Predicate>
binary_negate<Predicate> not2(const Predicate& pred);

Returns:

binary_negate<Predicate>(pred).

20.8.10 Function template bind

The function template bind returns an object that binds a callable object passed as an argument to additional arguments.

20.8.10.1 Function object binders

This subclause describes a uniform mechanism for binding arguments of callable objects.

20.8.10.1.1 Class template is_bind_expression

namespace std {

  template<class T> struct is_bind_expression :
      integral_constant<bool, 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.

2 If T is a type returned from bind, is_bind_expression<T> shall be publicly derived from integral_constant<bool, true>, otherwise from integral_constant<bool, false>.

3 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.

4 If T is the type of std::placeholders::_J, is_placeholder<T> shall be publicly derived from integral_constant<int, J>, otherwise from integral_constant<int, 0>.

20.8.10.1.2 Function template bind

In the text that follows, the following names have the following meanings:

— FD is the type decay<F>::type,
— fd is an lvalue of type FD constructed from std::forward<F>(f),
— Ti is the i\textsuperscript{th} type in the template parameter back BoundArgs,
— TiD is the type decay<Ti>::type,
— ti is the $i^{th}$ argument in the function parameter pack `bound_args`,
— tid is an lvalue of type TiD constructed from `std::forward<Ti>(ti)`,
— Uj is the $j^{th}$ deduced type of the `UnBoundArgs&&...` parameter of the forwarding call wrapper, and
— uj is the $j^{th}$ argument associated with Uj.

```cpp
template<class F, class... BoundArgs>
unspecified bind(F&& f, BoundArgs&&... bound_args);
```

2 Requires: `is_constructible<FD, F>::value` shall be `true`. For each Ti in `BoundArgs`, `is_constructible<TiD, Ti>::value` shall be `true`. `INVOKE(fd, w1, w2, ..., wN)` (20.8.2) shall be a valid expression for some values $w1, w2, ..., wN$, where $N == \text{sizeof...}(\text{bound_args})$.

3 Returns: A forwarding call wrapper g with a weak result type (20.8.2). The effect of $g(u1, u2, ..., uM)$ shall be `INVOKE(fd, v1, v2, ..., vN, \text{result_of<FD cv (V1, V2, ..., VN)::type})`, where `cv` represents the `cv`-qualifiers of g and the values and types of the bound arguments $v1, v2, ..., vN$ are determined as specified below. The copy constructor and move constructor of the forwarding call wrapper shall throw an exception if and only if the corresponding constructor of FD or of any of the types TiD throws an exception.

4 Throws: Nothing unless the construction of fd or of one of the values tid throws an exception.

5 Remarks: The return type shall satisfy the requirements of `MoveConstructible`. If all of FD and TiD satisfy the requirements of `CopyConstructible`, then the return type shall satisfy the requirements of `CopyConstructible`. [Note: this implies that all of FD and TiD are `MoveConstructible`. — end note]

```cpp
template<class R, class F, class... BoundArgs>
unspecified bind(F&& f, BoundArgs&&... bound_args);
```

6 Requires: `is_constructible<FD, F>::value` shall be `true`. For each Ti in `BoundArgs`, `is_constructible<TiD, Ti>::value` shall be `true`. `INVOKE(fd, w1, w2, ..., wN)` shall be a valid expression for some values $w1, w2, ..., wN$, where $N == \text{sizeof...}(\text{bound_args})$.

7 Returns: A forwarding call wrapper g with a nested type `result_type` defined as a synonym for R. The effect of $g(u1, u2, ..., uM)$ shall be `INVOKE(fd, v1, v2, ..., vN, R)`, where the values and types of the bound arguments $v1, v2, ..., vN$ are determined as specified below. The copy constructor and move constructor of the forwarding call wrapper shall throw an exception if and only if the corresponding constructor of FD or of any of the types TiD throws an exception.

8 Throws: Nothing unless the construction of fd or of one of the values tid throws an exception.

9 Remarks: The return type shall satisfy the requirements of `MoveConstructible`. If all of FD and TiD satisfy the requirements of `CopyConstructible`, then the return type shall satisfy the requirements of `CopyConstructible`. [Note: this implies that all of FD and TiD are `MoveConstructible`. — end note]

10 The values of the `bound arguments` $v1, v2, ..., vN$ and their corresponding types $V1, V2, ..., VN$ depend on the types TiD derived from the call to bind and the `cv`-qualifiers `cv` of the call wrapper g as follows:

— if TiD is `reference_wrapper<T>`, the argument is `tid.get()` and its type $V1$ is `T&`;
— if the value of `is_bind_expression<TiD>::value` is `true`, the argument is `tid(std::forward<Uj>(uj)...)` and its type $V1$ is `result_of<TiD cv (Uj...)>::type`;

§ 20.8.10.1.2
— if the value \( j \) of `is_placeholder<TiD>::value` is not zero, the argument is `std::forward<Uj>(uj)` and its type \( Vi \) is `Uj&&`;

— otherwise, the value is \( tid \) and its type \( Vi \) is `TiD cv &`.

### 20.8.10.1.3 Placeholders

```cpp
namespace std {
    namespace placeholders {
        // M is the implementation-defined number of placeholders
        extern unspecified _1;
        extern unspecified _2;
        ...
        extern unspecified _M;
    }
}
```

All placeholder types shall be `DefaultConstructible` and `CopyConstructible`, and their default constructors and copy/move constructors shall not throw exceptions. It is implementation-defined whether placeholder types are `CopyAssignable`. `CopyAssignable` placeholders' copy assignment operators shall not throw exceptions.

### 20.8.11 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 \( f(x) \).

    template <class Arg, class Result>
    pointer_to_unary_function<Arg, Result> ptr_fun(Result (*f)(Arg));

    Returns: `pointer_to_unary_function<Arg, Result>(f)`.
}
```

```cpp
template <class Arg1, class Arg2, class Result>
class pointer_to_binary_function : public binary_function<Arg1,Arg2,Result> {
    public:
        explicit pointer_to_binary_function(Result (*f)(Arg1, Arg2));
        Result operator()(Arg1 x, Arg2 y) const;
    }

    operator() returns \( f(x,y) \).

    template <class Arg1, class Arg2, class Result>
    pointer_to_binary_function<Arg1,Arg2,Result> ptr_fun(Result (*f)(Arg1, Arg2));

    Returns: `pointer_to_binary_function<Arg1,Arg2,Result>(f)`.
```
Example:

```c
int compare(const char*, const char*);
replace_if(v.begin(), v.end(),
not1(bind2nd(ptr_fun(compare), "abc")), "def");
```

replaces each `abc` with `def` in sequence `v`. — end example]

### 20.8.12 Adaptors for pointers to members

The purpose of the following is to provide the same facilities for pointer to members as those provided for pointers to functions in 20.8.11.

```cpp
template <class S, class T> class mem_fun_t
  : public unary_function<T*, S> {
public:
  explicit mem_fun_t(S (T::*p)());
  S operator()(T* p) const;
};
mem_fun_t calls the member function it is initialized with given a pointer argument.
```

```cpp
template <class S, class T, class A> class mem_fun1_t
  : public binary_function<T*, A, S> {
public:
  explicit mem_fun1_t(S (T::*p)(A));
  S operator()(T* p, A x) const;
};
mem_fun1_t calls the member function it is initialized with given a pointer argument and an additional argument of the appropriate type.
```

```cpp
template<class S, class T> mem_fun_t<S,T> mem_fun(S (T::*f)());
template<class S, class T, class A> mem_fun1_t<S,T,A> mem_fun(S (T::*f)(A));
mem_fun(&X::f) returns an object through which X::f can be called given a pointer to an X followed by the argument required for f (if any).
```

```cpp
template <class S, class T> class mem_fun_ref_t
  : public unary_function<T, S> {
public:
  explicit mem_fun_ref_t(S (T::*p)());
  S operator()(T& p) const;
};
mem_fun_ref_t calls the member function it is initialized with given a reference argument.
```

```cpp
template <class S, class T, class A> class mem_fun1_ref_t
  : public binary_function<T, A, S> {
public:
  explicit mem_fun1_ref_t(S (T::*p)(A));
  S operator()(T& p, A x) const;
};
mem_fun1_ref_t calls the member function it is initialized with given a reference argument and an additional argument of the appropriate type.
```
template<class S, class T> mem_fun_t<S,T>
    mem_fun_t(S (T::*f)());

template<class S, class T, class A> mem_fun1_t<S,T,A>
    mem_fun1_t(S (T::*f)(A));

mem_fun_t(&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).

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.
20.8.13 Function template mem_fn

\texttt{template<typename R, class T>}
\texttt{\textnormal{\emph{mem_fn}}(R T::* pm);}  
\texttt{\textnormal{\emph{mem_fn}}(R T::* pm)(Args...));}  
\texttt{\textnormal{\emph{mem_fn}}(R T::* pm)(Args... const);}  
\texttt{\textnormal{\emph{mem_fn}}(R T::* pm)(Args... volatile);}  
\texttt{\textnormal{\emph{mem_fn}}(R T::* pm)(Args... const volatile);}  
\texttt{\textnormal{\emph{mem_fn}}(R T::* pm)(Args... &);}  
\texttt{\textnormal{\emph{mem_fn}}(R T::* pm)(Args... const &);}  
\texttt{\textnormal{\emph{mem_fn}}(R T::* pm)(Args... volatile &);}  
\texttt{\textnormal{\emph{mem_fn}}(R T::* pm)(Args... const volatile &);}  
\texttt{\textnormal{\emph{mem_fn}}(R T::* pm)(Args... &&);}  
\texttt{\textnormal{\emph{mem_fn}}(R T::* pm)(Args... const &&);}  
\texttt{\textnormal{\emph{mem_fn}}(R T::* pm)(Args... volatile &&);}  
\texttt{\textnormal{\emph{mem_fn}}(R T::* pm)(Args... const volatile &&);}  

\textbf{Returns:} A simple call wrapper \texttt{(20.8.1)} \texttt{fn} such that the expression \texttt{fn(t, a2, \ldots, aN)} is equivalent to \texttt{\textnormal{\emph{invoke}}(pm, t, a2, \ldots, aN)} \texttt{(20.8.2)}. \texttt{fn} shall have a nested type \texttt{\emph{result_type}} that is a synonym for the return type of \texttt{pm} when \texttt{pm} is a pointer to member function.

\textbf{The simple call wrapper shall be derived from} \texttt{std::\emph{unary_function}}<\texttt{cv T*, Ret}> when \texttt{pm} is a pointer to member function with \texttt{cv}-qualifier \texttt{cv} and taking no arguments, where \texttt{Ret} is \texttt{pm}'s return type.

\textbf{The simple call wrapper shall be derived from} \texttt{std::\emph{binary_function}}<\texttt{cv T*, T1, Ret}> when \texttt{pm} is a pointer to member function with \texttt{cv}-qualifier \texttt{cv} and taking one argument of type \texttt{T1}, where \texttt{Ret} is \texttt{pm}'s return type.

\textbf{Throws:} Nothing.

20.8.14 Polymorphic function wrappers

This subclause describes a polymorphic wrapper class that encapsulates arbitrary callable objects.

20.8.14.1 Class bad_function_call

An exception of type \texttt{bad_function_call} is thrown by \texttt{function::\textnormal{\emph{operator}}()} \texttt{(20.8.14.2.4)} when the function wrapper object has no target.
namespace std {
    class bad_function_call : public std::exception {
        public:
            // 20.8.14.1.1, constructor:
            bad_function_call();
    } // namespace std

20.8.14.1.1 bad_function_call constructor  [func.wrap.badcall.const]
bad_function_call();

Effects: constructs a bad_function_call object.

20.8.14.2 Class template function  [func.wrap.func]

namespace std {
    template<class> class function; // undefined

template<class R, class... ArgTypes>
    class function<R(ArgTypes...)> : public unary_function<T1, R>
        // iff sizeof...(ArgTypes) == 1 and ArgTypes contains T1
        : public binary_function<T1, T2, R>
        // iff sizeof...(ArgTypes) == 2 and ArgTypes contains T1 and T2
    {
        public:
            typedef R result_type;

            // 20.8.14.2.1, construct/copy/destroy:
            explicit function();
            function(nullptr_t);
            function(const function&);
            function(function&&);
            template<class F> function(F);
            template<class A> function(allocator_arg_t, const A&);
            template<class A> function(allocator_arg_t, const A&, nullptr_t);
            template<class A> function(allocator_arg_t, const A&,
                const function&);
            template<class A> function(allocator_arg_t, const A&,
                function&&);
            template<class F, class A> function(allocator_arg_t, const A&,
                const F&);
            function& operator=(const function&);
            function& operator=(function&&);
            function& operator=(nullptr_t);
            template<class F> function& operator=(F&&);
            template<class F> function& operator=(reference_wrapper<F>);

            ~function();

            // 20.8.14.2.2, function modifiers:
            void swap(function&);
            template<class F, class A> void assign(F&&, const A&);

§ 20.8.14.2
// 20.8.14.2.3, 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;

// 20.8.14.2.4, function invocation:
R operator()(ArgTypes...) const;

// 20.8.14.2.5, function target access:
const std::type_info& target_type() const;
template <typename T> T* target();
template <typename T> const T* target() const;
};

// 20.8.14.2.6, Null pointer comparisons:
template <class R, class... ArgTypes>
    bool operator==(const function<R(ArgTypes...)>&, nullptr_t);

    template <class R, class... ArgTypes>
        bool operator==(nullptr_t, const function<R(ArgTypes...)>&);

    template <class R, class... ArgTypes>
        bool operator!=(const function<R(ArgTypes...)>&, nullptr_t);

    template <class R, class... ArgTypes>
        bool operator!=(nullptr_t, const function<R(ArgTypes...)>&);

// 20.8.14.2.7, specialized algorithms:
template <class R, class... ArgTypes>
    void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&);

    template<class R, class... ArgTypes, class Alloc>
    struct uses_allocator<function<R(ArgTypes...)>, Alloc>
        : true_type { };
};

1 The function class template provides polymorphic wrappers that generalize the notion of a function pointer. Wrappers can store, copy, and call arbitrary callable objects (20.8.1), given a call signature (20.8.1), allowing functions to be first-class objects.

2 A callable object \( f \) of type \( F \) is Callable for argument types \( \text{ArgTypes} \) and return type \( R \) if the expression \( \text{INVOLVE}(f, \text{declval}<\text{ArgTypes}>(), ..., R) \), considered as an unevaluated operand (Clause 5), is well formed (20.8.2).

3 The function class template is a call wrapper (20.8.1) whose call signature (20.8.1) is \( R(<\text{ArgTypes}>) \).

20.8.14.2.1 function construct/copy/destroy

1 When any function constructor that takes a first argument of type allocator_arg_t is invoked, the second argument shall have a type that conforms to the requirements for Allocator (Table 20.2.5). A copy of the
allocator argument is used to allocate memory, if necessary, for the internal data structures of the constructed
function object.

explicit function();
template <class A> function(allocation_arg_t, const A& a);

Postconditions: !*this.

Throws: nothing.

function(nullptr_t);
template <class A> function(allocation_arg_t, const A& a, nullptr_t);

Postconditions: !*this.

Throws: nothing.

function(const function& f);
template <class A> function(allocation_arg_t, const A& a, 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 callable object passed via reference_wrapper or
a function pointer. Otherwise, may throw bad_alloc or any exception thrown by the copy constructor
of the stored callable object. [ Note: Implementations are encouraged to avoid the use of dynamically
allocated memory for small callable objects, for example, where f’s target is an object holding only a
pointer or reference to an object and a member function pointer. — end note ]

function(function&& f);
template <class A> function(allocation_arg_t, const A& a, 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(allocation_arg_t, const A& a, F f);

Requires: F shall be CopyConstructible. f shall be Callable (20.8.14.2) for argument types ArgTypes
and return type R. 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. [ Note: implementations are
encouraged to avoid the use of dynamically allocated memory for small callable objects, for example,
where f’s target is an object holding only a pointer or reference to an object and a member function
pointer. — end note ]

Throws: shall not throw exceptions when f is a function pointer or a reference_wrapper<T> for some
T. Otherwise, may throw bad_alloc or any exception thrown by F’s copy or move constructor.

function& operator=(const function& f);

Effects: function(f).swap(*this);
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=(nullptr_t);

*Effects: If *this != NULL, destroys the target of this.*

Returns: *this

void swap(function& other);

*Effects: interchanges the targets of *this and other.*

Throws: nothing.

template<class F> function& operator=(F&& f);

*Effects: function(std::forward<F>(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.*

```

20.8.14.2.2  function modifiers  [func.wrap.func.mod]

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(allocator_arg, a, std::forward<F>(f)).swap(*this)

20.8.14.2.3  function capacity  [func.wrap.func.cap]

explicit operator bool() const

*Returns: true if *this has a target, otherwise false.*

Throws: nothing.

20.8.14.2.4  function invocation  [func.wrap.func.inv]

R operator()(ArgTypes... args) const

§ 20.8.14.2.4

548
Effects: \( \text{INVOKE}(f, t_1, t_2, \ldots, t_N, R) \) (20.8.2), where \( f \) is the target object (20.8.1) of \(*\text{this} \) and \( t_1, t_2, \ldots, t_N \) are the values in \text{args}....

Returns: Nothing if \( R \) is \text{void}, otherwise the return value of \( \text{INVOKE}(f, t_1, t_2, \ldots, t_N, R) \).

Throws: \text{bad_function_call} if \(!*\text{this} \); otherwise, any exception thrown by the wrapped callable object.

### 20.8.14.2.5 function target access

```
const std::type_info& target_type() const;
```

Returns: If \(*\text{this} \) has a target of type \( T \), \text{typeid}(T); otherwise, \text{typeid(void)}.

Throws: nothing.

```
template<typename T> T* target();
template<typename T> const T* target() const;
```

Requires: \( T \) shall be a type that is Callable (20.8.14.2) for parameter types \text{ArgTypes} \) and return type \( R \).

Returns: If \( \text{target_type()} == \text{typeid}(T) \) a pointer to the stored function target; otherwise a null pointer.

Throws: nothing.

### 20.8.14.2.6 null pointer comparison operators

```
template <class R, class... ArgTypes>
bool operator==(const function<R(ArgTypes...)>& f, nullptr_t);
```

Returns: \!f.

Throws: nothing.

```
template <class R, class... ArgTypes>
bool operator!=(const function<R(ArgTypes...)>& f, nullptr_t);
```

Returns: (bool) \!f.

Throws: nothing.

### 20.8.14.2.7 specialized algorithms

```
template<class R, class... ArgTypes>
void swap(function<R(ArgTypes...)>& f1, function<R(ArgTypes...)>& f2);
```

Effects: \( f_1.\text{swap}(f_2) \);
20.8.15 Class template `hash` [unord.hash]

1 The unordered associative containers defined in Clause 23.5 use specializations of the class template `hash` as the default hash function. For all object types `Key` for which there exists a specialization `hash<Key>`, the instantiation `hash<Key>` shall:

   — satisfy the `Hash` requirements (20.2.4), with `Key` as the function call argument type, the `DefaultConstructible` requirements (33), the `CopyAssignable` requirements (37),
   — be swappable (20.2.2) for lvalues,
   — provide two nested types `result_type` and `argument_type` which shall be synonyms for `size_t` and `Key`, respectively,
   — satisfy the requirement that if `k1 == k2` is true, `h(k1) == h(k2)` is also true, where `h` is an object of type `hash<Key>` and `k1` and `k2` are objects of type `Key`.

```cpp
template <> struct hash<bool>;
template <> struct hash<char>;
template <> struct hash<signed char>;
template <> struct hash<unsigned char>;
template <> struct hash<char16_t>;
template <> struct hash<char32_t>;
template <> struct hash<wchar_t>;
template <> struct hash<short>;
template <> struct hash<unsigned short>;
template <> struct hash<int>;
template <> struct hash<unsigned int>;
template <> struct hash<long>;
template <> struct hash<unsigned long>;
template <> struct hash<long long>;
template <> struct hash<unsigned long long>;
template <> struct hash<float>;
template <> struct hash<double>;
template <> struct hash<long double>;
template <> struct hash<T*>;
```

2 Requires: the template specializations shall meet the requirements of class template `hash` (20.8.15).

20.9 Memory [memory]

1 Header `<memory>` synopsis

   namespace std {
   // 20.9.1, allocator argument tag
   struct allocator_arg_t { };
   constexpr allocator_arg_t allocator_arg = allocator_arg_t();

   // 20.9.2, uses_allocator
   template <class T, class Alloc> struct uses_allocator;

   // 20.9.3, pointer traits
   template <class Ptr> struct pointer_traits;
   template <class T> struct pointer_traits<T*>

   // 20.9.4, allocator traits
   template <class Alloc> struct allocator_traits;
// 20.9.5, the default allocator:
template <class T> class allocator;
template <> class allocator<void>;
template <class T, class U>
  bool operator==(const allocator<T>&, const allocator<U>&) throw();
template <class T, class U>
  bool operator!=(const allocator<T>&, const allocator<U>&) throw();

// 20.9.6, scoped allocator adaptor
template <class OuterAlloc, class... InnerAlloc>
class scoped_allocator_adaptor;
template <class OuterA1, class OuterA2, class... InnerAllocs>
  bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a, const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b);
template <class OuterA1, class OuterA2, class... InnerAllocs>
  bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a, const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b);

// 20.9.7, raw storage iterator:
template <class OutputIterator, class T> class raw_storage_iterator;

// 20.9.8, temporary buffers:
template <class T>
  pair<T*,ptrdiff_t> get_temporary_buffer(ptrdiff_t n);
template <class T>
  void return_temporary_buffer(T* p);

// 20.9.9, specialized algorithms:
template <class T> T* addressof(T& r);
template <class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_copy(InputIterator first, InputIterator last, ForwardIterator result);
template <class InputIterator, class Size, class ForwardIterator>
  ForwardIterator uninitialized_copy_n(InputIterator first, Size n, ForwardIterator result);
template <class ForwardIterator, class T>
  void uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x);
template <class ForwardIterator, class Size, class T>
  void uninitialized_fill_n(ForwardIterator first, Size n, const T& x);

// 20.9.10 Class unique_ptr:
template <class T> class default_delete;
template <class T> class default_delete<T[]>;
template <class T, class D = default_delete<T>> class unique_ptr;
template <class T, class D> class unique_ptr<T[], D>;

template <class 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);

// 20.9.11.5, shared_ptr atomic access:
template<class T>
  bool atomic_is_lock_free(const shared_ptr<T> * p);

template<class T>
  shared_ptr<T> atomic_load(const shared_ptr<T> * p);
template<class T>
  shared_ptr<T> atomic_load_explicit(const shared_ptr<T> * p, memory_order mo);

template<class T>
  void atomic_store(shared_ptr<T> * p, shared_ptr<T> r);
template<class T>
  void atomic_store_explicit(shared_ptr<T> * p, shared_ptr<T> r, memory_order mo);

template<class T>
  shared_ptr<T> atomic_exchange(shared_ptr<T> * p, shared_ptr<T> r);
template<class T>
  shared_ptr<T> atomic_exchange_explicit(shared_ptr<T> * p, shared_ptr<T> r, memory_order mo);

template<class T>
  bool atomic_compare_exchange_weak(
    shared_ptr<T> * p, shared_ptr<T> * v, shared_ptr<T> w);
template<class T>
  bool atomic_compare_exchange_strong(
    shared_ptr<T> * p, shared_ptr<T> * v, shared_ptr<T> w);
template<class T>
  bool atomic_compare_exchange_weak_explicit(
    shared_ptr<T> * p, shared_ptr<T> * v, shared_ptr<T> w,
    memory_order success, memory_order failure);
template<class T>
  bool atomic_compare_exchange_strong_explicit(
    shared_ptr<T> * p, shared_ptr<T> * v, shared_ptr<T> w,
    memory_order success, memory_order failure);

// 20.9.11.6 Hash support
template <class T> struct hash;
template <class T, class D> struct hash<unique_ptr<T, D> >;
template <class T> struct hash<shared_ptr<T> >;

// 20.9.12, Pointer safety
eenum class pointer_safety { relaxed, preferred, strict };
  void declare_reachable(void *p);
template <class T> T *undeclare_reachable(T *p);
  void declare_no_pointers(char *p, size_t n);
  void undeclare_no_pointers(char *p, size_t n);
  pointer_safety get_pointer_safety();

// 20.9.13, Pointer alignment function
  void *align(std::size_t alignment, std::size_t size,
              void * &ptr, std::size_t &space);
20.9.1 Allocator argument tag

namespace std {
    struct allocator_arg_t { 
    const allocator_arg_t allocator_arg = allocator_arg_t();
}
}

1 The allocator_arg_t struct is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, several types (see tuple 20.4) have constructors with allocator_arg_t as the first argument, immediately followed by an argument of a type that satisfies the Allocator requirements (20.2.5).

20.9.2 uses_allocator

20.9.2.1 uses_allocator trait

template <class T, class Alloc> struct uses_allocator;

1 Remark: automatically detects whether T has a nested allocator_type that is convertible from Alloc. Meets the BinaryTypeTrait requirements (20.7.1). The implementation shall provide a definition that is derived from true_type if a type T::allocator_type exists and is_convertible<Alloc, T::allocator_type>::value != false, otherwise it shall be derived from false_type. A program may specialize this template to derive from true_type for a user-defined type T that does not have a nested allocator_type but nonetheless can be constructed with an allocator where either:

— the first argument of a constructor has type allocator_arg_t and the second argument has type Alloc or
— the last argument of a constructor has type Alloc.

20.9.2.2 uses-allocator construction

Uses-allocator construction with allocator Alloc refers to the construction of an object obj of type T, using constructor arguments v1, v2, ..., vN of types V1, V2, ..., VN, respectively, and an allocator alloc of type Alloc, according to the following rules:

— if uses_allocator<T, Alloc>::value is false and is_constructible<T, V1, V2, ..., VN>::value is true, then obj is initialized as obj(v1, v2, ..., vN);
— otherwise, if uses_allocator<T, Alloc>::value is true and is_constructible<T, allocator_arg_t, Alloc, V1, V2, ..., VN>::value is true, then obj is initialized as obj(allocator_arg, alloc, v1, v2, ..., vN);
— otherwise, if uses_allocator<T, Alloc>::value is true and is_constructible<T, V1, V2, ..., VN>::value is true, then obj is initialized as obj(v1, v2, ..., vN, alloc);
— otherwise, the request for uses-allocator construction is ill-formed. [Note: an error will result if uses_allocator<T, Alloc>::value is true but the specific constructor does not take an allocator. This definition prevents a silent failure to pass the allocator to an element. — end note]

20.9.3 Pointer traits

The class template pointer_traits supplies a uniform interface to certain attributes of pointer-like types.

namespace std {
    template <class Ptr> struct pointer_traits {

§ 20.9.3

554
typedef Ptr pointer;
typedef see below element_type;
typedef see below difference_type;

template <class U> using rebind = see below;

static pointer pointer_to(see below r);
};
template <class T> struct pointer_traits<T*> {
    typedef T element_type;
    typedef T* pointer;
    typedef ptrdiff_t difference_type;

    template <class U> using rebind = U*;

    static pointer pointer_to(see below r);
};

20.9.3.1 Pointer traits member types

typedef see below element_type;

Type: Ptr::element_type if such a type exists; otherwise, T if Ptr is a class template instantiation of the form SomePointer<T, Args>, where Args is zero or more type arguments; otherwise, the specialization is ill-formed.

typedef see below difference_type;

Type: Ptr::difference_type if such a type exists; otherwise, std::ptrdiff_t.

template <class U> using rebind = see below;

Template alias: Ptr::rebind<U> if such a type exists; otherwise, SomePointer<U, Args> if Ptr is a class template instantiation of the form SomePointer<T, Args>, where Args is zero or more type arguments; otherwise, the instantiation of rebind is ill-formed.

20.9.3.2 Pointer traits member functions

static pointer pointer_traits::pointer_to(see below r);
static pointer pointer_traits<T*>::pointer_to(see below r);

Remark: if element_type is (possibly cv-qualified) void, the type of r is unspecified; otherwise, it is T&.

Returns: the first template function returns a dereferenceable pointer to r obtained by calling Ptr::pointer_to(r); an instantiation of this function is ill-formed if Ptr does not have a matching pointer_to static member function. The second template function returns std::addressof(r).

20.9.4 Allocator traits

The class template allocator_traits supplies a uniform interface to all allocator types. An allocator cannot be a non-class type, however, even if allocator_traits supplies the entire required interface. [ Note: thus, it is always possible to create a derived class from an allocator. — end note ]
namespace std {
    template <class Alloc> struct allocator_traits {
        typedef Alloc allocator_type;
        typedef typename Alloc::value_type value_type;
        typedef see below pointer;
        typedef see below const_pointer;
        typedef see below void_pointer;
        typedef see below const_void_pointer;
        typedef see below difference_type;
        typedef see below size_type;
        typedef see below propagate_on_container_copy_assignment;
        typedef see below propagate_on_container_move_assignment;
        typedef see below propagate_on_container_swap;
        template <class T> using rebind_alloc = see below;
        template <class T> using rebind_traits = allocator_traits<rebind_alloc<T>>;
        static pointer allocate(Alloc& a, size_type n);
        static pointer allocate(Alloc& a, size_type n, const_void_pointer hint);
        static void deallocate(Alloc& a, pointer p, size_type n);
        template <class T, class... Args>
            static void construct(Alloc& a, T* p, Args&&... args);
        template <class T>
            static void destroy(Alloc& a, T* p);
        static size_type max_size(const Alloc& a);
        static Alloc select_on_container_copy_construction(const Alloc& rhs);
    };
}

20.9.4.1 Allocator traits member types

typedef see below pointer;

1  Type: Alloc::pointer if such a type exists; otherwise, value_type*.

typedef see below const_pointer;

2  Type: Alloc::const_pointer if such a type exists; otherwise, pointer_traits<pointer>::rebind<const value_type>.

typedef see below void_pointer;

3  Type: Alloc::void_pointer if such a type exists; otherwise, pointer_traits<pointer>::rebind<void>.

typedef see below const_void_pointer;

4  Type: Alloc::const_void_pointer if such a type exists; otherwise, pointer_traits<pointer>::rebind<const void>.

§ 20.9.4.1

556
typedef see below difference_type;

5    Type: Alloc::difference_type if such a type exists; otherwise, ptrdiff_t.

typedef see below size_type;

6    Type: Alloc::size_type if such a type exists; otherwise, size_t.

typedef see below propagate_on_container_copy_assignment;

7    Type: Alloc::propagate_on_container_copy_assignment if such a type exists, otherwise false_type.

typedef see below propagate_on_container_move_assignment;

8    Type: Alloc::propagate_on_container_move_assignment if such a type exits, otherwise false_type.

typedef see below propagate_on_container_swap;

9    Type: Alloc::propagate_on_container_swap if such a type exits, otherwise false_type.

template <class T> using rebind_alloc = see below;

10   Template alias: Alloc::rebind<T>::other if such a type exists; otherwise, Alloc<T, Args> if Alloc is a class template instantiation of the form Alloc<U, Args>, where Args is zero or more type arguments; otherwise, the instantiation of rebind_alloc is ill-formed.

20.9.4.2 Allocator traits static member functions

[allocator.traits.members]

static pointer allocate(Alloc& a, size_type n);

1    Returns: a.allocate(n).

static pointer allocate(Alloc& a, size_type n, const void_pointer hint);

2    Returns: a.allocate(n, hint) if that expression is well-formed; otherwise, a.allocate(n).

static void deallocate(Alloc& a, pointer p, size_type n);

3    Effects: calls a.deallocate(p, n).

template <class T, class... Args>

static void construct(Alloc& a, T* p, Args&&... args);

4    Effects: calls a.construct(p, std::forward<Args>(args)... if that call is well-formed; otherwise, invokes ::new (static_cast<void*>(p)) T(std::forward<Args>(args)...).

template <class T>

static void destroy(Alloc& a, T* p);

5    Effects: calls a.destroy(p) if that call is well-formed; otherwise, invokes p->~T().

static size_type max_size(Alloc& a);

6    Returns: a.max_size() if that expression is well-formed; otherwise, numeric_limits<size_type>::max().

static Alloc select_on_container_copy_construction(const Alloc& rhs);

7    Returns: rhs.select_on_container_copy_construction() if that expression is well-formed; otherwise, rhs.
20.9.5 The default allocator

namespace std {
    template <class T> class allocator;

    // specialize for void:
    template <> class allocator<void> {
        public:
            typedef void* pointer;
            typedef const void* const_pointer;
            // reference-to-void members are impossible.
            typedef void value_type;
            template <class U> struct rebind { typedef allocator<U> other; };
    };

    template <class T> class allocator {
        public:
            typedef size_t size_type;
            typedef ptrdiff_t difference_type;
            typedef T* pointer;
            typedef const T* const_pointer;
            typedef T& reference;
            typedef const T& const_reference;
            typedef T value_type;
            template <class U> struct rebind { typedef allocator<U> other; };
            allocator() throw();
            allocator(const allocator&) throw();
            template <class U> allocator(const allocator<U>&) throw();
            ~allocator() throw();
            pointer address(reference x) const;
            const_pointer address(const_reference x) const;

            pointer allocate(
                size_type, allocator<void>::const_pointer hint = 0);
            void deallocate(pointer p, size_type n);
            size_type max_size() const throw();

            template<class U, class... Args>
            void construct(U* p, Args&&... args);
            template <class U>
            void destroy(U* p);
    };
}

20.9.5.1 allocator members

Except for the destructor, member functions of the default allocator shall not introduce data races (1.10) as a result of concurrent calls to those member functions from different threads. Calls to these functions that allocate or deallocate a particular unit of storage shall occur in a single total order, and each such deallocation call shall happen before the next allocation (if any) in this order.

pointer address(reference x) const;
Returns: The actual address of the object referenced by \texttt{x}, even in the presence of an overloaded operator

\begin{verbatim}
const_pointer address(const_reference x) const;
\end{verbatim}

Returns: The actual address of the object referenced by \texttt{x}, even in the presence of an overloaded operator

\begin{verbatim}
pointer allocate(size_type n, allocator<void>::const_pointer hint=0);
\end{verbatim}

\begin{verbatim}
[ Note: In a container member function, the address of an adjacent element is often a good choice to pass for the \texttt{hint} argument. — end note ]
\end{verbatim}

Returns: a pointer to the initial element of an array of storage of size \texttt{n} \times \texttt{sizeof(T)}, aligned appropriately for objects of type \texttt{T}. It is implementation-defined whether over-aligned types are supported (3.11).

Remark: the storage is obtained by calling ::\texttt{operator new(std::size_t)} (18.6.1), but it is unspecified when or how often this function is called. The use of \texttt{hint} is unspecified, but intended as an aid to locality if an implementation so desires.

Throws: \texttt{bad_alloc} if the storage cannot be obtained.

\begin{verbatim}
void deallocate(pointer p, size_type n);
\end{verbatim}

\begin{verbatim}
Requires: \texttt{p} shall be a pointer value obtained from \texttt{allocate()}. \texttt{n} shall equal the value passed as the first argument to the invocation of allocate which returned \texttt{p}.
\end{verbatim}

Effects: Deallocates the storage referenced by \texttt{p}.

Remarks: Uses ::\texttt{operator delete(void*)} (18.6.1), but it is unspecified when this function is called.

\begin{verbatim}
size_type max_size() const throw();
\end{verbatim}

Returns: the largest value \texttt{N} for which the call \texttt{allocate(N,0)} might succeed.

\begin{verbatim}
template <class U, class... Args>
void construct(U* p, Args&&... args);
\end{verbatim}

\begin{verbatim}
Effects: \texttt{::new((void *)p)} \texttt{U(std::forward<Args>(args)...)}
\end{verbatim}

\begin{verbatim}
template <class U>
void destroy(U* p);
\end{verbatim}

\begin{verbatim}
Effects: \texttt{p-> U()}
\end{verbatim}

20.9.5.2 allocator globals

\begin{verbatim}
template <class T1, class T2>
bool operator==(const allocator<T1>&, const allocator<T2>&) throw();
\end{verbatim}

Returns: true.

\begin{verbatim}
template <class T1, class T2>
bool operator!=(const allocator<T1>&, const allocator<T2>&) throw();
\end{verbatim}

Returns: false.
20.9.6 Scoped allocator adaptor

The class template `scoped_allocator_adaptor` is an allocator template that specifies the memory resource (the outer allocator) to be used by a container (as any other allocator does) and also specifies an inner allocator resource to be passed to the constructor of every element within the container. This adaptor is instantiated with one outer and zero or more inner allocator types. If instantiated with only one allocator type, the inner allocator becomes the `scoped_allocator_adaptor` itself, thus using the same allocator resource for the container and every element within the container and, if the elements themselves are containers, each of their elements recursively. If instantiated with more than one allocator, the first allocator is the outer allocator for use by the container, the second allocator is passed to the constructors of the container’s elements, and, if the elements themselves are containers, the third allocator is passed to the elements’ elements, and so on. If containers are nested to a depth greater than the number of allocators, the last allocator is used repeatedly, as in the single-allocator case, for any remaining recursions. [Note: the `scoped_allocator_adaptor` is derived from the outer allocator type so it can be substituted for the outer allocator type in most expressions. — end note]

```cpp
namespace std {
  template <class OuterAlloc, class... InnerAllocs>
  class scoped_allocator_adaptor : public OuterAlloc {
    private:
      typedef allocator_traits<OuterAlloc> OuterTraits; // exposition only
      scoped_allocator_adaptor<InnerAllocs...> inner; // exposition only
    public:
      typedef OuterAlloc outer_allocator_type;
      typedef see below inner_allocator_type;
      typedef typename OuterTraits::value_type value_type;
      typedef typename OuterTraits::size_type size_type;
      typedef typename OuterTraits::difference_type difference_type;
      typedef typename OuterTraits::pointer pointer;
      typedef typename OuterTraits::const_pointer const_pointer;
      typedef typename OuterTraits::void_pointer void_pointer;
      typedef typename OuterTraits::const_void_pointer const_void_pointer;

      typedef see below propagate_on_container_copy_assignment;
      typedef see below propagate_on_container_move_assignment;
      typedef see below propagate_on_container_swap;

    template <class Tp>
    struct rebind {
      typedef scoped_allocator_adaptor<
        OuterTraits::template rebind_alloc<Tp>, InnerAllocs...> other;
    };

    scoped_allocator_adaptor();
    template <class OuterA2>
    scoped_allocator_adaptor(OuterA2&& outerAlloc,
                            const InnerAllocs&... innerAllocs)
    scoped_allocator_adaptor(const scoped_allocator_adaptor& other);

    template <class OuterA2>
    scoped_allocator_adaptor(const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& other);
    template <class OuterA2>
    scoped_allocator_adaptor(const scoped_allocator_adaptor<OuterA2, InnerAllocs...>&& other);
  }
}
```
```cpp
scoped_allocator_adaptor();

inner_allocator_type& inner_allocator();
const inner_allocator_type& inner_allocator() const;
outer_allocator_type& outer_allocator();
const outer_allocator_type& outer_allocator() const;

pointer allocate(size_type n);
pointer allocate(size_type n, const_void_pointer hint);
void deallocate(pointer p, size_type n);
size_type max_size() const;

template <class T, class... Args>
void construct(T* p, Args&& args);

template <class T1, class T2, class... Args1, class... Args2>
void construct(pair<T1, T2>* p, piecewise_construct t,
               tuple<Args1...> x, tuple<Args2...> y);

template <class T1, class T2>
void construct(pair<T1, T2>* p);

template <class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, const pair<U, V>& x);

template <class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, pair<U, V>&& x);

template <class T>
void destroy(T* p);

scoped_allocator_adaptor select_on_container_copy_construction() const;

};

template <class OuterA1, class OuterA2, class... InnerAllocs>
bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
                const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b);

template <class OuterA1, class OuterA2, class... InnerAllocs>
bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
                const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b);

20.9.6.1 Scoped allocator adaptor member types

typedef see below inner_allocator_type;

1 Type: scoped_allocator_adaptor<OuterAlloc> if sizeof...(InnerAllocs) is zero; otherwise,
       scoped_allocator_adaptor<InnerAllocs...>.

typedef see below propagate_on_container_copy_assignment;

2 Type: true_type if allocator_traits<A>::propagate_on_container_copy_assignment::value is
        true for any A in the set of OuterAlloc and InnerAllocs...; otherwise, false_type.

typedef see below propagate_on_container_move_assignment;
```

---

20.9.6.1 Scoped allocator adaptor member types

1 Type: scoped_allocator_adaptor<OuterAlloc> if sizeof...(InnerAllocs) is zero; otherwise,
       scoped_allocator_adaptor<InnerAllocs...>.

typedef see below propagate_on_container_copy_assignment;

2 Type: true_type if allocator_traits<A>::propagate_on_container_copy_assignment::value is
        true for any A in the set of OuterAlloc and InnerAllocs...; otherwise, false_type.

typedef see below propagate_on_container_move_assignment;
Type: true_type if allocator_traits<A>::propagate_on_container_move_assignment::value is true for any A in the set of OuterAlloc and InnerAllo... otherwise, false_type.

typedef see below propagate_on_container_swap;

Type: true_type if allocator_traits<A>::propagate_on_container_swap::value is true for any A in the set of OuterAlloc and InnerAllo... otherwise, false_type.

20.9.6.2 Scoped allocator adaptor constructors

scoped_allocator_adaptor();

Effects: value-initializes the OuterAlloc base class and the inner allocator object.

template <class OuterA2>
scoped_allocator_adaptor(OuterA2&& outerAlloc,
    const InnerAllocs&... innerAllocs);

Requires: OuterAlloc shall be constructible from OuterA2.

Effects: initializes the OuterAlloc base class with std::forward<OuterA2>(outerAlloc) and inner with innerAllocs... (hence recursively initializing each allocator within the adaptor with the corresponding allocator from the argument list).

scoped_allocator_adaptor(const scoped_allocator_adaptor& other);

Effects: initializes each allocator within the adaptor with the corresponding allocator from other.

template <class OuterA2>
scoped_allocator_adaptor(const scoped_allocator_adaptor<OuterA2,
    InnerAllocs...>& other);

Requires: OuterAlloc shall be constructible from OuterA2.

Effects: initializes each allocator within the adaptor with the corresponding allocator from other.

template <class OuterA2>
scoped_allocator_adaptor(const scoped_allocator_adaptor<OuterA2,
    InnerAllocs...>&& other);

Requires: OuterAlloc shall be constructible from OuterA2.

Effects: initializes each allocator within the adaptor with the corresponding allocator rvalue from other.

20.9.6.3 Scoped allocator adaptor members

In the construct member functions, OUTERMOST(x) is x if x does not have an outer_allocator() member function and OUTERMOST(x.outer_allocator()) otherwise; OUTERMOST_ALLOC_TRAITS(x) is allocator_traits<decltype(OUTERMOST(x))>. [Note: OUTERMOST(x) and OUTERMOST_ALLOC_TRAITS(x) are recursive operations. It is incumbent upon the definition of outer_allocator() to ensure that the recursion terminates. It will terminate for all instantiations of scoped_allocator_adaptor. — end note]

inner_allocator_type& inner_allocator();
const inner_allocator_type& inner_allocator() const;

Returns: *this if sizeof...(InnerAllocs) is zero; otherwise, inner.
outer_allocator_type& outer_allocator();

Returns: static_cast<Outer&>(*this).

const outer_allocator_type& outer_allocator() const;

Returns: static_cast<const Outer&>(*this).

pointer allocate(size_type n);

Returns: allocator_traits<OuterAlloc>::allocate(outer_allocator(), n).

pointer allocate(size_type n, const_void_pointer hint);

Returns: allocator_traits<OuterAlloc>::allocate(outer_allocator(), n, hint).

void deallocate(pointer p, size_type n);

Effects: allocator_traits<OuterAlloc>::deallocate(outer_allocator(), p, n);

size_type max_size() const;

returns allocator_traits<OuterAlloc>::max_size(outer_allocator()).

template <class T, class... Args>
void construct(T* p, Args&&... args);

Effects:

— If uses_allocator<T, inner_allocator_type>::value is false and is_constructible<T, Args...>::value is true, calls OUTERMOST_ALLOC_TRAITS(*this):: construct(OUTERMOST(*this), p, std::forward<Args>(args)...).

— Otherwise, if uses_allocator<T, inner_allocator_type>::value is true and is_constructible<T, allocator_arg_t, inner_allocator_type, Args...>::value is true, calls OUTERMOST_ALLOC_TRAITS(*this):: construct(OUTERMOST(*this), p, allocator_arg, inner_allocator(), std::forward<Args>(args)...).

— Otherwise, the program is ill-formed. [Note: an error will result if uses_allocator evaluates to true but the specific constructor does not take an allocator. This definition prevents a silent failure to pass an inner allocator to a contained element. — end note]

template <class T1, class T2, class... Args1, class... Args2>
void construct(pair<T1, T2>* p, piecewise_construct_t,
  tuple<Args1...> x, tuple<Args2...> y);

Requires: all of the types in Args1 and Args2 shall be CopyConstructible (Table 35).

Effects: Constructs a tuple object xprime from x by the following rules:

— If uses_allocator<T1, inner_allocator_type>::value is false and is_constructible<T1, Args1...>::value is true, then xprime is x.

— Otherwise, if uses_allocator<T1, inner_allocator_type>::value is true and is_constructible<T1, allocator_arg_t, inner_allocator_type, Args1...>::value is true, then xprime is tuple_
cat(tuple<allocator_arg_t, inner_allocator_type&>(allocator_arg, inner_allocator_type()), x).

— Otherwise, if uses_allocator<T1, inner_allocator_type>::value is true and is_constructible<T1, Args1..., inner_allocator_type>::value is true, then xprime is tuple_cat(x, tuple<inner_allocator_type&>(inner_allocator_type())).

— Otherwise, the program is ill-formed.

and constructs a tuple object yprime from y by the following rules:

— If uses_allocator<T2, inner_allocator_type>::value is false and is_constructible<T2, Args2...>::value is true, then yprime is y.

— Otherwise, if uses_allocator<T2, inner_allocator_type>::value is true and is_constructible<T2, allocator_arg_t, inner_allocator_type, Args2...>::value is true, then yprime is tuple_cat(tuple<allocator_arg_t, inner_allocator_type&>(allocator_arg, inner_allocator_type()), y).

— Otherwise, if uses_allocator<T2, inner_allocator_type>::value is true and is_constructible<T2, Args2..., inner_allocator_type>::value is true, then yprime is tuple_cat(y, tuple<inner_allocator_type&>(inner_allocator_type())).

— Otherwise, the program is ill-formed.

then calls OUTERMOST_ALLOC_TRAITS(*this)::construct(OUTERMOST(*this), p, piecewise_construct, xprime, yprime).

template <class T1, class T2>
void construct(pair<T1, T2>* p);

Effects: equivalent to this->construct(p, piecewise_construct, tuple<>(), tuple<>()).

template <class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, U&& x, V&& y);

Effects: equivalent to this->construct(p, piecewise_construct, pack_arguments(std::forward<U>(x)), pack_arguments(std::forward<V>(y))).

template <class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, const pair<U, V>& x);

Effects: equivalent to this->construct(p, piecewise_construct, pack_arguments(x.first), pack_arguments(x.second)).

template <class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, pair<U, V>&& x);

Effects: equivalent to this->construct(p, piecewise_construct, pack_arguments(std::forward<U>(x.first)), pack_arguments(std::forward<V>(x.second))).

template <class T>
void destroy(T* p);

Effects: calls OUTERMOST_ALLOC_TRAITS(*this)::destroy(OUTERMOST(*this), p).

scoped_allocator_adaptor select_on_container_copy_construction() const;
Returns: a new scoped_allocator_adaptor object where each allocator A in the adaptor is initialized from the result of calling allocator_traits<A>::select_on_container_copy_construction() on the corresponding allocator in *this.

20.9.7 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 (24.2.4).

```cpp
namespace std {
    template <class OutputIterator, class T>
    class raw_storage_iterator
        : public iterator<input_iterator_tag,void,void,void,void> {
        public:
            explicit raw_storage_iterator(OutputIterator x);
            raw_storage_iterator<OutputIterator,T>& operator*();
            raw_storage_iterator<OutputIterator,T>& operator=(const T& element);
            raw_storage_iterator<OutputIterator,T>& operator++();
            raw_storage_iterator<OutputIterator,T> operator++(int);
    };
}
```

raw_storage_iterator(OutputIterator x);

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.

raw_storage_iterator<OutputIterator,T>& operator++();

Returns: A reference to the iterator.

raw_storage_iterator<OutputIterator,T> operator++(int);

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.

20.9.8 Temporary buffers

```cpp
template <class T>
pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n);
```

Effects: Obtains a pointer to storage sufficient to store up to n adjacent T objects. It is implementation-defined whether over-aligned types are supported (3.11).

Returns: A pair containing the buffer’s address and capacity (in the units of sizeof(T)), or a pair of 0 values if no storage can be obtained or if n <= 0.
template <class T> void return_temporary_buffer(T* p);

Effects: Deallocation the buffer to which p points.

Requires: The buffer shall have been previously allocated by get_temporary_buffer.

20.9.9 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 (24.2.3). In all of the following algorithms, the formal template parameter ForwardIterator is required to satisfy the requirements of a forward iterator (24.2.5), and is required to have the property that no exceptions are thrown from increment, assignment, comparison, or dereference of valid iterators. In the following algorithms, if an exception is thrown there are no effects.

20.9.9.1 addressof

template <class T> T* addressof(T& r);

Returns: the actual address of the object or function referenced by r, even in the presence of an overloaded operator&.

Throws: nothing.

20.9.9.2 uninitialized_copy

template <class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_copy(InputIterator first, InputIterator last, ForwardIterator result);

Effects:

for (; first != last; ++result, ++first)
::new (static_cast<void*>(&*result))
typename iterator_traits<ForwardIterator>::value_type(*first);

Returns: result

20.9.9.3 uninitialized_fill

template <class ForwardIterator, class T>
void uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x);

§ 20.9.9.3
### 20.9.10 Class template `unique_ptr`

A *unique pointer* is an object that owns another object and manages that other object through a pointer. More precisely, a unique pointer is an object `u` that stores a pointer to a second object `p` and will dispose of `p` when `u` is itself destroyed (e.g., when leaving block scope (6.7)). In this context, `u` is said to *own* `p`.

The mechanism by which `u` disposes of `p` is known as `p`'s associated *deleter*, a function object whose correct invocation results in `p`'s appropriate disposition (typically its deletion).

Let the notation `u.p` denote the pointer stored by `u`, and let `u.d` denote the associated deleter. Upon request, `u` can *reset* (replace) `u.p` and `u.d` with another pointer and deleter, but must properly dispose of its owned object via the associated deleter before such replacement is considered completed.

Additionally, `u` can, upon request, *transfer ownership* to another unique pointer `u2`. Upon completion of such a transfer, the following postconditions hold:

- `u2.p` is equal to the pre-transfer `u.p`,
- `u.p` is equal to `nullptr`, and
- if the pre-transfer `u.d` maintained state, such state has been transferred to `u2.d`.

As in the case of a reset, `u2` must properly dispose of its pre-transfer owned object via the pre-transfer associated deleter before the ownership transfer is considered complete. [Note: A deleter’s state need never be copied, only moved or swapped as ownership is transferred. — end note]

Each object of a type `U` instantiated form the `unique_ptr` template specified in this subclause has the strict ownership semantics, specified above, of a unique pointer. In partial satisfaction of these semantics, each such `U` is *MoveConstructible* and *MoveAssignable*, but is not *CopyConstructible* nor *CopyAssignable*. The template parameter `T` of `unique_ptr` may be an incomplete type.

[Note: The uses of `unique_ptr` include providing exception safety for dynamically allocated memory, passing ownership of dynamically allocated memory to a function, and returning dynamically allocated memory from a function. — end note]

```cpp
namespace std {
    template<class T> struct default_delete;
    template<class T> struct default_delete<T[]>
    {
        template<class T, class D = default_delete<T>> class unique_ptr;
```
template<class T, class D> class unique_ptr<T[], D>;

template<class T, class 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);

20.9.10.1 Default deleters

20.9.10.1.1 In general

1 The class template default_delete serves as the default deleter (destruction policy) for the class template unique_ptr.

2 The template parameter T of default_delete may be an incomplete type.

20.9.10.1.2 default_delete

namespace std {
    template <class T> struct default_delete {
        constexpr default_delete();
        template <class U> default_delete(const default_delete<U>& other);
        void operator()(T* ptr) const;
    };
}

constexpr default_delete();

1 Effects: Default constructs a default_delete object.

template <class U> default_delete(const default_delete<U>& other);

2 Effects: Constructs a default_delete object from another default_delete<U> object.

3 Remarks: This constructor shall not participate in overload resolution unless U* is implicitly convertible to T*.

void default_delete<T*()>(T *ptr) const;

4 Effects: calls delete on ptr.

5 Remarks: If T is an incomplete type, the program is ill-formed.

20.9.10.1.3 default_delete<T[]>

§ 20.9.10.1.3
namespace std {
    template <class T> struct default_delete<T[]> {
        constexpr default_delete();
        void operator()(T*) const;
        template <class U> void operator()(U*) const = delete;
    };
}

void operator()(T* ptr) const;

Effects: calls delete[] on ptr.

Remarks: If T is an incomplete type, the program is ill-formed.

20.9.10.2 unique_ptr for single objects [unique.ptr.single]

namespace std {
    template <class T, class D = default_delete<T>> class unique_ptr {
        public:
            typedef see below pointer;
            typedef T element_type;
            typedef D deleter_type;

            // constructors
            constexpr unique_ptr();
            explicit unique_ptr(pointer p);
            unique_ptr(pointer p, implementation-defined d1);
            unique_ptr(pointer p, implementation-defined d2);
            unique_ptr(unique_ptr&& u);
            unique_ptr(nullptr_t) : unique_ptr() { }
            template <class U, class E>
                unique_ptr(unique_ptr<U, E>&& u);
            template <class U>
                explicit unique_ptr(auto_ptr<U>& u);
            template <class U>
                unique_ptr(auto_ptr<U>&& 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=(nullptr_t);

            // observers
            typename add_lvalue_reference<T>::type operator*() const;
            pointer operator->() const;
            pointer get() const;
            deleter_type& get_deleter();
            const deleter_type& get_deleter() const;
            explicit operator bool() const;

            // modifiers
            pointer release();
    };
}

§ 20.9.10.2
```cpp
void reset(pointer p = pointer());
void swap(unique_ptr& u);

// disable copy from lvalue
unique_ptr(const unique_ptr&) = delete;
unique_ptr& operator=(const unique_ptr&) = delete;
```

1. The default type for the template parameter `D` is `default_delete`. A client-supplied template argument `D` shall be a function object type (20.8), lvalue-reference to function, or lvalue-reference to function object type for which, given a value `d` of type `D` and a value `ptr` of type `unique_ptr<T, D>::pointer`, the expression `d(ptr)` is valid and has the effect of disposing of the pointer as appropriate for that deleter.

2. If the deleter's type `D` is not a reference type, `D` shall satisfy the requirements of `Destructible` (38).

3. If the type `remove_reference<D>::type::pointer` exists, then `unique_ptr<T, D>::pointer` shall be a synonym for `remove_reference<D>::type::pointer`. Otherwise `unique_ptr<T, D>::pointer` shall be a synonym for `T*`. The type `unique_ptr<T, D>::pointer` shall satisfy the requirements of `NullablePointer` (20.2.3).

4. [Example: Given an allocator type `X` (20.2.5) and letting `A` be a synonym for `allocator_traits<X>`, the types `A::pointer`, `A::const_pointer`, `A::void_pointer`, and `A::const_void_pointer` may be used as `unique_ptr<T, D>::pointer`. — end example]

20.9.10.2.1 unique_ptr constructors

```cpp
constexpr unique_ptr();
constexpr unique_ptr();
```

1. **Requires:** `D` shall satisfy the requirements of `DefaultConstructible` (33), and that construction shall not throw an exception.

2. **Effects:** Constructs a `unique_ptr` object that owns nothing, value-initializing the stored pointer and the stored deleter.

3. **Postconditions:** `get() == nullptr`.

4. **Throws:** nothing.

5. **Remarks:** If this constructor is instantiated with a pointer type or reference type for the template argument `D`, the program is ill-formed.

```cpp
unique_ptr(pointer p);
```

6. **Requires:** `D` shall satisfy the requirements of `DefaultConstructible` (33), and that construction shall not throw an exception.

7. **Effects:** Constructs a `unique_ptr` which owns `p`, initializing the stored pointer with `p` and value-initializing the stored deleter.

8. **Postconditions:** `get() == p`.

9. **Throws:** nothing.

10. **Remarks:** If this constructor is instantiated with a pointer type or reference type for the template argument `D`, the program is ill-formed.

```cpp
unique_ptr(pointer p, see below d1);
unique_ptr(pointer p, see below d2);
```
The signature of these constructors depends upon whether D is a reference type. If D is non-reference type A, then the signatures are:

unique_ptr(pointer p, const A& d);
unique_ptr(pointer p, A&& d);

If D is an lvalue-reference type A&, then the signatures are:

unique_ptr(pointer p, A& d);
unique_ptr(pointer p, A&& d);

If D is an lvalue-reference type const A&, then the signatures are:

unique_ptr(pointer p, const A& d);
unique_ptr(pointer p, const A&& d);

Requires:

— If D is not an lvalue-reference type then
  — If d is an lvalue or const rvalue then the first constructor of this pair will be selected. D shall satisfy the requirements of CopyConstructible (Table 35), and the copy constructor of D shall not throw an exception. This unique_ptr will hold a copy of d.
  — Otherwise, d is a non-const rvalue and the second constructor of this pair will be selected. D shall satisfy the requirements of MoveConstructible (Table 34), and the move constructor of D shall not throw an exception. This unique_ptr will hold a value move constructed from d.

— Otherwise D is an lvalue-reference type. d shall be reference-compatible with one of the constructors. If d is an rvalue, it will bind to the second constructor of this pair and the program is ill-formed. [Note: The diagnostic could be implemented using a static_assert which assures that D is not a reference type. — end note] Else d is an lvalue and will bind to the first constructor of this pair. The type which D references need not be CopyConstructible nor MoveConstructible. This unique_ptr will hold a D which refers to the lvalue d. [Note: D may not be an rvalue-reference type. — end note]

Effects: Constructs a unique_ptr object which owns p, initializing the stored pointer with p and initializing the deleter as described above.

Postconditions: get() == p. get_deleter() returns a reference to the stored deleter. If D is a reference type then get_deleter() returns a reference to the lvalue d.

Throws: nothing.

[Example:

D d;
unique_ptr<int, D> p1(new int, D()); // D must be MoveConstructible
unique_ptr<int, D> p2(new int, d); // D must be CopyConstructible
unique_ptr<int, D&> p3(new int, d); // p3 holds a reference to d
unique_ptr<int, const D&> p4(new int, D()); // error: rvalue deleter object combined // with reference deleter type

— end example]

unique_ptr(unique_ptr&& u);
Requires: If D is not a reference type, D shall satisfy the requirements of MoveConstructible (34). Construction of the deleter from an rvalue of type D shall not throw an exception.

Effects: Constructs a unique_ptr by transferring ownership from u to *this. If D is a reference type, this deleter is copy constructed from u’s deleter; otherwise, this deleter is move constructed from u’s deleter. [Note: The deleter constructor can be implemented with std::forward<D>. — end note]

Postconditions: get() yields the value u.get() yielded before the construction. get_deleter() returns a reference to the stored deleter that was constructed from u.get_deleter(). If D is a reference type then get_deleter() and u.get_deleter() both reference the same lvalue deleter.

Throws: nothing.

```cpp
template <class U, class E> unique_ptr(unique_ptr<U, E>&& u);
```

Requires: If E is not a reference type, construction of the deleter from an rvalue of type E shall be well formed and shall not throw an exception. Otherwise, E is a reference type and construction of the deleter from an lvalue of type E shall be well formed and shall not throw an exception.

Remarks: This constructor shall not participate in overload resolution unless:

- unique_ptr<U, E>::pointer is implicitly convertible to pointer,
- U is not an array type, and
- either D is a reference type and E is the same type as D, or D is not a reference type and E is implicitly convertible to D.

Effects: Constructs a unique_ptr by transferring ownership from u to *this. If E is a reference type, this deleter is copy constructed from u’s deleter; otherwise, this deleter is move constructed from u’s deleter. [Note: The deleter constructor can be implemented with std::forward<E>. — end note]

Postconditions: get() yields the value u.get() yielded before the construction. get_deleter() returns a reference to the stored deleter that was constructed from u.get_deleter().

Throws: nothing.

```cpp
template <class U> explicit unique_ptr(auto_ptr<U>& u);
template <class U> unique_ptr(auto_ptr<U>&& u);
```

Effects: Constructs a unique_ptr object, initializing the stored pointer with u.release() and value-initializing the stored deleter.

Postconditions: get() yields the value u.get() yielded before the construction. u.get() == nullptr. get_deleter() returns a reference to the stored deleter.

Throws: Nothing.

Remarks: These constructors shall not participate in overload resolution unless U* is implicitly convertible to T* and D is the same type as default_delete<T>.

20.9.10.2.2 unique_ptr destructor

```
~unique_ptr();
```

§ 20.9.10.2.2
Requires: The expression `get_deleter()(get())` shall be well formed, shall have well-defined behavior, and shall not throw exceptions. [Note: The use of `default_delete` requires `T` to be a complete type. — end note]

Effects: If `get() == nullptr` there are no effects. Otherwise `get_deleter()(get())`.

Throws: nothing.

20.9.10.2.3 unique_ptr assignment [unique.ptr.single.asgn]

unique_ptr& operator=(unique_ptr&& u);

Requires: If `D` is not a reference type, `D` shall satisfy the requirements of `MoveAssignable` (36) and assignment of the deleter from an rvalue of type `D` shall not throw an exception. Otherwise, `D` is a reference type; `remove_reference<D>::type` shall satisfy the `CopyAssignable` requirements and assignment of the deleter from an lvalue of type `D` shall not throw an exception.

Effects: Transfers ownership from `u` to `*this` as if by calling `reset(u.release())` followed by an assignment from `std::forward<D>(u.get_deleter())`.

Returns: `*this`.

Throws: nothing.

template <class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u);

Requires: If `E` is not a reference type, assignment of the deleter from an rvalue of type `E` shall be well-formed and shall not throw an exception. Otherwise, `E` is a reference type and assignment of the deleter from an lvalue of type `E` shall be well-formed and shall not throw an exception.

Remarks: This operator shall not participate in overload resolution unless:

— `unique_ptr<U, E>::pointer` is implicitly convertible to `pointer` and

— `U` is not an array type.

Effects: Transfers ownership from `u` to `*this` as if by calling `reset(u.release())` followed by an assignment from `std::forward<D>(u.get_deleter())`.

Returns: `*this`.

Throws: nothing.

unique_ptr& operator=(nullptr_t);

Effects: `reset()`.

Postcondition: `get() == nullptr`

Returns: `*this`.

Throws: nothing.

20.9.10.2.4 unique_ptr observers [unique.ptr.single.observers]

typename add_lvalue_reference<T>::type operator*() const;

Requires: `get() != nullptr`.

Returns: `*get()`.
pointer operator->() const;
3    Requires: get() != nullptr.
4    Returns: get().
5    Throws: nothing.
6    Note: use typically requires that T be a complete type.

pointer get() const;
7    Returns: The stored pointer.
8    Throws: nothing.

deleter_type& get_deleter();
const deleter_type& get_deleter() const;
9    Returns: A reference to the stored deleter.
10    Throws: nothing.

explicit operator bool() const;
11    Returns: get() != nullptr.
12    Throws: nothing.

20.9.10.2.5 unique_ptr modifiers
[unique.ptr.single.modifiers]

pointer release();
1    Postcondition: get() == nullptr.
2    Returns: The value get() had at the start of the call to release.
3    Throws: nothing.

void reset(pointer p = pointer());
4    Requires: The expression get_deleter()(get()) shall be well formed, shall have well-defined behavior, and shall not throw exceptions.
5    Effects: assigns p to the stored pointer, and then if the old value of the stored pointer, old_p, was not equal to nullptr, calls get_deleter()(old_p). [Note: the order of these operations is significant because the call to get_deleter() may destroy *this. — end note]
6    Postconditions: get() == p.
7    Throws: nothing.

void swap(unique_ptr& u);
8    Requires: get_deleter() shall be swappable (20.2.2) and shall not throw an exception under swap.
9    Effects: Invokes swap on the stored pointers and on the stored deleters of *this and u.
10    Throws: nothing.
20.9.10.3 unique_ptr for array objects with a runtime length

namespace std {
    template <class T, class D> class unique_ptr<T[], D> {
public:
    typedef implementation-defined pointer;
    typedef T element_type;
    typedef D deleter_type;

    // constructors
    constexpr unique_ptr();
    explicit unique_ptr(pointer p);
    unique_ptr(pointer p, implementation-defined d);
    unique_ptr(pointer p, implementation-defined d);
    unique_ptr(unique_ptr&& u);
    unique_ptr(nullptr_t) : unique_ptr() { }

    // destructor
    ~unique_ptr();

    // assignment
    unique_ptr& operator=(unique_ptr&& u);
    unique_ptr& operator=(nullptr_t);

    // observers
    T& operator[](size_t i) const;
    pointer get() const;
    deleter_type& get_deleter();
    const deleter_type& get_deleter() const;
    explicit operator bool() const;

    // modifiers
    pointer release();
    void reset(pointer p = pointer());
    void reset(nullptr_t);
    template <class U> void reset(U) = delete;
    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 produce an ill-formed program.
   — 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[].

§ 20.9.10.3
Descriptions are provided below only for member functions that have behavior different from the primary template.

The template argument \( T \) shall be a complete type.

20.9.10.3.1 unique_ptr constructors

unique_ptr(pointer p);
unique_ptr(pointer p, implementation-defined d);
unique_ptr(pointer 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 pointer. [Note: One implementation technique is to create private templated overloads of these members. — end note]

20.9.10.3.2 unique_ptr observers

T& operator[](size_t i) const;

Requires: \( i < \) the size of the array to which the stored pointer points.

Returns: \( \text{get()[i]} \).

20.9.10.3.3 unique_ptr modifiers

void reset(pointer p = pointer());
void reset(nullptr_t p);

Effects: If \( \text{get()} == \text{nullptr} \) there are no effects. Otherwise \( \text{get_deleter()}(\text{get()}) \).

Postcondition: \( \text{get()} == p \).

Throws: nothing.

20.9.10.4 unique_ptr specialized algorithms

template <class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y);

Effects: Calls \( x.\text{swap}(y) \).

template <class T1, class D1, class T2, class D2>
bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

Returns: \( x.\text{get()} == y.\text{get()} \).

template <class T1, class D1, class T2, class D2>
bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

Returns: \( x.\text{get()} != y.\text{get()} \).

template <class T1, class D1, class T2, class D2>
bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

Returns: \( x.\text{get()} < y.\text{get()} \).

template <class T1, class D1, class T2, class D2>
bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

Returns: \( x.\text{get()} \leq y.\text{get()} \).

§ 20.9.10.4
5 \hspace{1em} \textit{Returns:} \ x.get() \leq y.get().

\begin{verbatim}
template <class T1, class D1, class T2, class D2>
bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
\end{verbatim}

6 \hspace{1em} \textit{Returns:} \ x.get() > y.get().

\begin{verbatim}
template <class T1, class D1, class T2, class D2>
bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
\end{verbatim}

7 \hspace{1em} \textit{Returns:} \ x.get() \geq y.get.

20.9.11 Smart pointers \[\text{util.smartptr}\]

20.9.11.1 Class \texttt{bad\_weak\_ptr} \[\text{util.smartptr.weakptr}\]

\begin{verbatim}
namespace std {
  class bad_weak_ptr: public std::exception {
    public:
      bad_weak_ptr();
    }
  } // namespace std
\end{verbatim}

1 An exception of type \texttt{bad\_weak\_ptr} is thrown by the \texttt{shared\_ptr} constructor taking a \texttt{weak\_ptr}.

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

2 \hspace{1em} \textit{Postconditions:} \texttt{what()} returns "\texttt{bad\_weak\_ptr}".

3 \hspace{1em} \textit{Throws:} nothing.

20.9.11.2 Class template \texttt{shared\_ptr} \[\text{util.smartptr.shared}\]

1 The \texttt{shared\_ptr} class template stores a pointer, usually obtained via \texttt{new}. \texttt{shared\_ptr} implements semantics of shared ownership; the last remaining owner of the pointer is responsible for destroying the object, or otherwise releasing the resources associated with the stored pointer. A \texttt{shared\_ptr} object is \textit{empty} if it does not own a pointer.

\begin{verbatim}
namespace std {
  template<class T> class shared_ptr {
    public:
      typedef T element_type;

      // 20.9.11.2.1, constructors:
      constexpr shared_ptr();
      template<class Y> explicit shared_ptr(Y* p);
      template<class Y, class D> shared_ptr(Y* p, D d);
      template<class Y, class D, class A> shared_ptr(Y* p, D d, A a);
      template <class D> shared_ptr(nullptr_t p, D d)
      template <class D, class A> shared_ptr(nullptr_t p, D d, A a)
      template<class Y> shared_ptr(const shared_ptr<Y>& r, 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> shared_ptr(auto_ptr<Y>&& r);
  }
\end{verbatim}
template <class Y, class D> shared_ptr(unique_ptr<Y, D>&& r);
shared_ptr(nullptr_t) : shared_ptr() {}}

// 20.9.11.2.2, destructor:
~shared_ptr();

// 20.9.11.2.3, assignment:
shared_ptr& operator=(const shared_ptr& r);
template<class Y> shared_ptr& operator=(const shared_ptr<Y>& r);
shared_ptr& operator=(shared_ptr&& r);
template<class Y> shared_ptr& operator=(shared_ptr<Y>&& r);
template<class Y> shared_ptr& operator=(auto_ptr<Y>&& r);
template <class Y, class D> shared_ptr& operator=(unique_ptr<Y, D>&& r);

// 20.9.11.2.4, 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);

// 20.9.11.2.5, observers:
T* get() const;
T& operator*() const;
T* operator->() const;
long use_count() const;
bool unique() const;
explicit operator bool() const;
template<class U> bool owner_before(shared_ptr<U> const& b) const;
template<class U> bool owner_before(weak_ptr<U> const& b) const;
);

// 20.9.11.2.6, 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);

// 20.9.11.2.7, 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);

// 20.9.11.2.8, 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);

// 20.9.11.2.9, shared_ptr specialized algorithms:
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b);

// 20.9.11.2.10, shared_ptr casts:
template<class T, class U>
    shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r);
Specializations of `shared_ptr` shall be CopyConstructible, CopyAssignable, and LessThanComparable, allowing their use in standard containers. Specializations of `shared_ptr` shall be convertible to bool, allowing their use in boolean expressions and declarations in conditions. The template parameter `T` of `shared_ptr` may be an incomplete type.

[Example:
if(shared_ptr<X> px = dynamic_pointer_cast<X>(py)) {
    // do something with px
}
— end example]

For purposes of determining the presence of a data race, member functions shall access and modify only the `shared_ptr` and `weak_ptr` objects themselves and not objects they refer to. Changes in `use_count()` do not reflect modifications that can introduce data races.

### 20.9.11.2.1 shared_ptr constructors

**constexpr 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);**

**template <class D> shared_ptr(nullptr_t p, D d);**

**template <class D, class A> shared_ptr(nullptr_t p, D d, A a);**

*Requires:* `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 (20.2.5). The copy constructor and destructor of `A` shall not throw exceptions.
Effects: Constructs a shared_ptr object that owns the object p and the deleter d. The second and fourth constructors shall use a copy of a to allocate memory for internal use.

Postconditions: use_count() == 1 && get() == p.

Throws: bad_alloc, or an implementation-defined exception when a resource other than memory could not be obtained.

Exception safety: If an exception is thrown, d(p) is called.

template<class Y> shared_ptr(const shared_ptr<Y>& r, T *p);

Effects: Constructs a shared_ptr instance that stores p and shares ownership with r.

Postconditions: get() == p && use_count() == r.use_count().

Throws: nothing.

[Note: to avoid the possibility of a dangling pointer, the user of this constructor must ensure that p remains valid at least until the ownership group of r is destroyed. — end note]

[Note: this constructor allows creation of an empty shared_ptr instance with a non-NULL stored pointer. — end note]

shared_ptr(const shared_ptr& r);

template<class Y> shared_ptr(const shared_ptr<Y>& r);

Requires: The second constructor shall not participate in the overload resolution unless Y* is implicitly convertible to T*.

Effects: If r is empty, constructs an empty shared_ptr object; otherwise, constructs a shared_ptr object that shares ownership with r.

Postconditions: get() == r.get() && use_count() == r.use_count().

Throws: nothing.

shared_ptr(shared_ptr& r);

template<class Y> shared_ptr(shared_ptr<Y>&& r);

Remark: The second constructor shall not participate in overload resolution unless Y* is convertible to T*.

Effects: Move-constrasts a shared_ptr instance from r.

Postconditions: *this shall contain the old value of r. r shall be empty. r.get() == 0.

Throws: nothing.

template<class Y> explicit shared_ptr(const weak_ptr<Y>& r);

Requires: Y* shall be convertible to T*.

Effects: Constructs a shared_ptr object that shares ownership with r and stores a copy of the pointer stored in r.

Postconditions: use_count() == r.use_count().

Throws: bad_weak_ptr when r.expired().

Exception safety: If an exception is thrown, the constructor has no effect.

template<class Y> shared_ptr(auto_ptr<Y>&& r);

§ 20.9.11.2.1
Requires: \( r.\text{release}() \) shall be convertible to \( T^* \). \( Y \) shall be a complete type. The expression \texttt{delete} \( r.\text{release}() \) shall be well formed, shall have well defined behavior, and shall not throw exceptions.

Effects: Constructs a \texttt{shared\_ptr} object that stores and \textit{owns} \( r.\text{release}() \).

Postconditions: \texttt{use\_count()} \( == 1 \&\& r.\text{get}() == 0. \)

Throws: \texttt{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.

```
template <class Y, class D> shared_ptr(unique_ptr<Y, D>&&r);
```

Effects: Equivalent to \texttt{shared\_ptr(r.\text{release}(), r.\text{get\_deleter}())} when \( D \) is not a reference type, otherwise \texttt{shared\_ptr(r.\text{release}(), \text{ref}(r.\text{get\_deleter}()))}.

Exception safety: If an exception is thrown, the constructor has no effect.

### 20.9.11.2.2 \texttt{shared\_ptr} destructor

\[ \texttt{\textasciitilde shared\_ptr()}; \]

Effects:

- If \( *\texttt{this} \) is \texttt{empty} or shares ownership with another \texttt{shared\_ptr} instance (\texttt{use\_count()} \( > 1 \)), there are no side effects.

- Otherwise, if \( *\texttt{this} \) \textit{owns} an object \( p \) and a deleter \( d \), \( d(p) \) is called.

- Otherwise, \( *\texttt{this} \) \textit{owns} a pointer \( p \), and \texttt{delete} \( p \) is called.

Throws: nothing.

\[ \texttt{\[ Note: \textit{Since the destruction of } *\texttt{this} \textit{decreases the number of instances that share ownership with } *\texttt{this} \textit{by one, after } *\texttt{this} \textit{has been destroyed all } shared\_ptr \textit{instances that shared ownership with } *\texttt{this} \textit{will report a } use\_count() \textit{that is one less than its previous value. \textit{— end note}]} \]

### 20.9.11.2.3 \texttt{shared\_ptr} assignment

\[ \texttt{shared\_ptr\& operator=(const shared\_ptr\& r);} \]

```
template<class Y> shared_ptr\& operator=(const shared_ptr<Y>& r);
template<class Y> shared_ptr\& operator=(auto_ptr<Y>&& r);
```

Effects: Equivalent to \texttt{shared\_ptr(r).\text{swap}(\ast\this)}.

Returns: \( \ast\this \).

\[ \texttt{\[ Note: \textit{The use count updates caused by the temporary object construction and destruction are not observable side effects, so the implementation may meet the effects (and the implied guarantees) via different means, without creating a temporary. In particular, in the example:} \]

\begin{verbatim}
shared_ptr<int> p(new int);
shared_ptr<void> q(p);
p = p;
q = p;
\end{verbatim}

\textit{both assignments may be no-ops. \textit{— end note}]}
shared_ptr& operator=(shared_ptr&& r);
    template<class Y> shared_ptr& operator=(shared_ptr<Y>&& r);
4  Effects: Equivalent to shared_ptr(std::move(r)).swap(*this).
5  Returns: *this.

template <class Y, class D> shared_ptr& operator=(unique_ptr<Y, D>&& r);
6  Effects: Equivalent to shared_ptr(std::move(r)).swap(*this).
7  Returns: *this

20.9.11.2.4 shared_ptr modifiers
        [util.smartptr.shared.mod]

void swap(shared_ptr& r);
1  Effects: Exchanges the contents of *this and r.
2  Throws: nothing.

void reset();
3  Effects: Equivalent to shared_ptr().swap(*this).

    template<class Y> void reset(Y* p);
4  Effects: Equivalent to shared_ptr(p).swap(*this).

    template<class Y, class D> void reset(Y* p, D d);
5  Effects: Equivalent to shared_ptr(p, d).swap(*this).

    template<class Y, class D, class A> void reset(Y* p, D d, A a);
6  Effects: Equivalent to shared_ptr(p, d, a).swap(*this).

20.9.11.2.5 shared_ptr observers
        [util.smartptr.shared.obs]

T* get() const;
1  Returns: the stored pointer.
2  Throws: nothing.

T& operator*() const;
3  Requires: get() != 0.
4  Returns: *get().
5  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().
8    \textbf{Throws:} nothing.

9    \begin{verbatim}
long use_count() const;
9    \end{verbatim}
10   \textbf{Returns:} the number of \texttt{shared\_ptr} objects, \texttt{*this} included, that \textit{share ownership} with \texttt{*this}, or 0 when \texttt{*this} is empty.
11   \textbf{Throws:} nothing.
12   \[ \text{\textit{Note: use\_count()} is not necessarily efficient.} \quad \textit{end\ note} \]
13   \begin{verbatim}
bool unique() const;
13    \end{verbatim}
14   \textbf{Returns:} \texttt{use\_count()} == 1.
15   \textbf{Throws:} nothing.
16   \[ \text{\textit{Note: unique()} may be faster than use\_count(). If you are using unique() to implement copy on write, do not rely on a specific value when get() == 0.} \quad \textit{end\ note} \]
17   \begin{verbatim}
explicit operator bool() const;
17    \end{verbatim}
18   \textbf{Returns:} \texttt{get()} != 0.
19   \textbf{Throws:} nothing.

\begin{verbatim}
template<class U> bool owner_before(shared_ptr<U> const& b) const;
template<class U> bool owner_before(weak_ptr<U> const& b) const;
\end{verbatim}
20   \textbf{Returns:} an unspecified value such that
21   \begin{itemize}
22      \item \texttt{x.owner\_before(y)} defines a strict weak ordering as defined in 25.4;
23      \item under the equivalence relation defined by \texttt{owner\_before}, \texttt{!a.owner\_before(b) && !b.owner\_before(a)}, two \texttt{shared\_ptr} or \texttt{weak\_ptr} instances are equivalent if and only if they share ownership or are both empty.
24   \end{itemize}

\subsection*{20.9.11.2.6 \texttt{shared\_ptr} creation}

\begin{verbatim}
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);
\end{verbatim}
1   \textbf{Requires:} The expression \texttt{::new (pv) T(std::forward<Args>(args)...)}, where \texttt{pv} has type \texttt{void*} and points to storage suitable to hold an object of type \texttt{T}, shall be well formed. \texttt{A} shall be an \texttt{allocator} (20.2.5). The copy constructor and destructor of \texttt{A} shall not throw exceptions.
2   \textbf{Effects:} Allocates memory suitable for an object of type \texttt{T} and constructs an object in that memory via the placement new expression \texttt{::new (pv) T()} or \texttt{::new (pv) T(std::forward<Args>(args)...)}. The template \texttt{allocate\_shared} uses a copy of \texttt{a} to allocate memory. If an exception is thrown, the functions have no effect.
3   \textbf{Returns:} A \texttt{shared\_ptr} instance that stores and owns the address of the newly constructed object of type \texttt{T}.
4   \textbf{Postconditions:} \texttt{get()} != 0 && \texttt{use\_count()} == 1
5   \textbf{Throws:} \texttt{bad\_alloc}, or an exception thrown from \texttt{A::allocate} or from the constructor of \texttt{T}.
6   \textbf{Remarks:} Implementations are encouraged, but not required, to perform no more than one memory allocation. \[ \text{\textit{Note: this provides efficiency equivalent to an intrusive smart pointer.}} \quad \textit{end\ note} \]
[Note: these functions will typically allocate more memory than sizeof(T) to allow for internal bookkeeping structures such as the reference counts. — end note]

### 20.9.11.2.7 `shared_ptr` comparison

```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:** less<V>()(a.get(), b.get()), where V is the composite pointer type (5.9) of T* and U*.

**Throws:** nothing.

[Note: Defining a comparison operator allows `shared_ptr` objects to be used as keys in associative containers. — end note]

### 20.9.11.2.8 `shared_ptr` I/O

```cpp
template<class E, class T, class Y>
basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, shared_ptr<Y> const& p);
```

**Effects:** os << p.get();

**Returns:** os.

### 20.9.11.2.9 `shared_ptr` specialized algorithms

```cpp
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b);
```

**Effects:** Equivalent to a.swap(b).

**Throws:** nothing.

### 20.9.11.2.10 `shared_ptr` casts

```cpp
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.

**Postconditions:** w.get() == static_cast<T*>(r.get()) and w.use_count() == r.use_count(), where w is the return value.

**Throws:** nothing.

[Note: The seemingly equivalent expression shared_ptr<T>(static_cast<T*>(r.get())) will eventually result in undefined behavior, attempting to delete the same object twice. — end note]

```cpp
template<class T, class U> shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r);
```
6 \textit{Requires}: The expression $\text{dynamic\_cast<T*>(r.get())}$ shall be well formed and shall have well defined behavior.

7 \textit{Returns}:

- When $\text{dynamic\_cast<T*>(r.get())}$ returns a nonzero value, a $\text{shared\_ptr<T>}$ object that stores a copy of it and \textit{shares ownership} with $r$;
- Otherwise, an \textit{empty} $\text{shared\_ptr<T>}$ object.

8 \textit{Postcondition}: $w.get() == \text{dynamic\_cast<T*>(r.get())}$, where $w$ is the return value.

9 \textit{Throws}: nothing.

[ \textit{Note}: The seemingly equivalent expression $\text{shared\_ptr<T>(dynamic\_cast<T*>(r.get()))}$ will eventually result in undefined behavior, attempting to delete the same object twice. \textit{- end note}]

\begin{verbatim}
template<class T, class U> shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r);
\end{verbatim}

10 \textit{Requires}: The expression $\text{const\_cast<T*>(r.get())}$ shall be well formed.

11 \textit{Returns}: If $r$ is empty, an empty $\text{shared\_ptr<T>}$; otherwise, a $\text{shared\_ptr<T>}$ object that stores $\text{const\_cast<T*>(r.get())}$ and shares ownership with $r$.

12 \textit{Postconditions}: $w.get() == \text{const\_cast<T*>(r.get())}$ and $w.use\_count() == r.use\_count()$, where $w$ is the return value.

13 \textit{Throws}: nothing.

[ \textit{Note}: The seemingly equivalent expression $\text{shared\_ptr<T>(const\_cast<T*>(r.get()))}$ will eventually result in undefined behavior, attempting to delete the same object twice. \textit{- end note}]

20.9.11.2.11 \textbf{get\_deleter} \hfill [util.smartptr.getdeleter]

\begin{verbatim}
template<class D, class T> D* get_deleter(const shared_ptr<T>& p);
\end{verbatim}

1 \textit{Returns}: If $p$ \textit{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 $\text{shared\_ptr}$ instance that owns $d$. [ \textit{Note}: It is unspecified whether the pointer remains valid longer than that. This can happen if the implementation doesn’t destroy the deleter until all $\text{weak\_ptr}$ instances that share ownership with $p$ have been destroyed. \textit{- end note}]

2 \textit{Throws}: nothing.

20.9.11.3 \textbf{Class template weak\_ptr} \hfill [util.smartptr.weak]

The $\text{weak\_ptr}$ class template stores a weak reference to an object that is already managed by a $\text{shared\_ptr}$. To access the object, a $\text{weak\_ptr}$ can be converted to a $\text{shared\_ptr}$ using the member function $\text{lock}$.

namespace std {
    template<class T> class weak_ptr {
        public:
            typedef T element_type;

            // constructors
            constexpr 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);
    }
}

§ 20.9.11.3  585
Specializations of `weak_ptr` shall be CopyConstructible and CopyAssignable, allowing their use in standard containers. The template parameter `T` of `weak_ptr` may be an incomplete type.

### 20.9.11.3.1 weak_ptr constructors

```cpp
constexpr weak_ptr();
1
   **Effects:** Constructs an *empty* `weak_ptr` object.
2
   **Postconditions:** `use_count() == 0`.
3
   **Throws:** nothing.

weak_ptr(const weak_ptr& r);
weak_ptr(const weak_ptr<Y>& r);
weak_ptr(const shared_ptr<Y>& r);
4
   **Requires:** 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.
```

### 20.9.11.3.2 weak_ptr destructor

```cpp
~weak_ptr();
1
   **Effects:** Destroys this `weak_ptr` object but has no effect on the object its stored pointer points to.
```

§ 20.9.11.3.2
**20.9.11.3.3 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.

**20.9.11.3.4 weak_ptr modifiers**

```
void swap(weak_ptr& r);
void reset();
```

1. **Effects:** Exchanges the contents of `*this` and `r`.
2. **Throws:** nothing.
3. **Effects:** Equivalent to `weak_ptr().swap(*this)`.

**20.9.11.3.5 weak_ptr observers**

```
long use_count() const;
bool expired() const;
```

1. **Returns:** 0 if `*this` is empty; otherwise, the number of `shared_ptr` instances that share ownership with `*this`.
2. **Throws:** nothing.
3. **Note:** `use_count()` is not necessarily efficient. — end note
4. **Returns:** `use_count() == 0`.
5. **Throws:** nothing.
6. **Note:** `expired()` may be faster than `use_count()`. — end note

```
shared_ptr<T> lock() const;
```

1. **Returns:** `expired() ? shared_ptr<T>() : shared_ptr<T>(*this)`.
2. **Throws:** nothing.

```
template<class U> bool owner_before(shared_ptr<U> const& b);
template<class U> bool owner_before(weak_ptr<U> const& b);
```

1. **Returns:** an unspecified value such that
   - `x.owner_before(y)` defines a strict weak ordering as defined in 25.4;

§ 20.9.11.3.5
under the equivalence relation defined by owner_before, !a.owner_before(b) && !b.owner_before(a), two shared_ptr or weak_ptr instances are equivalent if and only if they share ownership or are both empty.

20.9.11.3.6 weak_ptr specialized algorithms

```cpp
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b)
```

Effects: Equivalent to a.swap(b).

Throws: nothing.

20.9.11.3.7 Class template owner_less

The class template owner_less allows ownership-based mixed comparisons of shared and weak pointers.

```cpp
namespace std {
    template<class T> struct owner_less;
    template<class T> struct owner_less<shared_ptr<T> >
        : binary_function<shared_ptr<T>, shared_ptr<T>, bool> {
            typedef bool result_type;
            bool operator()(shared_ptr<T> const&, shared_ptr<T> const&) const;
            bool operator()(shared_ptr<T> const&, weak_ptr<T> const&) const;
            bool operator()(weak_ptr<T> const&, shared_ptr<T> const&) const;
        };
    template<class T> struct owner_less<weak_ptr<T> >
        : binary_function<weak_ptr<T>, weak_ptr<T>, bool> {
            typedef bool result_type;
            bool operator()(weak_ptr<T> const&, weak_ptr<T> const&) const;
            bool operator()(shared_ptr<T> const&, weak_ptr<T> const&) const;
            bool operator()(weak_ptr<T> const&, shared_ptr<T> const&) const;
        };
}
```

operator()(x,y) shall return x.owner_before(y). [Note: Note that

— operator() defines a strict weak ordering as defined in 25.4;

— under the equivalence relation defined by operator(), !operator()(a, b) && !operator()(b, a),
  two shared_ptr or weak_ptr instances are equivalent if and only if they share ownership or are both empty.

20.9.11.4 Class template enable_shared_from_this

A class T can inherit from enable_shared_from_this<T> to inherit the shared_from_this member functions that obtain a shared_ptr instance pointing to *this.

```cpp
struct X: public enable_shared_from_this<X> {
};
```
```cpp
int main() {
    shared_ptr<X> p(new X);
    shared_ptr<X> q = p->shared_from_this();
    assert(p == q);
    assert(!(p < q) && !(q < p)); // p and q share ownership
}
```

--- end example

namespace std {
    template<class T> class enable_shared_from_this {
    protected:
        constexpr enable_shared_from_this();
        enable_shared_from_this(enable_shared_from_this const&);
        enable_shared_from_this& operator=(enable_shared_from_this const&);
    public:
        shared_ptr<T> shared_from_this();
        shared_ptr<T const> shared_from_this() const;
    }; // namespace std

The template parameter T of `enable_shared_from_this` may be an incomplete type.

```cpp
constexpr 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.

```cpp
enable_shared_from_this<T>& operator=(const enable_shared_from_this<T>&);
```

**Returns:** *this.

**Throws:** nothing.

```cpp
~enable_shared_from_this();
```

**Effects:** Destroys *this.

**Throws:** nothing.

```cpp
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:

```cpp
template<class T> class enable_shared_from_this {
    private:
        weak_ptr<T> __weak_this;
    protected:
        constexpr enable_shared_from_this() : __weak_this() { }
    };
```
enable_shared_from_this(enable_shared_from_this const &) { }

enable_shared_from_this& operator=(enable_shared_from_this const &) { return *this; }

"enable_shared_from_this() { }

public:

shared_ptr<T> shared_from_this() { return shared_ptr<T>(_weak_this); }

shared_ptr<T const> shared_from_this() const { return shared_ptr<T const>(_weak_this); }

};

The shared_ptr constructors that create unique pointers can detect the presence of an enable_shared_from_this base and assign the newly created shared_ptr to its __weak_this member. — end note

20.9.11.5 shared_ptr atomic access [util.smartptr.shared.atomic]

Concurrent access to a shared_ptr object from multiple threads does not introduce a data race if the access is done exclusively via the functions in this section and the instance is passed as their first argument.

The meaning of the arguments of type memory_order is explained in 29.3.

template<class T>
bool atomic_is_lock_free(const shared_ptr<T>* p);

Requires: p shall not be null.

Returns: true if atomic access to *p is lock-free, false otherwise.

Throws: nothing.

template<class T>
shared_ptr<T> atomic_load(const shared_ptr<T>* p);

Requires: p shall not be null.

Returns: atomic_load_explicit(p, memory_order_seq_cst).

template<class T>
shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo);

Requires: p shall not be null.

Requires: mo shall not be memory_order_release or memory_order_acq_rel.

Returns: *p.

Throws: nothing.

template<class T>
void atomic_store(shared_ptr<T>* p, shared_ptr<T> r);

Requires: p shall not be null.

Effects: atomic_store_explicit(p, r, memory_order_seq_cst).

template<class T>
void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);

Requires: p shall not be null.

Requires: mo shall not be memory_order_acquire or memory_order_acq_rel.

Effects: p->swap(r).
Throws: nothing.

```cpp
template<class T>
shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r);
```

Requires: p shall not be null.

Returns: atomic_exchange_explicit(p, r, memory_order_seq_cst).

```cpp
template<class T>
shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r,
                                       memory_order mo);
```

Requires: p shall not be null.

Effects: p->swap(r).

Returns: the previous value of *p.

Throws: nothing.

```
template<class T>
bool atomic_compare_exchange_weak(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
```

Requires: p shall not be null.

Returns: atomic_compare_exchange_weak_explicit(p, v, w, memory_order_seq_cst, memory_order_seq_cst).

```
template<class T>
bool atomic_compare_exchange_strong(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
```

Returns: atomic_compare_exchange_strong_explicit(p, v, w, memory_order_seq_cst, memory_order_seq_cst).

```
template<class T>
bool atomic_compare_exchange_weak_explicit(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
    memory_order success, memory_order failure);
```

```
template<class T>
bool atomic_compare_exchange_strong_explicit(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
    memory_order success, memory_order failure);
```

Requires: p shall not be null.

Requires: failure shall not be memory_order_release, memory_order_acq_rel, or stronger than success.

Effects: If *p is equivalent to *v, assigns w to *p and has synchronization semantics corresponding to the value of success, otherwise assigns *p to *v and has synchronization semantics corresponding to the value of failure.

Returns: true if *p was equivalent to *v, false otherwise.

Throws: nothing.

Remarks: two shared_ptr objects are equivalent if they store the same pointer value and share ownership.
Remarks: the weak forms may fail spuriously. See 29.6.

20.9.11.6 Hash support

\[\text{util.smartptr.hash}\]

\[
\text{template } \langle\text{class } T, \text{ class } D\rangle \text{ struct hash<\text{unique_ptr<}T, \text{ D}> ;}
\]

1 Requires: The template specialization shall meet the requirements of class template hash (20.8.15). For an object \(p\) of type UP, where UP is unique_ptr\(<T, D>\), hash<UP>()(p) shall evaluate to the same value as hash<typename UP::pointer>()(p.get()). The specialization hash<typename UP::pointer> shall be well-formed.

\[
\text{template } \langle\text{class } T\rangle \text{ struct hash<\text{shared_ptr<}T\rangle ;}
\]

2 Requires: The template specialization shall meet the requirements of class template hash (20.8.15). For an object \(p\) of type shared_ptr\(<T>\), hash<shared_ptr<T> >()(p) shall evaluate to the same value as hash<T*>()(p.get()).

20.9.12 Pointer safety

\[\text{util.dynamic.safety}\]

1 A complete object is declared reachable while the number of calls to declare_reachable with an argument referencing the object exceeds the number of calls to undeclare_reachable with an argument referencing the object.

\[
\text{void declare_reachable(}\text{void *p);}
\]

2 Requires: \(p\) shall be a safely-derived pointer (3.7.4.3) or a null pointer value.

3 Effects: If \(p\) is not null, the complete object referenced by \(p\) is subsequently declared reachable (3.7.4.3).

4 Throws: May throw std::bad_alloc if the system cannot allocate additional memory that may be required to track objects declared reachable.

\[
\text{template } \langle\text{class } T\rangle \text{ T *undeclare_reachable(}T \text{ *p);}
\]

5 Requires: If \(p\) is not null, the complete object referenced by \(p\) shall have been previously declared reachable, and shall be live (3.8) from the time of the call until the last undeclare_reachable(p) call on the object.

6 Returns: a safely derived copy of \(p\) which shall compare equal to \(p\).

7 Throws: nothing.

8 \[\text{Note: It is expected that calls to declare_reachable(p) will consume a small amount of memory in addition to that occupied by the referenced object until the matching call to undeclare_reachable(p) is encountered. Long running programs should arrange that calls are matched. — end note}\]

\[
\text{void declare_no_pointers(}\text{char *p, size_t n);}
\]

! Requires: No bytes in the specified range have been previously registered with declare_no_pointers(). If the specified range is in an allocated object, then it must be entirely within a single allocated object. The object must be live until the corresponding undeclare_no_pointers() call. \[\text{Note: In a garbage-collecting implementation, the fact that a region in an object is registered with declare_no_pointers() should not prevent the object from being collected. — end note}\]

9 Effects: The \(n\) bytes starting at \(p\) no longer contain traceable pointer locations, independent of their type. Hence pointers located there may not be dereferenced if the object they point to was created

\section{20.9.12}
by global operator new and not previously declared reachable. [Note: This may be used to inform a
garbage collector or leak detector that this region of memory need not be traced. — end note]

Throws: nothing. [Note: Under some conditions implementations may need to allocate memory.
However, the request can be ignored if memory allocation fails. — end note]

```c
void undeclare_no_pointers(char *p, size_t n);
```

Requires: The same range must previously have been passed to declare_no_pointers().

Effects: Unregisters a range registered with declare_no_pointers() for destruction. It must be called
before the lifetime of the object ends.

Throws: nothing.

```c
pointer_safety get_pointer_safety();
```

Returns: `pointer_safety::strict` if the implementation has strict pointer safety (3.7.4.3). It is im-
plementation defined whether get_pointer_safety returns `pointer_safety::relaxed` or `pointer_-
safety::preferred` if the implementation has relaxed pointer safety.

Throws: Nothing.

### 20.9.13 Align [ptr.align]

```c
void *align(std::size_t alignment, std::size_t size, void *&ptr, std::size_t& space);
```

Effects: If it is possible to fit `size` bytes of storage aligned by `alignment` into the buffer pointed to by
`ptr` with length `space`, the function updates `ptr` to point to the first possible address of such storage
and decreases `space` by the number of bytes used for alignment. Otherwise, the function does nothing.

Requires:

— `alignment` shall be a fundamental alignment value or an extended alignment value supported by
the implementation in this context

— `ptr` shall point to contiguous storage of at least `space` bytes

Returns: a null pointer if the requested aligned buffer would not fit into the available space, otherwise
the adjusted value of `ptr`.

[Note: the function updates its `ptr` and `space` arguments so that it can be called repeatedly with
possibly different `alignment` and `size` arguments for the same buffer.]

### 20.9.14 C Library [c.malloc]

Table 54 describes the header `<cstdlib>`.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functions</strong></td>
<td><code>calloc</code> <code>malloc</code></td>
</tr>
<tr>
<td></td>
<td><code>free</code> <code>realloc</code></td>
</tr>
</tbody>
</table>

235) `pointer_safety::preferred` might be returned to indicate that a leak detector is running so that
the program can avoid spurious leak reports.
The contents are the same as the Standard C library header `<stdlib.h>`, with the following changes:

The functions `calloc()`, `malloc()`, and `realloc()` do not attempt to allocate storage by calling `::operator new()` (18.6).

The function `free()` does not attempt to deallocate storage by calling `::operator delete()`.

See also: ISO C Clause 7.11.2.

Storage allocated directly with `malloc()`, `calloc()`, or `realloc()` is implicitly declared reachable (see 3.7.4.3) on allocation, ceases to be declared reachable on deallocation, and need not cease to be declared reachable as the result of an `undeclare_reachable()` call. [Note: This allows existing C libraries to remain unaffected by restrictions on pointers that are not safely derived, at the expense of providing far fewer garbage collection and leak detection options for `malloc()`-allocated objects. It also allows `malloc()` to be implemented with a separate allocation arena, bypassing the normal `declare_reachable()` implementation. The above functions should never intentionally be used as a replacement for `declare_reachable()`, and newly written code is strongly encouraged to treat memory allocated with these functions as though it were allocated with `operator new`. — end note]

Table 55 describes the header `<cstring>`.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro:</td>
<td>NULL</td>
</tr>
<tr>
<td>Type:</td>
<td>size_t</td>
</tr>
<tr>
<td>Functions:</td>
<td>memchr memcmp memcpy memmove memset</td>
</tr>
</tbody>
</table>

The contents are the same as the Standard C library header `<string.h>`, with the change to `memchr()` specified in 21.7.

See also: ISO C Clause 7.11.2.

20.10 Time utilities

This subclause describes the chrono library that provides generally useful time utilities.

Header `<chrono>` synopsis

```cpp
namespace std {
    namespace chrono {

        template <class Rep, class Period = ratio<1>> class duration;
        template <class Clock, class Duration = typename Clock::duration> class time_point;

    } // namespace chrono

    // common_type traits
    template <class Rep1, class Period1, class Rep2, class Period2>
    struct common_type<chrono::duration<Rep1, Period1>, chrono::duration<Rep2, Period2>>;

    template <class Clock, class Duration1, class Duration2>
    struct common_type<chrono::time_point<Clock, Duration1>, chrono::time_point<Clock, Duration2>>;

    namespace chrono {
```

§ 20.10 594
// customization traits
template <class Rep> struct treat_as_floating_point;
template <class Rep> struct duration_values;

// duration arithmetic
template <class Rep1, class Period1, class Rep2, class Period2>
type common_type<duration<Rep1, Period1>, duration<Rep2, Period2>>::type
operator+(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
type common_type<duration<Rep1, Period1>, duration<Rep2, Period2>>::type
operator-(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
type duration<typename common_type<Rep1, Rep2>::type, Period>
operator*(const duration<Rep1, Period1>& d, const Rep2& s);
type duration<typename common_type<Rep1, Rep2>::type, Period>
operator*(const Rep1& s, const duration<Rep2, Period1>& d);
type duration<typename common_type<Rep1, Rep2>::type, Period>
operator/(const duration<Rep1, Period1>& d, const Rep2& s);
type common_type<duration<Rep1, Period1>, duration<Rep2, Period2>>::type
operator%(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

// duration comparisons
template <class Rep1, class Period1, class Rep2, class Period2>
bool operator==(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
bool operator!=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
bool operator<( const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
bool operator<=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
bool operator>( const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
bool operator>=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

// duration_cast
template <class ToDuration, class Rep, class Period>
ToDuration duration_cast(const duration<Rep, Period>& d);

// convenience typedefs
typedef duration<signed integral type of at least 64 bits, nano> nanoseconds;
typedef duration<signed integral type of at least 55 bits, micro> microseconds;
typedef duration<signed integral type of at least 45 bits, milli> milliseconds;
typedef duration<signed integral type of at least 35 bits, > seconds;
typedef duration<signed integral type of at least 29 bits, ratio< 60>> minutes;
typedef duration<signed integral type of at least 23 bits, ratio<3600>> hours;

// time_point arithmetic
template <class Clock, class Duration1, class Rep2, class Period2>
  time_point<Clock, typename common_type<Duration1, duration<Rep2, Period2>>::type>
  operator+(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Clock, class Duration2>
  time_point<Clock, typename common_type<duration<Rep1, Period1>, Duration2>::type>
  operator+(const duration<Rep1, Period1>& lhs, const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Rep2, class Period2>
  time_point<Clock, typename common_type<Duration1, duration<Rep2, Period2>>::type>
  operator-(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Clock, class Duration1, class Duration2>
  typename common_type<Duration1, Duration2>::type
  operator-(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

// time_point comparisons
template <class Clock, class Duration1, class Duration2>
  bool operator==(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
  bool operator!=(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
  bool operator< (const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
  bool operator>(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

// time_point_cast
template <class ToDuration, class Clock, class Duration>
  time_point<Clock, ToDuration> time_point_cast(const time_point<Clock, Duration>& t);

// Clocks
class system_clock;
class monotonic_clock;
class high_resolution_clock;
}

namespace std {
} // namespace std

20.10.1 Clock requirements [time.clock.req]

A clock is a bundle consisting of a native duration, a native time_point, and a function now() to get the current time_point. The origin of the clock’s time_point is referred to as the clock’s epoch. A clock shall meet the requirements in Table 56.

In Table 56 C1 and C2 denote clock types. t1 and t2 are values returned by C1::now() where the call returning t1 happens before (1.10) the call returning t2 and both of these calls happen before C1::time_point::max().
### Table 56 — Clock requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>C1::rep</code></td>
<td>An arithmetic type or a class</td>
<td>The representation type of the native duration and time_point.</td>
</tr>
<tr>
<td></td>
<td>emulating an arithmetic type</td>
<td></td>
</tr>
<tr>
<td><code>C1::period</code></td>
<td>a specialization of <code>ratio</code></td>
<td>The tick period of the clock in seconds.</td>
</tr>
<tr>
<td><code>C1::duration</code></td>
<td><code>chrono::duration&lt;C1::rep, C1::period&gt;</code></td>
<td>The native duration type of the clock.</td>
</tr>
<tr>
<td><code>C1::time_point</code></td>
<td><code>chrono::time_point&lt;C1&gt;</code> or <code>chrono::time_point&lt;C2, C1::duration&gt;</code></td>
<td>The native time_point type of the clock. C1 and C2 shall refer to the same epoch.</td>
</tr>
<tr>
<td><code>C1::is_monotonic</code></td>
<td><code>const bool</code></td>
<td><code>true</code> if <code>t1 &lt;= t2</code> is always true, otherwise <code>false</code>. [Note: A clock that can be adjusted backwards is not monotonic. — end note]</td>
</tr>
<tr>
<td><code>C1::now()</code></td>
<td><code>C1::time_point</code></td>
<td>Returns a <code>time_point</code> object representing the current point in time.</td>
</tr>
</tbody>
</table>

#### 20.10.2 Time-related traits

**20.10.2.1 treat_as_floating_point**

```cpp
template <class Rep> struct treat_as_floating_point :
    is_floating_point<Rep> {}
;
```

The `duration` template uses the `treat_as_floating_point` trait to help determine if a `duration` object can be converted to another `duration` with a different tick period. If `treat_as_floating_point<Rep>::value` is true, then `Rep` is a floating-point type and implicit conversions are allowed among `duration`s. Otherwise, the implicit convertibility depends on the tick periods of the `duration`s. If `Rep` is a class type which emulates a floating-point type, the author of `Rep` can specialize `treat_as_floating_point` so that `duration` will treat this `Rep` as if it were a floating-point type. Otherwise `Rep` is assumed to be an integral type or a class emulating an integral type.

**20.10.2.2 duration_values**

```cpp
template <class Rep>
struct duration_values {
    public:
        static constexpr Rep zero();
        static constexpr Rep min();
        static constexpr Rep max();
};
```

The `duration` template uses the `duration_values` trait to construct special values of the `duration` representation (`Rep`). This is done because the representation might be a class type with behavior which requires some other implementation to return these special values. In that case, the author of that class type should specialize `duration_values` to return the indicated values.

§ 20.10.2.2
static constexpr Rep zero();

Returns: Rep(0). [Note: Rep(0) is specified instead of Rep() because Rep() may have some other meaning, such as an uninitialized value. —end note]

Remark: The value returned shall be the additive identity.

static constexpr Rep min();

Returns: numeric_limits<Rep>::lowest().

Remark: The value returned shall compare less than or equal to zero().

static constexpr Rep max();

Returns: numeric_limits<Rep>::max().

Remark: The value returned shall compare greater than zero().

20.10.2.3 Specializations of common_type

template <class Rep1, class Period1, class Rep2, class Period2>
struct common_type<chrono::duration<Rep1, Period1>, chrono::duration<Rep2, Period2>> {
    typedef chrono::duration<typename common_type<Rep1, Rep2>::type, see below> type;
};

The period of the duration indicated by this specialization of common_type shall be the greatest common divisor of Period1 and Period2. [Note: This can be computed by forming a ratio of the greatest common divisor of Period1::num and Period2::num and the least common multiple of Period1::den and Period2::den. —end note]

[Note: The typedef name type is a synonym for the duration with the largest tick period possible where both duration arguments will convert to it without requiring a division operation. The representation of this type is intended to be able to hold any value resulting from this conversion with no truncation error, although floating-point durations may have round-off errors. —end note]

template <class Clock, class Duration1, class Duration2>
struct common_type<chrono::time_point<Clock, Duration1>, chrono::time_point<Clock, Duration2>> {
    typedef chrono::time_point<Clock, typename common_type<Duration1, Duration2>::type> type;
};

The common type of two time_point types is a time_point with the same clock as the two types and the common type of their two durations.

20.10.3 Class template duration

dauration type measures time between two points in time (time_points). A duration has a representation which holds a count of ticks and a tick period. The tick period is the amount of time which occurs from one tick to the next, in units of seconds. It is expressed as a rational constant using the template ratio.

template <class Rep, class Period = ratio<1>>
class duration {
    public:
        typedef Rep rep;
        typedef Period period;
    private:
        rep rep_; // exposition only
    public:
// 20.10.3.1, construct/copy/destroy:
constexpr duration() = default;
template <class Rep2>
    constexpr explicit duration(const Rep2& r);
template <class Rep2, class Period2>
    constexpr duration(const duration<Rep2, Period2>& d);
~duration() = default;
duration(const duration&) = default;
duration& operator=(const duration&) = default;

// 20.10.3.2, observer:
constexpr rep count() const;

// 20.10.3.3, arithmetic:
constexpr duration operator+() const;
constexpr duration operator-() const;
duration& operator++();
duration operator++(int);
duration& operator--();
duration operator--(int);

duration& operator+=(const duration& d);
duration& operator-=(const duration& d);

duration& operator*=(const rep& rhs);
duration& operator/=(const rep& rhs);
duration& operator%=(const rep& rhs);
duration& operator%=(const duration& rhs);

// 20.10.3.4, special values:
static constexpr duration zero();
static constexpr duration min();
static constexpr duration max();
};

Requires: Rep shall be an arithmetic type or a class emulating an arithmetic type.
Remarks: If duration is instantiated with a duration type for the template argument Rep, the program is ill-formed.
Remarks: If Period is not a specialization of ratio, the program is ill-formed.
Remarks: If Period::num is not positive, the program is ill-formed.
Requires: Members of duration shall not throw exceptions other than those thrown by the indicated operations on their representations.

[Example:
duration<long, ratio<60>> d0; // holds a count of minutes using a long
duration<long long, milli> d1; // holds a count of milliseconds using a long long
duration<double, ratio<1, 30>> d2; // holds a count with a tick period of \frac{1}{30} of a second
    // (30 Hz) using a double
— end example]
20.10.3.1  duration constructors

```cpp
template <class Rep2>
constexpr explicit duration(const Rep2& r);
```

1  **Remarks:** This constructor shall not participate in overload resolution unless Rep2 is implicitly convertible to rep and
   - treat_asFloatingPoint<rep>::value is true or
   - treat_asFloatingPoint<Rep2>::value is false.

   ```cpp
duration<int, milli> d(3);  // OK
duration<int, milli> d(3.5); // error
```

2  **Effects:** Constructs an object of type duration.

   ```cpp
template <class Rep2, class Period2>
constexpr duration(const duration<Rep2, Period2>& d);
```

4  **Remarks:** This constructor shall not participate in overload resolution unless treat_asFloatingPoint<rep>::value is true or both ratio_divide<Period2, period>::den is 1 and treat_asFloatingPoint<Rep2>::value is false. [Note: This requirement prevents implicit truncation error when converting between integral-based duration types. Such a construction could easily lead to confusion about the value of the duration. — end note]  

   ```cpp
duration<int, milli> ms(3);
duration<int, micro> us = ms;  // OK
duration<int, milli> ms2 = us;  // error
```

5  **Effects:** Constructs an object of type duration, constructing rep_ from duration_cast<duration>(d).count().

20.10.3.2  duration observer

```cpp
constexpr rep count() const;
```

1  **Returns:** rep_

20.10.3.3  duration arithmetic

```cpp
constexpr duration operator+() const;
```

1  **Returns:** *this.

```cpp
constexpr duration operator-() const;
```

2  **Returns:** duration(-rep_);

```cpp
duration& operator++();
```
Effects: ++rep_.
Returns: *this.

duration operator++(int);
Returns: duration(rep_++);

duration& operator--();
Effects: --rep_.
Returns: *this.

duration operator--(int);
Returns: duration(rep_--);

duration& operator+=(const duration& d);
Effects: rep_ += d.count().
Returns: *this.

duration& operator-=(const duration& d);
Effects: rep_ -= d.count().
Returns: *this.

duration& operator*=(const rep& rhs);
Effects: rep_ *= rhs.
Returns: *this.

duration& operator/=(const rep& rhs);
Effects: rep_ /= rhs.
Returns: *this.

duration& operator%=(const rep& rhs);
Effects: rep_ %= rhs.
Returns: *this.

duration& operator%=(const duration& rhs);
Effects: rep_ %= rhs.count().
Returns: *this.

20.10.3.4 duration special values

static constexpr duration zero();
Returns: duration(duration_values<rep>::zero());

static constexpr duration min();
Returns: duration(duration_values<rep>::min());

§ 20.10.3.4
static constexpr duration max();

Returns: duration(duration_values<rep>::max()).

20.10.3.5 duration non-member arithmetic

template <class Rep1, class Period1, class Rep2, class Period2>
type common_type<duration<Rep1, Period1>, duration<Rep2, Period2>::type
operator+(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: CD(lhs) += rhs.

template <class Rep1, class Period1, class Rep2, class Period2>
type common_type<duration<Rep1, Period1>, duration<Rep2, Period2>::type
operator-(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: CD(lhs) -= rhs.

template <class Rep1, class Period, class Rep2>
duration<typename common_type<Rep1, Rep2>::type, Period>
operator*(const duration<Rep1, Period>& d, const Rep2& s);

Remarks: This operator shall not participate in overload resolution unless Rep2 is implicitly convertible
to CR(Rep1, Rep2).

Returns: duration<CR(Rep1, Rep2), Period>(d) *= s.

template <class Rep1, class Period, class Rep2>
duration<typename common_type<Rep1, Rep2>::type, Period>
operator%(const duration<Rep1, Period>& d, const Rep2& s);

Remarks: This operator shall not participate in overload resolution unless Rep2 is implicitly convertible
to CR(Rep1, Rep2).

Returns: d * s.

template <class Rep1, class Period1, class Rep2, class Period2>
type common_type<duration<Rep1, Period1>, duration<Rep2, Period2>::type
operator/(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: CD(lhs).count() / CD(rhs).count().

Remarks: This operator shall not participate in overload resolution unless Rep2 is implicitly convertible
to CR(Rep1, Rep2) and Rep2 is not an instantiation of duration.

§ 20.10.3.5
template <class Rep1, class Period1, class Rep2, class Period2>
    typename common_type<duration<Rep1, Period1>, duration<Rep2, Period2>>::type
    operator%(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

    Returns: common_type<duration<Rep1, Period1>, duration<Rep2, Period2>> ::type(lhs) %= rhs.

20.10.3.6 duration comparisons

In the function descriptions that follow, CT represents common_type<A, B>::type, where A and B are the types of the two arguments to the function.

    template <class Rep1, class Period1, class Rep2, class Period2>
    bool operator==(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

    Returns: CT(lhs).count() == CT(rhs).count().

    template <class Rep1, class Period1, class Rep2, class Period2>
    bool operator!=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

    Returns: !(lhs == rhs).

    template <class Rep1, class Period1, class Rep2, class Period2>
    bool operator<(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

    Returns: CT(lhs).count() < CT(rhs).count().

    template <class Rep1, class Period1, class Rep2, class Period2>
    bool operator>(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

    Returns: rhs < lhs.

20.10.3.7 duration_cast

    template <class ToDuration, class Rep, class Period>
    ToDuration duration_cast(const duration<Rep, Period>& d);

    Remarks: This function shall not participate in overload resolution unless ToDuration is an instantiation of duration.

    Returns: Let CF be ratio_divide<Period, typename ToDuration::period>, and CR be common_type<
type name ToDuration::rep, Rep, intmax_t>::type.

        — If CF::num == 1 and CF::den == 1, returns
            ToDuration(static_cast<typename ToDuration::rep>(d.count()))
— otherwise, if \( \text{CF}::\text{num} \neq 1 \) and \( \text{CF}::\text{den} = 1 \), returns

\[
\text{ToDuration}(\text{static\_cast<typename ToDuration::rep>(}
\text{static\_cast<CR>(d.count())} \times \text{static\_cast<CR>(CF::num)}))
\]

— otherwise, if \( \text{CF}::\text{num} = 1 \) and \( \text{CF}::\text{den} \neq 1 \), returns

\[
\text{ToDuration}(\text{static\_cast<typename ToDuration::rep>(}
\text{static\_cast<CR>(d.count())} / \text{static\_cast<CR>(CF::den)}))
\]

— otherwise, returns

\[
\text{ToDuration}(\text{static\_cast<typename ToDuration::rep>(}
\text{static\_cast<CR>(d.count())} \times \text{static\_cast<CR>(CF::num)} / \text{static\_cast<CR>(CF::den)}))
\]

Notes: This function does not use any implicit conversions; all conversions are done with \text{static\_cast}. It avoids multiplications and divisions when it is known at compile time that one or more arguments is 1. Intermediate computations are carried out in the widest representation and only converted to the destination representation at the final step.

20.10.4 Class template \text{time\_point}  

\begin{verbatim}
template <class Clock, class Duration = typename Clock::duration>
class time_point {
public:
  typedef Clock clock;
  typedef Duration duration;
  typedef typename duration::rep rep;
  typedef typename duration::period period;
private:
  duration d_;   // exposition only

public:
  // 20.10.4.1, construct
  time_point();   // has value epoch
  explicit time_point(const duration& d); // same as time_point() + d
  template <class Duration2>
  time_point(const time_point<Clock, Duration2>& t);

  // 20.10.4.2, observer:
  duration time_since_epoch() const;

  // 20.10.4.3, arithmetic:
  time_point& operator+=(const duration& d);
  time_point& operator-=(const duration& d);

  // 20.10.4.4, special values:
  static constexpr time_point min();
  static constexpr time_point max();
};
\end{verbatim}

Clock shall meet the Clock requirements (20.10.5).

2 If \text{Duration} is not an instance of \text{duration}, the program is ill-formed.

20.10.4.1 time_point constructors
time_point();

1    Effects: Constructs an object of type time_point, initializing \( d_0 \) with duration::zero(). Such a time_point object represents the epoch.

time_point(const duration& d);

2    Effects: Constructs an object of type time_point, initializing \( d_0 \) with \( d \). Such a time_point object represents the epoch + \( d \).

template <class Duration2>
    time_point(const time_point<clock, Duration2>& t);

3    Remarks: This constructor shall not participate in overload resolution unless Duration2 is implicitly convertible to duration.

4    Effects: Constructs an object of type time_point, initializing \( d_0 \) with \( t.time_since_epoch() \).

20.10.4.2     time_point observer

duration time_since_epoch() const;

1    Returns: \( d_0 \).

20.10.4.3     time_point arithmetic

    time_point& operator+=(const duration& d);

1     Effects: \( d_0 += d \).

2     Returns: *this.

time_point& operator-=(const duration& d);

3     Effects: \( d_0 -= d \).

4     Returns: *this.

20.10.4.4     time_point special values

static constexpr time_point min();

1    Returns: time_point(duration::min()).

static constexpr time_point max();

2    Returns: time_point(duration::max()).

20.10.4.5     time_point non-member arithmetic

template <class Clock, class Duration1, class Rep2, class Period2>
    time_point<Clock, typename common_type<Duration1, duration<Rep2, Period2>>::type>
        operator+(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);

1    Returns: CT(lhs) += rhs, where CT is the type of the return value.
template <class Rep1, class Period1, class Clock, class Duration2>
  time_point<Clock, typename common_type<duration<Rep1, Period1>, Duration2>::type>
  operator+(const duration<Rep1, Period1>& lhs, const time_point<Clock, Duration2>& rhs);

   Returns: rhs + lhs.

template <class Clock, class Duration1, class Rep2, class Period2>
  time_point<Clock, typename common_type<Duration1, duration<Rep2, Period2>>::type>
  operator-(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);

   Returns: lhs + (-rhs).

template <class Clock, class Duration1, class Duration2>
  typename common_type<Duration1, Duration2>::type
  operator-(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

   Returns: lhs.time_since_epoch() - rhs.time_since_epoch().

20.10.4.6 time_point comparisons  [time.point.comparisons]

template <class Clock, class Duration1, class Duration2>
  bool operator==(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

   Returns: lhs.time_since_epoch() == rhs.time_since_epoch().

template <class Clock, class Duration1, class Duration2>
  bool operator!=(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

   Returns: !(lhs == rhs).

template <class Clock, class Duration1, class Duration2>
  bool operator<(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

   Returns: lhs.time_since_epoch() < rhs.time_since_epoch().

template <class Clock, class Duration1, class Duration2>
  bool operator<=(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

   Returns: !(rhs < lhs).

template <class Clock, class Duration1, class Duration2>
  bool operator>(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

   Returns: rhs < lhs.

template <class Clock, class Duration1, class Duration2>
  bool operator>=(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

   Returns: !(lhs < rhs).

20.10.4.7 time_point_cast  [time.point.cast]

template <class ToDuration, class Clock, class Duration>
  time_point<Clock, ToDuration> time_point_cast(const time_point<Clock, Duration>& t);

   Remarks: This function shall not participate in overload resolution unless ToDuration is an instantiation of duration.

   Returns: time_point<Clock, ToDuration>(duration_cast<ToDuration>(t.time_since_epoch())).
20.10.5 Clocks

The types defined in this subclause shall satisfy the Clock requirements (20.10.1).

20.10.5.1 Class system_clock

Objects of class system_clock represent wall clock time from the system-wide realtime clock.

```cpp
class system_clock {
public:
    typedef unspecified rep;
    typedef ratio<unspecified, unspecified> period;
    typedef chrono::duration<rep, period> duration;
    typedef chrono::time_point<system_clock> time_point;
    static const bool is_monotonic = unspecified;

    static time_point now();

    // Map to C API
    static time_t to_time_t (const time_point& t);
    static time_point from_time_t(time_t t);
};
```

2 Requires: system_clock::duration::min() < system_clock::duration::zero() shall be true.

[Note: This implies that rep is a signed type. — end note]

```cpp
time_t to_time_t(const time_point& t);
```

3 Returns: A time_t object that represents the same point in time as t when both values are restricted to the coarser of the precisions of time_t and time_point. It is implementation defined whether values are rounded or truncated to the required precision.

```cpp
time_point from_time_t(time_t t);
```

4 Returns: A time_point object that represents the same point in time as t when both values are restricted to the coarser of the precisions of time_t and time_point. It is implementation defined whether values are rounded or truncated to the required precision.

20.10.5.2 Class monotonic_clock

Objects of class monotonic_clock represent clocks for which values of time_point never decrease as physical time advances. monotonic_clock may be a synonym for system_clock if system_clock::is_monotonic is true.

2 The class monotonic_clock is conditionally supported.

```cpp
class monotonic_clock {
public:
    typedef unspecified rep;
    typedef ratio<unspecified, unspecified> period;
    typedef chrono::duration<rep, period> duration;
    typedef chrono::time_point<unspecified, duration> time_point;
    static const bool is_monotonic = true;

    static time_point now();
};
```

§ 20.10.5.2
20.10.5.3 Class high_resolution_clock

Objects of class high_resolution_clock represent clocks with the shortest tick period. high_resolution_clock may be a synonym for system_clock or monotonic_clock.

class high_resolution_clock {
public:
    typedef unspecified rep;
    typedef ratio<unspecified, unspecified> period;
    typedef chrono::duration<rep, period> duration;
    typedef chrono::time_point<unspecified, duration> time_point;
    static const bool is_monotonic = unspecified;

    static time_point now();
};

20.11 Date and time functions

Table 57 describes the header <ctime>.

Table 57 — Header <ctime> synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros:</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>CLOCKS_PER_SEC</td>
</tr>
<tr>
<td>Types:</td>
<td>size_t</td>
</tr>
<tr>
<td></td>
<td>clock_t</td>
</tr>
<tr>
<td></td>
<td>time_t</td>
</tr>
<tr>
<td>Struct:</td>
<td>tm</td>
</tr>
<tr>
<td>Functions:</td>
<td>asctime</td>
</tr>
<tr>
<td></td>
<td>ctime</td>
</tr>
<tr>
<td></td>
<td>gmtime</td>
</tr>
<tr>
<td></td>
<td>mktime</td>
</tr>
<tr>
<td></td>
<td>time</td>
</tr>
</tbody>
</table>

The contents are the same as the Standard C library header <time.h>. The functions asctime, ctime, gmtime, and localtime are not required to avoid data races (17.6.4.8).

See also: ISO C Clause 7.12, Amendment 1 Clause 4.6.4.

20.12 Class type_index

20.12.1 Header <typeindex> synopsis

namespace std {
    class type_index;
    template <class T> struct hash;
    template<> struct hash<type_index>;
}

20.12.2 type_index overview

namespace std {
    class type_index {
    public:
        type_index(const type_info& rhs);
        bool operator==(const type_index& rhs) const;
    }

---

236) strftime supports the C conversion specifiers C, D, e, F, g, G, h, r, R, t, T, u, V, and z, and the modifiers E and 0.
The class `type_index` provides a simple wrapper for `type_info` which can be used as an index type in associative containers (23.4) and in unordered associative containers (23.5).

### 20.12.3 `type_index` members

#### `type_index(const type_info& rhs);`

**Effects:** constructs a `type_index` object, the equivalent of `target = &rhs`.

**Returns:**

1. `bool operator==(const type_index& rhs) const;`  
   **Returns:** `*target == *rhs.target`
2. `bool operator!=(const type_index& rhs) const;`  
   **Returns:** `*target != *rhs.target`
3. `bool operator<(const type_index& rhs) const;`  
   **Returns:** `target->before(*rhs.target)`
4. `bool operator<=(const type_index& rhs) const;`  
   **Returns:** `!rhs.target->before(*target)`
5. `bool operator>(const type_index& rhs) const;`  
   **Returns:** `rhs.target->before(*target)`
6. `bool operator>=(const type_index& rhs) const;`  
   **Returns:** `!target->before(*rhs.target)`
7. `size_t hash_code() const;`  
   **Returns:** `target->hash_code()`
8. `const char* name() const;`  
   **Returns:** `target->name()`

### 20.12.4 Hash support

```
template <> struct hash<type_index>;
```
Requires: the template specialization shall meet the requirements of class template `hash` (20.8.15). For an object `index` of type `type_index`, `hash<type_index>()(index)` shall evaluate to the same result as `index.hash_code()`.
21 Strings library

21.1 General

This Clause describes components for manipulating sequences of any non-array POD (3.9) type. In this Clause such types are called *char-like types*, and objects of char-like types are called *char-like objects* or simply *characters*.

The following subclauses describe a character traits class, a string class, and null-terminated sequence utilities, as summarized in Table 58.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.2 Character traits</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>21.3 String classes</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cwchar&gt;</td>
</tr>
<tr>
<td>21.7 Null-terminated sequence utilities</td>
<td>&lt;cstring&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cwctype&gt;</td>
</tr>
</tbody>
</table>

21.2 Character traits

This subclause defines requirements on classes representing *character traits*, and defines a class template *char_traits*[charT], along with 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 21.3 and 27 need a set of related types and functions to complete the definition of their semantics. These types and functions are provided as a set of member typedefs and functions in the template parameter ‘traits’ used by each such template. This subclause defines the semantics 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 four explicit specializations of it, *char_traits*[char], *char_traits*[char16_t], *char_traits*[char32_t], and *char_traits*[wchar_t], all of which appear in the header <string> and satisfy the requirements below.

21.2.1 Character traits requirements

In Table 59, *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 \texttt{X::pos	extunderscore type}; state denotes a value of type \texttt{X::state	extunderscore type}; and \texttt{r} denotes an lvalue of type \texttt{CharT}. Operations on Traits shall not throw exceptions.

Table 59 — Character traits requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{X::char	extunderscore type}</td>
<td>\texttt{charT}</td>
<td>(described in 21.2.2)</td>
<td>compile-time</td>
</tr>
<tr>
<td>\texttt{X::int	extunderscore type}</td>
<td>\texttt{X::int	extunderscore type}</td>
<td>(described in 21.2.2)</td>
<td>compile-time</td>
</tr>
<tr>
<td>\texttt{X::off	extunderscore type}</td>
<td>\texttt{X::off	extunderscore type}</td>
<td>(described in 21.2.2)</td>
<td>compile-time</td>
</tr>
<tr>
<td>\texttt{X::pos	extunderscore type}</td>
<td>\texttt{X::pos	extunderscore type}</td>
<td>(described in 21.2.2)</td>
<td>compile-time</td>
</tr>
<tr>
<td>\texttt{X::state	extunderscore type}</td>
<td>\texttt{X::state	extunderscore type}</td>
<td>(described in 21.2.2)</td>
<td>compile-time</td>
</tr>
<tr>
<td>\texttt{X::eq(c,d)}</td>
<td>\texttt{bool}</td>
<td>yields: whether (c) is to be treated as equal to (d).</td>
<td>constant</td>
</tr>
<tr>
<td>\texttt{X::lt(c,d)}</td>
<td>\texttt{bool}</td>
<td>yields: whether (c) is to be treated as less than (d).</td>
<td>constant</td>
</tr>
<tr>
<td>\texttt{X::compare(p,q,n)}</td>
<td>\texttt{int}</td>
<td>yields: 0 if for each (i) in ([0,n)), \texttt{X::eq(p[i],q[i])} is true; else, a negative value if, for some (j) in ([0,n]), \texttt{X::lt(p[j],q[j])} is true and for each (i) in ([0,j]) \texttt{X::eq(p[i],q[i])} is true; else a positive value.</td>
<td>linear</td>
</tr>
<tr>
<td>\texttt{X::length(p)}</td>
<td>\texttt{std::size	extunderscore t}</td>
<td>yields: the smallest (i) such that \texttt{X::eq(p[i],\texttt{charT}())} is true.</td>
<td>linear</td>
</tr>
<tr>
<td>\texttt{X::find(p,n,c)}</td>
<td>\texttt{const X::char	extunderscore type*}</td>
<td>yields: the smallest (q) in ([p,p+n)) such that \texttt{X::eq(*q,c)} is true, zero otherwise.</td>
<td>linear</td>
</tr>
<tr>
<td>\texttt{X::move(s,p,n)}</td>
<td>\texttt{X::char	extunderscore type*}</td>
<td>for each (i) in ([0,n)), performs \texttt{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>\texttt{X::copy(s,p,n)}</td>
<td>\texttt{X::char	extunderscore type*}</td>
<td>pre: (p) not in ([s,s+n)). yields: (s). for each (i) in ([0,n)), performs \texttt{X::assign(s[i],p[i])}.</td>
<td>linear</td>
</tr>
<tr>
<td>\texttt{X::assign(r,d)}</td>
<td>(not used)</td>
<td>assigns \texttt{r=d}.</td>
<td>constant</td>
</tr>
<tr>
<td>\texttt{X::assign(s,n,c)}</td>
<td>\texttt{X::char	extunderscore type*}</td>
<td>for each (i) in ([0,n)), performs \texttt{X::assign(s[i],c)}. yields: (s).</td>
<td>linear</td>
</tr>
<tr>
<td>\texttt{X::not	extunderscore eof(e)}</td>
<td>\texttt{int	extunderscore type}</td>
<td>yields: (e) if \texttt{X::eq	extunderscore int	extunderscore type(e,X::eof())} is false, otherwise a value (f) such that \texttt{X::eq	extunderscore int	extunderscore type(f,X::eof())} is false.</td>
<td>constant</td>
</tr>
<tr>
<td>\texttt{X::to	extunderscore char	extunderscore type(e)}</td>
<td>\texttt{X::char	extunderscore type}</td>
<td>yields: if for some (c), \texttt{X::eq	extunderscore int	extunderscore type(e,X::to	extunderscore int	extunderscore type(c))} is true, (c) else some unspecified value.</td>
<td>constant</td>
</tr>
</tbody>
</table>
Table 59 — Character traits requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::to_int_type(c)</td>
<td>X::int_type</td>
<td>yields: some value e, constrained by the definitions of to_char_type and eq_int_type.</td>
<td>constant</td>
</tr>
<tr>
<td>X::eq_int_type(e,f)</td>
<td>bool</td>
<td>yields: for all c and d, X::eq(c,d) is equal to X::eq_int_type(X::to_int_type(c), X::to_int_type(d)); otherwise, yields true if e and f are both copies of X::eof(); otherwise, yields false if one of e and f is a copy of X::eof() and the other is not; otherwise the value is unspecified.</td>
<td>constant</td>
</tr>
<tr>
<td>X::eof()</td>
<td>X::int_type</td>
<td>yields: a value e such that X::eq_int_type(e,X::to_int_type(c)) is false for all values c.</td>
<td>constant</td>
</tr>
</tbody>
</table>

2 The struct template

    template<class charT> struct char_traits;

shall be provided in the header `<string>` as a basis for explicit specializations.

21.2.2 traits typedefs

typedef CHAR_T char_type;

1 The type `char_type` is used to refer to the character container type in the implementation of the library classes defined in 21.3 and Clause 27.

typedef INT_T int_type;

2 Requires: For a certain character container type `char_type`, a related container type `INT_T` shall be a type or class which can represent all of the valid characters converted from the corresponding `char_`-type values, as well as an end-of-file value, `eof()`. The type `int_type` represents a character container type which can hold end-of-file to be used as a return type of the iostream class member functions.\(^\text{237}\)

typedef implementation-defined off_type;

typedef implementation-defined pos_type;

3 Requires: Requirements for `off_type` and `pos_type` are described in 27.2.2.

typedef STATE_T state_type;

4 Requires: `state_type` shall meet the requirements of CopyAssignable, CopyConstructible (35), and DefaultConstructible types.

\(^{237}\) If `eof()` can be held in `char_type` then some iostreams operations may give surprising results.
21.2.3 char_traits specializations

```cpp
namespace std {
    template<> struct char_traits<char> {
        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);
        static constexpr int_type not_eof(int_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 off_type eof();
    };
}
```

1. The header `<string>` shall define four specializations of the template struct `char_traits`: `char_traits<char>`, `char_traits<char16_t>`, `char_traits<char32_t>`, and `char_traits<wchar_t>`.

2. The requirements for the members of these specializations are given in Clause 21.2.1.

21.2.3.1 struct char_traits<char>

```cpp
namespace std {
    template<> struct char_traits<char> {
        typedef char char_type;
        typedef int int_type;
        typedef streamoff off_type;
        typedef streampos pos_type;
        typedef mbstate_t state_type;

        static void assign(char_type& c1, const char_type& c2);
        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(int_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 off_type eof();
    };
}
```

1. The defined types for `int_type`, `pos_type`, `off_type`, and `state_type` shall be `int`, `streampos`, `streamoff`, and `mbstate_t` respectively.

2. The type `streampos` shall be an implementation-defined type that satisfies the requirements for `pos_type` in 21.2.2.

3. The type `streamoff` shall be an implementation-defined type that satisfies the requirements for `off_type` in 21.2.2.

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.
The two-argument member `assign` shall be defined identically to the built-in operator `=`. The two-argument members `eq` and `lt` shall be defined identically to the built-in operators `==` and `<` for type `unsigned char`.

The member `eof()` shall return EOF.

21.2.3.2 `struct char_traits<char16_t>`

namespace std {
  template<> struct char_traits<char16_t> {
    typedef char16_t char_type;
    typedef uint_least16_t int_type;
    typedef streamoff off_type;
    typedef u16streampos pos_type;
    typedef mbstate_t state_type;

    static void assign(char_type& c1, const char_type& c2);
    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(int_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 `u16streampos` shall be an implementation-defined type that satisfies the requirements for `POS_T` in 21.2.2.

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.

21.2.3.3 `struct char_traits<char32_t>`

namespace std {
  template<> struct char_traits<char32_t> {
    typedef char32_t char_type;
    typedef uint_least32_t int_type;
    typedef streamoff off_type;
    typedef u32streampos pos_type;
    typedef mbstate_t state_type;

    static void assign(char_type& c1, const char_type& c2);
    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(int_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();
  };
}

§ 21.2.3.3
The type `u32streampos` shall be an implementation-defined type that satisfies the requirements for `POS_T` in 21.2.2.

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.

### 21.2.3.4 struct char_traits<wchar_t>  

```cpp
class char_traits
{
public:
    static int compare(const char_type* s1, const char_type* s2, size_t n);
    static size_t length(const char_type* s);
    static const char_type* find(const char_type* s, size_t n,
                                  const char_type& a);
    static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);

    static constexpr int_type not_eof(int_type c);
    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();
};
```

```cpp
namespace std {
    template<> struct char_traits<wchar_t> {
    public:
        typedef wchar_t char_type;
        typedef wint_t int_type;
        typedef streamoff off_type;
        typedef wstreampos pos_type;
        typedef mbstate_t state_type;

        static void assign(char_type& c1, const char_type& c2);
        static constexpr bool eq(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(int_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 \texttt{int\_type}, \texttt{pos\_type}, and \texttt{state\_type} shall be \texttt{wint\_t}, \texttt{wstreampos}, and \texttt{mbstate\_t} respectively.

The type \texttt{wstreampos} shall be an implementation-defined type that satisfies the requirements for \texttt{POS\_T} in 21.2.2.

The type \texttt{mbstate\_t} is defined in \texttt{<cwchar>} and can represent any of the conversion states that can occur in an implementation-defined set of supported multibyte character encoding rules.

The two-argument members \texttt{assign}, \texttt{eq}, and \texttt{lt} shall be defined identically to the built-in operators \texttt{=}, \texttt{==}, and \texttt{<} respectively.

The member \texttt{eof()} shall return \texttt{WEOF}.

21.3 String classes

The header \texttt{<string>} defines the \texttt{basic\_string} class template for manipulating varying-length sequences of char-like objects and four typedefs, \texttt{string}, \texttt{u16string}, \texttt{u32string}, and \texttt{wstring}, that name the specializations \texttt{basic\_string<\texttt{char}>}, \texttt{basic\_string<\texttt{char16\_t}>}, \texttt{basic\_string<\texttt{char32\_t}>}, and \texttt{basic\_string<\texttt{wchar\_t}>}, respectively.

Header \texttt{<string>} synopsis

```c
namespace std {

#include <initializer_list>

// 21.2, character traits:
template<class charT> struct char_traits;
template <> struct char_traits<char>;
template <> struct char_traits<char16_t>;
template <> struct char_traits<char32_t>;
template <> struct char_traits<wchar_t>;

// 21.4, basic\_string:
template<class charT, class traits = char_traits<charT>,
class Allocator = allocator<charT>>
class basic\_string;

template<class charT, class traits, class Allocator>
basic\_string\langle charT, traits, Allocator\rangle
operator+(const basic\_string\langle charT, traits, Allocator\rangle& lhs,
const basic\_string\langle charT, traits, Allocator\rangle& rhs);
template<class charT, class traits, class Allocator>
basic\_string\langle charT, traits, Allocator\rangle
operator+(basic\_string\langle charT, traits, Allocator\rangle&& lhs,
const basic\_string\langle charT, traits, Allocator\rangle& rhs);
template<class charT, class traits, class Allocator>
basic\_string\langle charT, traits, Allocator\rangle
operator+(const basic\_string\langle charT, traits, Allocator\rangle& lhs,
basic\_string\langle charT, traits, Allocator\rangle&& rhs);
template<class charT, class traits, class Allocator>
basic\_string\langle charT, traits, Allocator\rangle
operator+(basic\_string\langle charT, traits, Allocator\rangle&& lhs,
basic\_string\langle charT, traits, Allocator\rangle&& rhs);

```
const basic_string<charT,traits,Allocator>& rhs);

template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+(const charT* lhs,
basic_string<charT,traits,Allocator>&& rhs);

template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+(charT lhs, const basic_string<charT,traits,Allocator>& rhs);

template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+(charT lhs, basic_string<charT,traits,Allocator>&& rhs);

template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+(const basic_string<charT,traits,Allocator>&& lhs,
const charT* rhs);

template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+(basic_string<charT,traits,Allocator>&& lhs,
const charT* rhs);

template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+(const basic_string<charT,traits,Allocator>& lhs,
const charT* rhs);

template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+(basic_string<charT,traits,Allocator>&& lhs,
const charT* rhs);

bool operator==(const basic_string<charT,traits,Allocator>& lhs,
const basic_string<charT,traits,Allocator>& rhs);

bool operator==(const charT* lhs,
const basic_string<charT,traits,Allocator>& rhs);

bool operator==(const basic_string<charT,traits,Allocator>& lhs,
const charT* rhs);

bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
const basic_string<charT,traits,Allocator>& rhs);

bool operator!=(const charT* lhs,
const basic_string<charT,traits,Allocator>& rhs);

bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
const charT* rhs);

bool operator< (const basic_string<charT,traits,Allocator>& lhs,
const basic_string<charT,traits,Allocator>& rhs);

bool operator< (const basic_string<charT,traits,Allocator>& lhs,
const charT* rhs);

bool operator< (const charT* lhs,
const basic_string<charT,traits,Allocator>& rhs);

§ 21.3
bool operator> (const basic_string<charT,traits,Allocator>& lhs,
    const basic_string<charT,traits,Allocator>& rhs);

// 21.4.8.8: swap
template<class charT, class traits, class Allocator>
void swap(basic_string<charT,traits,Allocator>& lhs,
    basic_string<charT,traits,Allocator>& rhs);

// 21.4.8.9: inserters and extractors
template<class charT, class traits, class Allocator>
basic_istream<charT,traits>&
    operator>>(basic_istream<charT,traits>&& is,
        basic_string<charT,traits,Allocator>& str);

// basic_string typedef names
typedef basic_string<char> string;
typedef basic_string<char16_t> u16string;

§ 21.3
typedef basic_string<char32_t> u32string;
typedef basic_string<wchar_t> wstring;

// 21.5: numeric conversions
int stoi(const string& str, size_t *idx = 0, int base = 10);
long stol(const string& str, size_t *idx = 0, int base = 10);
unsigned long stoul(const string& str, size_t *idx = 0, int base = 10);
long long stoll(const string& str, size_t *idx = 0, int base = 10);
unsigned long long stoull(const string& str, size_t *idx = 0);
float stof(const string& str, size_t *idx = 0);
double stod(const string& str, size_t *idx = 0);
long double stold(const string& str, size_t *idx = 0);
string to_string(int val);
string to_string(unsigned val);
string to_string(long val);
string to_string(unsigned long val);
string to_string(long long val);
string to_string(float val);
string to_string(double val);
string to_string(long double val);

int stoi(const wstring& str, size_t *idx = 0, int base = 10);
long stol(const wstring& str, size_t *idx = 0, int base = 10);
unsigned long stoul(const wstring& str, size_t *idx = 0, int base = 10);
long long stoll(const wstring& str, size_t *idx = 0, int base = 10);
unsigned long long stoull(const wstring& str, size_t *idx = 0);
float stof(const wstring& str, size_t *idx = 0);
double stod(const wstring& str, size_t *idx = 0);
long double stold(const wstring& str, size_t *idx = 0);
wstring to_wstring(int val);
wstring to_wstring(unsigned val);
wstring to_wstring(long val);
wstring to_wstring(unsigned long val);
wstring to_wstring(long long val);
wstring to_wstring(float val);
wstring to_wstring(double val);
wstring to_wstring(long double val);

// 21.6 Hash support
template <class T> struct hash;
template <> struct hash<string>;
template <> struct hash<u16string>;
template <> struct hash<u32string>;
template <> struct hash<2string>;
}

21.4 Class template basic_string

The class template basic_string describes objects that can store a sequence consisting of a varying number of arbitrary char-like objects with the first element of the sequence at position zero. Such a sequence is also called a “string” if the type of the char-like objects that it holds is clear from context. In the rest of this Clause, the type of the char-like objects held in a basic_string object is designated by charT.
The member functions of `basic_string` use an object of the `Allocator` class passed as a template parameter to allocate and free storage for the contained char-like objects.

The class template `basic_string` conforms to the requirements for a Sequence Container (23.2.3), for a Reversible Container (23.2), and for an Allocator-aware container (96), except that `basic_string` does not construct or destroy its elements using `allocator_traits<Alloc>::construct` and `allocator_traits<Alloc>::destroy`. The iterators supported by `basic_string` are random access iterators (24.2.7).

In all cases, `size() <= capacity()`.

The functions described in this Clause can report two kinds of errors, each associated with an exception type:

- a `length` error is associated with exceptions of type `length_error` (19.2.4);
- an `out-of-range` error is associated with exceptions of type `out_of_range` (19.2.5).

```cpp
namespace std {
    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT> >
    class basic_string {
        public:
            // types:
            typedef traits traits_type;
            typedef typename traits::char_type value_type;
            typedef Allocator allocator_type;
            typedef typename allocator_traits<Allocator>::size_type size_type;
            typedef typename allocator_traits<Allocator>::difference_type difference_type;

            typedef value_type& reference;
            typedef const value_type& const_reference;
            typedef typename allocator_traits<Allocator>::pointer pointer;
            typedef typename allocator_traits<Allocator>::const_pointer const_pointer;

            typedef implementation-defined iterator; // See 23.2
            typedef implementation-defined const_iterator; // See 23.2
            typedef std::reverse_iterator<iterator> reverse_iterator;
            typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

            static const size_type npos = -1;

            // 21.4.2 construct/copy/destroy:
            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(initializer_list<charT>, const Allocator& = Allocator());
            basic_string(const basic_string&, const Allocator&);
```
basic_string(basic_string&&, const Allocator&);

'basic_string();
basic_string& operator=(const basic_string& str);
basic_string& operator=(basic_string&& str);
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);
basic_string& operator=(initializer_list<charT>);

// 21.4.3 iterators:
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// 21.4.4 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 shrink_to_fit();
void clear();
bool empty() const;

// 21.4.5 element access:
const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
const_reference at(size_type n) const;
reference at(size_type n);

const charT& front() const;
charT& front();
const charT& back() const;
charT& back();

// 21.4.6 modifiers:
basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
basic_string& operator+=(initializer_list<charT>);
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str, size_type pos,
size_type n);
basic_string& append(const charT* s, size_type n);
basic_string& append(const charT* s);
basic_string& append(size_type n, charT c);
template<class InputIterator>
  basic_string& append(InputIterator first, InputIterator last);
basic_string& append(initializer_list<charT>);
void push_back(charT c);

basic_string& assign(const basic_string& str);
basic_string& assign(basic_string&& str);
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);
  (initializer_list<charT>);

basic_string& insert(size_type pos1, const basic_string& str);
basic_string& insert(size_type pos1, const basic_string& str,
  size_type pos2, size_type n);
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);
template<class InputIterator>
  iterator insert(const_iterator p, charT c);
  iterator insert(const_iterator p, size_type n, charT c);
template <class InputIterator>
  iterator insert(const_iterator p, InputIterator first, InputIterator last);
  (initializer_list<charT>);

basic_string& erase(size_type pos = 0, size_type n = npos);
iterator erase(const_iterator p);
iterator erase(const_iterator first, const_iterator last);

void pop_back();

basic_string& replace(size_type pos1, size_type n1,
  const basic_string& str);
basic_string& replace(size_type pos1, size_type n1,
  const basic_string& str, size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s,
  size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s);
basic_string& replace(size_type pos, size_type n1, size_type n2,
  charT c);

basic_string& replace(iterator i1, iterator i2,
  const basic_string& str);
basic_string& replace(iterator i1, iterator i2, const charT* s,
  size_type n);
basic_string& replace(iterator i1, iterator i2, const charT* s);
basic_string& replace(iterator i1, iterator i2,
size_type n, charT c);

template<class InputIterator>
  basic_string& replace(iterator i1, iterator i2,
                InputIterator j1, InputIterator j2);

basic_string& replace(iterator, iterator, initializer_list<charT>);

size_type copy(charT* s, size_type n, size_type pos = 0) const;
void swap(basic_string& str);

// 21.4.7 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_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_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,
21.4.1  basic_string general requirements  [string.require]

1  If any operation would cause size() to exceed max_size(), that operation shall throw an exception object of type length_error.

2  If any member function or operator of basic_string throws an exception, that function or operator shall have no other effect.

3  No erase() or pop_back() member function shall throw any exceptions.

4  In every specialization basic_string<charT, traits, Allocator>, the type allocator_traits<Allocator>::value_type shall name the same type as charT. Every object of type basic_string<charT, traits, Allocator> shall use an object of type Allocator to allocate and free storage for the contained charT objects as needed. The Allocator object used shall be obtained as described in §23.2.1.

5  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().

6  References, pointers, and iterators referring to the elements of a basic_string sequence may be invalidated by the following uses of that basic_string object:
   — as an argument to any standard library function taking a reference to non-const basic_string as an argument.239
   — Calling non-const member functions, except operator[], at, front, back, begin, rbegin, end, and rend.

21.4.2  basic_string constructors and assignment operators  [string.cons]

explicit basic_string(const Allocator& a = Allocator());

1  Effects: Constructs an object of class basic_string. The postconditions of this function are indicated in Table 60.

<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>

239) For example, as an argument to non-member functions swap() (21.4.8.8), operator>>() (21.4.8.9), and getline() (21.4.8.9), or as an argument to basic_string::swap()
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 61. In the second form, str is left in a valid state with an unspecified value.

Throws: The second form throws nothing if the allocator’s move constructor throws nothing.

Table 61 — 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<charT,traits,Allocator>& str,
             size_type pos, size_type n = npos,
             const Allocator& a = Allocator());

Requires: pos <= str.size()

Throws: out_of_range if pos > str.size().

Effects: Constructs an object of class basic_string and determines the effective length rlen of the initial string value as the smaller of n and str.size() - pos, as indicated in Table 62.

Table 62 — basic_string(const basic_string&, size_type, size_type, const Allocator&) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>points at the first element of an allocated copy of rlen consecutive elements of the string controlled by str beginning at position pos</td>
</tr>
<tr>
<td>size()</td>
<td>rlen</td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as size()</td>
</tr>
</tbody>
</table>

basic_string(const charT* s, size_type n,
             const Allocator& a = Allocator());

Requires: s 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 63.

Table 63 — basic_string(const charT*, size_type, const Allocator&) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>points at the first element of an allocated copy of the array whose first element is pointed at by s</td>
</tr>
<tr>
<td>size()</td>
<td>n</td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as size()</td>
</tr>
</tbody>
</table>

basic_string(const charT* s, const Allocator& a = Allocator());
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 64.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data()</code></td>
<td>points at the first element of an allocated copy of the array whose first element is pointed at by ( s )</td>
</tr>
<tr>
<td><code>size()</code></td>
<td><code>traits::length(s)</code></td>
</tr>
<tr>
<td><code>capacity()</code></td>
<td>a value at least as large as <code>size()</code></td>
</tr>
</tbody>
</table>

Remarks: Uses `traits::length()`.

basic_string(size_type n, charT c, const Allocator& a = Allocator());

Requires: \( n < \text{npos} \)

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 65.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data()</code></td>
<td>points at the first element of an allocated array of ( n ) elements, each storing the initial value ( c )</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>( n )</td>
</tr>
<tr>
<td><code>capacity()</code></td>
<td>a value at least as large as <code>size()</code></td>
</tr>
</tbody>
</table>

Effects: Uses `traits::length()`.

basic_string(const charT*, const Allocator& a = Allocator());

basic_string(size_t, charT, const Allocator& a = Allocator());

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 \([\text{begin}, \text{end})\), as indicated in the Sequence Requirements table (see 23.2.3).

basic_string(initializer_list<charT> il, const Allocator& a = Allocator());

Effects: Same as `basic_string(il.begin(), il.end(), a)`.

basic_string(const basic_string& str, const Allocator& alloc);

Effects: Constructs an object of class `basic_string` as indicated in Table 66. The stored allocator is constructed from `alloc`. In the second form, `str` is left in a valid state with an unspecified value.

Throws: The second form throws nothing if `alloc == str.get_allocator()` unless the copy constructor for `Allocator` throws.

basic_string<charT,traits,Allocator>&
    operator=(const basic_string<charT,traits,Allocator>& str);
Table 66 — `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 <code>str.data()</code></td>
</tr>
<tr>
<td>size()</td>
<td>the original 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>
<tr>
<td><code>get_allocator()</code></td>
<td>alloc</td>
</tr>
</tbody>
</table>

Effects: If `*this` and `str` are not the same object, modifies `*this` as shown in Table 67. If `*this` and `str` are the same object, the member has no effect.

Returns: `*this`

Table 67 — `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 <code>str.data()</code></td>
</tr>
<tr>
<td>size()</td>
<td><code>str.size()</code></td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as <code>size()</code></td>
</tr>
</tbody>
</table>

basic_string<charT, traits, Allocator>&
operator=(basic_string<charT, traits, Allocator>&& str);

Effects: If `*this` and `str` are not the same object, modifies `*this` as shown in Table 68. 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`

Table 68 — `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>

basic_string<charT, traits, Allocator>&
operator=(const charT* s);

Returns: `*this = basic_string<charT, traits, Allocator>(s)`.

Remarks: Uses `traits::length()`.

basic_string<charT, traits, Allocator>& operator=(charT c);
Returns: \*this = basic_string<charT,traits,Allocator>(1,c).

basic_string& operator=(initializer_list<charT> il);

Effects: \*this = basic_string(il).

Returns: \*this.

21.4.3 basic_string iterator support

iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;

Returns: an iterator referring to the first character in the string.

iterator end();
const_iterator end() const;
const_iterator cend() const;

Returns: an iterator which is the past-the-end value.

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
const_reverse_iterator crbegin() const;

Returns: an iterator which is semantically equivalent to reverse_iterator(end()).

reverse_iterator rend();
const_reverse_iterator rend() const;
const_reverse_iterator crend() const;

Returns: an iterator which is semantically equivalent to reverse_iterator(begin()).

21.4.4 basic_string capacity

size_type size() const;

Returns: a count of the number of char-like objects currently in the string.

Throws: nothing.

Complexity: constant time.

size_type length() const;

Returns: size().

size_type max_size() const;

Returns: The size of the largest possible string.

Complexity: constant time.

void resize(size_type n, charT c);

Requires: \( n \leq \) max_size()

Throws: length_error if \( n > \) max_size().

Effects: Alters the length of the string designated by \*this as follows:
— If \( n \leq \text{size()} \), 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 \( n > \text{size()} \), the function replaces the string designated by \*this with a string of length \( n \) whose first \text{size()} elements are a copy of the original string designated by \*this, and whose remaining elements are all initialized to \( c \).

```cpp
void resize(size_type n);
Effects: resize(n,charT()).
```

```cpp
type capacity() const;
Returns: the size of the allocated storage in the string.
```

```cpp
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 \leq \text{size()} is in effect a non-binding shrink-to-fit request. — end note]
Throws: length_error if res_arg > max_size().
```

```cpp
void shrink_to_fit();
Remarks: shrink_to_fit is a non-binding request to reduce capacity() to size(). [Note: The request is non-binding to allow latitude for implementation-specific optimizations. — end note]
```

```cpp
void clear();
Effects: Behaves as if the function calls:
erase(begin(), end());

bool empty() const;
Returns: size() == 0.
```

### 21.4.5 basic_string element access

#### [string.access]

```cpp
const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
Requires: pos \leq \text{size()}.
Returns: *(begin() + pos) if pos < \text{size()}, otherwise a reference to an object of type \( T \) with value charT(); the referenced value shall not be modified.
Throws: nothing.
Complexity: constant time.
```

```cpp
const_reference at(size_type pos) const;
reference at(size_type pos);
```

240) reserve() uses allocator_traits<Allocator>::allocate() which may throw an appropriate exception.
5     Requires: pos < size()
6     Throws: out_of_range if pos >= size().
7     Returns: operator[](pos).

const charT& front() const;
charT& front();
8     Requires: !empty()
9     Effects: Equivalent to operator[](0).

const charT& back() const;
charT& back();
10    Requires: !empty()
11    Effects: Equivalent to operator[](size() - 1).

21.4.6 basic_string modifiers

21.4.6.1 basic_string::operator+=

basic_string&
operator+=(const basic_string& str);
1     Effects: Calls append(str.data, str.size()).
2     Returns: *this.

basic_string& operator+=(const charT* s);
3     Effects: Calls append(s).
4     Returns: *this.

basic_string& operator+=(charT c);
5     Effects: Calls append(1, c);
6     Returns: *this.

basic_string& operator+=(initializer_list<charT> il);
7     Effects: Calls append(il.first(), il.size()).
8     Returns: *this.

21.4.6.2 basic_string::append

basic_string&
append(const basic_string& str);
1     Effects: Calls append(str.data(), str.size()).
2     Returns: *this.

basic_string&
append(const basic_string& str, size_type pos, size_type n);
basic_string&
append(const charT* s, size_type n);

Requires: s points to an array of at least n elements of charT.
Throws: length_error if size() + n > max_size().
Effects: The function replaces the string controlled by *this with a string of length size() + n whose first size() elements are a copy of the original string controlled by *this and whose remaining elements are a copy of the initial n elements of s.

Returns: *this.

basic_string& append(const charT* s);

Requires: s points to an array of at least traits::length(s) + 1 elements of charT.
Effects: Calls append(s, traits::length(s)).

Returns: *this.

basic_string&
append(size_type n, charT c);

Effects: Equivalent to append(basic_string(n, c)).

Returns: *this.

template<class InputIterator>
basic_string& append(InputIterator first, InputIterator last);

Requires: [first, last) is a valid range.
Effects: Equivalent to append(basic_string(first, last)).

Returns: *this.

basic_string& append(initializer_list<charT> il);

Effects: Calls append(il.begin(), il.size()).

*this.

void push_back(charT c)

Effects: Equivalent to append(static_cast<size_type>(1), c).

### 21.4.6.3 basic_string::assign

[string::assign]

basic_string&
assign(const basic_string& str);
Effects: Equivalent to `assign(str, 0, npos)`.

Returns: *this.

basic_string&
assign(basic_string& str);

Effects: The function replaces the string controlled by *this with a string of length `str.size()` whose elements are a copy of the string controlled by str. Leaves str in a valid but unspecified state. [Note: A valid implementation is `swap(str)`. — end note]

Throws: Nothing.

Returns: *this.

basic_string&
assign(const basic_string& str, size_type pos, size_type n);

Requires: pos ≤ str.size()

Throws: out_of_range if pos > str.size().

Effects: Determines the effective length rlen of the string to assign as the smaller of n and str.size() - pos and calls assign(str.data() + pos rlen).

Returns: *this.

basic_string&
assign(const charT* s, size_type n);

Requires: s points to an array of at least n elements of charT.

Throws: length_error if n > max_size().

Effects: Replaces the string controlled by *this with a string of length n whose elements are a copy of those pointed to by s.

Returns: *this.

basic_string& assign(const charT* s);

Requires: s points to an array of at least traits::length(s) + 1 elements of charT.

Effects: Calls assign(s, traits::length(s)).

Returns: *this.

basic_string& assign(initializer_list<charT> il);

Effects: Calls assign(il.begin(), il.size()).

*this.

basic_string&
assign(size_type n, charT c);

Effects: Equivalent to `assign(basic_string(n, c))`.

Returns: *this.

template<class InputIterator>
basic_string& assign(InputIterator first, InputIterator last);
Effects: Equivalent to assign(basic_string(first, last)).

Returns: *this.

### 21.4.6.4 basic_string::insert

```cpp
basic_string&
insert(size_type pos1,
const basic_string& str);
```

Requires: pos <= size().

Throws: out_of_range if pos > size().

Effects: Calls insert(pos, str.data(), str.size()).

Returns: *this.

```cpp
basic_string&
insert(size_type pos1,
const basic_string& str,
size_type pos2, size_type n);
```

Requires: pos1 <= size() and pos2 <= str.size()

Throws: out_of_range if pos1 > size() or pos2 > str.size().

Effects: Determines the effective length rlen of the string to insert as the smaller of n and str.size() - pos2 and calls insert(pos1, str.data() + pos2, rlen).

Returns: *this.

```cpp
basic_string&
insert(size_type pos, const charT* s, size_type n);
```

Requires: s points to an array of at least n elements of charT and pos <= size().

Throws: out_of_range if pos > size() or length_error if size() + n > max_size().

Effects: Replaces the string controlled by *this with a string of length size() + n whose first pos elements are a copy of the initial elements of the original string controlled by *this and whose next n elements are a copy of the elements in s and whose remaining elements are a copy of the remaining elements of the original string controlled by *this.

Returns: *this.

```cpp
basic_string&
insert(size_type pos, const charT* s);
```

Requires: pos <= size() and s points to an array of at least traits::length(s) + 1 elements of charT.

Effects: Calls insert(pos, s, traits::length(s)).

Returns: *this.

```cpp
basic_string&
insert(size_type pos, size_type n, charT c);
```

Effects: Equivalent to insert(pos basic_string(n, c)).

Returns: *this.

§ 21.4.6.4
iterator insert(const_iterator p, charT c);

Requires: p is a valid iterator on *this.
Effects: inserts a copy of c before the character referred to by p.
Returns: an iterator which refers to the copy of the inserted character.

iterator insert(const_iterator p, size_type n, charT c);

Requires: p is a valid iterator on *this.
Effects: inserts n copies of c before the character referred to by p.
Returns: an iterator which refers to the copy of the first inserted character, or p if n == 0.

template<class InputIterator>
iterator insert(const_iterator p, InputIterator first, InputIterator last);

Requires: p is a valid iterator on *this. [first,last) is a valid range.
Effects: Equivalent to insert(p - begin(), basic_string(first, last)).
Returns: an iterator which refers to the copy of the first inserted character, or p if first == last.

iterator insert(const_iterator p, initializer_list<charT> il);

Effects: insert(p, il.begin(), il.end()).
Returns: an iterator which refers to the copy of the first inserted character, or p if il is empty.

21.4.6.5 basic_string::erase

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(const_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(const_iterator first, const_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 \texttt{last} prior to the other elements being erased. If no such element exists, \texttt{end()} is returned.

\begin{verbatim}
void pop_back();
\end{verbatim}

\textbf{Requires:} \texttt{!empty()}

\textbf{Effects:} Equivalent to \texttt{erase(size() - 1, 1)}.

\section*{21.4.6.6 \texttt{basic_string::replace}}

\texttt{basic_string&}
\begin{verbatim}
replace(size_type pos1, size_type n1, const basic_string& str);
\end{verbatim}

\textbf{Requires:} \texttt{pos1 <= size()}.

\textbf{Throws:} \texttt{out_of_range} if \texttt{pos1 > size()}.

\textbf{Effects:} Calls \texttt{replace(pos1, n1, str.data(), str.size())}.

\textbf{Returns:} \texttt{*this}.

\texttt{basic_string&}
\begin{verbatim}
replace(size_type pos1, size_type n1, const basic_string& str, size_type pos2, size_type n2);
\end{verbatim}

\textbf{Requires:} \texttt{pos1 <= size()} and \texttt{pos2 <= str.size()}.

\textbf{Throws:} \texttt{out_of_range} if \texttt{pos1 > size()} or \texttt{pos2 > str.size()}.

\textbf{Effects:} Determines the effective length \texttt{rlen} of the string to be inserted as the smaller of \texttt{n2} and \texttt{str.size() - pos2} and calls \texttt{replace(pos1, n1, str.data() + pos2, rlen)}.

\textbf{Returns:} \texttt{*this}.

\texttt{basic_string&}
\begin{verbatim}
replace(size_type pos1, size_type n1, const charT* s, size_type n2);
\end{verbatim}

\textbf{Requires:} \texttt{pos1 <= size()} and \texttt{s} points to an array of at least \texttt{n2} elements of \texttt{charT}.

\textbf{Throws:} \texttt{out_of_range} if \texttt{pos1 > size()} or \texttt{length_error} if the length of the resulting string would exceed \texttt{max_size()} (see below).

\textbf{Effects:} Determines the effective length \texttt{xlen} of the string to be removed as the smaller of \texttt{n1} and \texttt{size()} - \texttt{pos1}. If \texttt{size()} - \texttt{xlen} >= \texttt{max_size()} - \texttt{n2} throws \texttt{length_error}. Otherwise, the function replaces the string controlled by \texttt{*this} with a string of length \texttt{size()} - \texttt{xlen} + \texttt{n2} whose first \texttt{pos1} elements are a copy of the initial elements of the original string controlled by \texttt{*this}, whose next \texttt{n2} elements are a copy of the initial \texttt{n2} elements of \texttt{s}, and whose remaining elements are a copy of the elements of the original string controlled by \texttt{*this} beginning at position \texttt{pos + xlen}.

\textbf{Returns:} \texttt{*this}.

\texttt{basic_string&}
\begin{verbatim}
replace(size_type pos, size_type n, const charT* s);
\end{verbatim}

\textbf{Requires:} \texttt{pos <= size()} and \texttt{s} points to an array of at least \texttt{traits::length(s) + 1} elements of \texttt{charT}.

\section*{§ 21.4.6.6}
Effects: Calls replace(pos, n, s, traits::length(s)).

Returns: *this.

basic_string&
replace(size_type pos1, size_type n1,
size_type n2, charT c);

Effects: Equivalent to replace(pos1, n1, basic_string(n2, c)).

Returns: *this.

basic_string& replace(iterator i1, iterator i2, const basic_string& str);

Requires: [begin(),i1) and [i1,i2) are valid ranges.

Effects: Calls replace(i1 - begin(), i2 - i1, str).

Returns: *this.

basic_string& replace(iterator i1, iterator i2, const charT* s, size_type n);

Requires: [begin(),i1) and [i1,i2) are valid ranges and s points to an array of at least n elements of charT.

Effects: Calls replace(i1 - begin(), i2 - i1, s, n).

Returns: *this.

basic_string& replace(iterator i1, iterator i2, const charT* s);

Requires: [begin(),i1) and [i1,i2) are valid ranges and s points to an array of at least traits::length(s) + 1 elements of charT.

Effects: Calls replace(i1 - begin(), i2 - i1, s, traits::length(s)).

Returns: *this.

basic_string& replace(iterator i1, iterator i2, size_type n, charT c);

Requires: [begin(),i1) and [i1,i2) are valid ranges.

Effects: Calls replace(i1 - begin(), i2 - i1, basic_string(n, c)).

Returns: *this.

template<class InputIterator>
basic_string& replace(iterator i1, iterator i2, InputIterator j1, InputIterator j2);

Requires: [begin(),i1), [i1,i2) and [j1,j2) are valid ranges.

Effects: Calls replace(i1 - begin(), i2 - i1, basic_string(j1, j2)).

Returns: *this.

basic_string& replace(iterator i1, iterator i2, initializer_list<charT> il);

Requires: [begin(),i1) and [i1,i2) are valid ranges.

Effects: Calls replace(i1 - begin(), i2 - i1, il.begin(), il.size()).
21.4.6.7  basic_string::copy

size_type copy(charT* s, size_type n, size_type pos = 0) const;

Requires: pos <= size()

Throws: out_of_range if pos > size().

Effects: Determines the effective length rlen of the string to copy as the smaller of n and size() - pos. s shall designate an array of at least rlen elements.

The function then replaces the string designated by s with a string of length rlen whose elements are a copy of the string controlled by *this beginning at position pos.

The function does not append a null object to the string designated by s.

Returns: rlen.

21.4.6.8  basic_string::swap

void swap(basic_string<charT,traits,Allocator>& s);

Throws: Nothing.

Postcondition: *this contains the same sequence of characters that was in s, s contains the same sequence of characters that was in *this.

Complexity: constant time.

21.4.7  basic_string string operations

21.4.7.1  basic_string accessors

const charT* c_str() const;
const charT* data() const;

Returns: a pointer p such that p + i == &operator[](i) for each i in [0,size()).

Throws: nothing.

Complexity: constant time.

Requires: The program shall not alter any of the values stored in the character array.

allocator_type get_allocator() const;

Returns: a copy of the Allocator object used to construct the string or, if that allocator has been replaced, a copy of the most recent replacement.

21.4.7.2  basic_string::find

size_type find(const basic_string& str, size_type pos = 0) const;

Effects: Determines the lowest position xpos, if possible, such that both of the following conditions obtain:
— \( \text{pos} \leq \text{xpos} \) and \( \text{xpos} + \text{str.size()} \leq \text{size()} \);
— traits::eq(at(xpos+I), str.at(I)) for all elements I of the string controlled by str.

\textit{Returns}: \text{xpos} if the function can determine such a value for \text{xpos}. Otherwise, returns \text{npos}.

\textit{Remarks}: Uses traits::eq().

\begin{verbatim}
size_type find(const charT* s, size_type pos, size_type n) const;
\end{verbatim}

\textit{Returns}: find(basic_string<charT,traits,Allocator>(s,n),pos).

\begin{verbatim}
size_type find(const charT* s, size_type pos = 0) const;
\end{verbatim}

\textit{Requires}: \text{s} points to an array of at least traits::length(s) + 1 elements of charT.

\textit{Returns}: find(basic_string(s), pos).

\begin{verbatim}
size_type find(charT c, size_type pos = 0) const;
\end{verbatim}

\textit{Returns}: find(basic_string<charT,traits,Allocator>(1,c),pos).

\subsection{21.4.7.3 basic_string::rfind} \[\text{string::rfind}\]

\begin{verbatim}
size_type rfind(const basic_string& str, size_type pos = npos) const;
\end{verbatim}

\textit{Effects}: Determines the highest position \text{xpos}, if possible, such that both of the following conditions obtain:
— \( \text{xpos} \leq \text{pos} \) and \( \text{xpos} + \text{str.size()} \leq \text{size()} \);
— traits::eq(at(xpos+I), str.at(I)) for all elements I of the string controlled by str.

\textit{Returns}: \text{xpos} if the function can determine such a value for \text{xpos}. Otherwise, returns \text{npos}.

\textit{Remarks}: Uses traits::eq().

\begin{verbatim}
size_type rfind(const charT* s, size_type pos, size_type n) const;
\end{verbatim}

\textit{Returns}: rfind(basic_string(s, n), pos).

\begin{verbatim}
size_type rfind(const charT* s, size_type pos = npos) const;
\end{verbatim}

\textit{Requires}: \text{s} points to an array of at least traits::length(s) + 1 elements of charT.

\textit{Returns}: rfind(basic_string(s), pos).

\begin{verbatim}
size_type rfind(charT c, size_type pos = npos) const;
\end{verbatim}

\textit{Returns}: rfind(basic_string<charT,traits,Allocator>(1,c),pos).

\subsection{21.4.7.4 basic_string::find_first_of} \[\text{string::find.first.of}\]

\begin{verbatim}
size_type find_first_of(const basic_string& str, size_type pos = 0) const;
\end{verbatim}

\textit{Effects}: Determines the lowest position \text{xpos}, if possible, such that both of the following conditions obtain:
— \( \text{pos} \leq \text{xpos} \) and \( \text{xpos} < \text{size()} \);

\section{§ 21.4.7.4}
- \texttt{traits::eq(at(xpos), str.at(I))} for some element \( I \) of the string controlled by \( \text{str} \).

\textit{Returns:} \( \text{xpos} \) if the function can determine such a value for \( \text{xpos} \). Otherwise, returns \( \text{npos} \).

\textit{Remarks:} Uses \texttt{traits::eq()}.  

\begin{verbatim}
size_type
    find_first_of(const charT* s, size_type pos, size_type n) const;

Returns: \( \text{find_first_of(basic_string(s, n), pos)} \).

size_type find_first_of(const charT* s, size_type pos = 0) const;

Requires: \( s \) points to an array of at least \texttt{traits::length(s) + 1} elements of \texttt{charT}.

Returns: \( \text{find_first_of(basic_string(s), pos)} \).

size_type find_first_of(charT c, size_type pos = 0) const;

Returns: \( \text{find_first_of(basic_string<charT,traits,Allocator>(1,c),pos)} \).
\end{verbatim}

\subsection*{21.4.7.5 \texttt{basic_string::find_last_of} [string::find.last.of]}

\begin{verbatim}
size_type
    find_last_of(const basic_string& str,
                 size_type pos = npos) const;

Effects: Determines the highest position \( \text{xpos} \), if possible, such that both of the following conditions obtain:
- \( \text{xpos} \leq \text{pos} \) and \( \text{xpos} < \text{size()} \);
- \texttt{traits::eq(at(xpos), str.at(I))} for some element \( I \) of the string controlled by \( \text{str} \).

Returns: \( \text{xpos} \) if the function can determine such a value for \( \text{xpos} \). Otherwise, returns \( \text{npos} \).

Remarks: Uses \texttt{traits::eq()}.  

size_type find_last_of(const charT* s, size_type pos, size_type n) const;

Returns: \( \text{find_last_of(basic_string(s, n), pos)} \).

size_type find_last_of(const charT* s, size_type pos = npos) const;

Requires: \( s \) points to an array of at least \texttt{traits::length(s) + 1} elements of \texttt{charT}.

Returns: \( \text{find_last_of(basic_string(s), pos)} \).

size_type find_last_of(charT c, size_type pos = npos) const;

Returns: \( \text{find_last_of(basic_string<charT,traits,Allocator>(1,c),pos)} \).
\end{verbatim}

\subsection*{21.4.7.6 \texttt{basic_string::find_first_not_of} [string::find.first.not.of]}

\begin{verbatim}
size_type
    find_first_not_of(const basic_string& str,
                      size_type pos = 0) const;

Effects: Determines the lowest position \( \text{xpos} \), if possible, such that both of the following conditions obtain:
- \( \text{xpos} \leq \text{pos} \) and \( \text{xpos} < \text{size()} \);
- \texttt{traits::eq(at(xpos), str.at(I))} for some element \( I \) of the string controlled by \( \text{str} \).

\end{verbatim}
size_type
find_first_not_of(const charT* s, size_type pos, size_type n) const;

Returns: find_first_not_of(basic_string(s, n), pos).

size_type find_first_not_of(const charT* s, size_type pos = 0) const;

Requires: s points to an array of at least traits::length(s) + 1 elements of charT.

Returns: find_first_not_of(basic_string(s), pos).

size_type find_first_not_of(charT c, size_type pos = 0) const;

Returns: find_first_not_of(basic_string(1, c), pos).

21.4.7.7 basic_string::find_last_not_of

size_type
find_last_not_of(const basic_string& str,
     size_type pos = npos) const;

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(s, n), pos).

size_type find_last_not_of(const charT* s, size_type pos = npos) const;

Requires: s points to an array of at least traits::length(s) + 1 elements of charT.

Returns: find_last_not_of(basic_string(s), pos).

size_type find_last_not_of(charT c, size_type pos = npos) const;

Returns: find_last_not_of(basic_string(1, c), pos).

21.4.7.8 basic_string::substr

basic_string<charT,traits,Allocator>
substr(size_type pos = 0, size_type n = npos) 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.
4  Returns: basic_string<charT,traits,Allocator>(data()+pos,rlen).

21.4.7.9 basic_string::compare

int compare(const basic_string& str) const

Effects: Determines the effective length rlen of the strings to compare as the smallest of size() and str.size(). The function then compares the two strings by calling traits::compare(data(), str.data(), rlen).

Returns: the nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table 69.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>size() &lt; str.size()</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>size() == str.size()</td>
<td>0</td>
</tr>
<tr>
<td>size() &gt; str.size()</td>
<td>&gt; 0</td>
</tr>
</tbody>
</table>

int compare(size_type pos1, size_type n1,
const basic_string& str) const;

Returns:

basic_string(*this,pos1,n1).compare(str);

int compare(size_type pos1, size_type n1,
const basic_string& str,
size_type pos2, size_type n2 ) const;

Returns:

basic_string(*this,pos1,n1).compare(
  basic_string(str,pos2,n2)).

int compare(const charT *s) const;

Returns: compare(basic_string(s)).

int compare(size_type pos, size_type n1,
const charT *s) const;

Returns:

basic_string(*this,pos,n1).compare(
  basic_string(*s))

int compare(size_type pos, size_type n1,
const charT *s, size_type n2) const;

§ 21.4.7.9
Returns:

\[
\text{basic_string(*this,pos,n1).compare(basic_string(s,n2))}
\]

21.4.8 basic_string non-member functions

21.4.8.1 operator+

\[
\text{template<class charT, class traits, class Allocator>}
\]

\[
\text{basic_string<charT,traits,Allocator>}
\]

\[
\text{operator+(const basic_string<charT,traits,Allocator>& lhs,}
\]

\[
\text{const basic_string<charT,traits,Allocator>& rhs);}
\]

Returns: basic_string<charT,traits,Allocator>(lhs).append(rhs)

\[
\text{template<class charT, class traits, class Allocator>}
\]

\[
\text{basic_string<charT,traits,Allocator>}
\]

\[
\text{operator+(const basic_string<charT,traits,Allocator>&& lhs,}
\]

\[
\text{const basic_string<charT,traits,Allocator>&& rhs);}
\]

Returns: std::move(lhs.append(rhs))

\[
\text{template<class charT, class traits, class Allocator>}
\]

\[
\text{basic_string<charT,traits,Allocator>}
\]

\[
\text{operator+(basic_string<charT,traits,Allocator>&& lhs,}
\]

\[
\text{const basic_string<charT,traits,Allocator>& rhs);}
\]

Returns: std::move(rhs.insert(0, lhs))

\[
\text{template<class charT, class traits, class Allocator>}
\]

\[
\text{basic_string<charT,traits,Allocator>}
\]

\[
\text{operator+(const charT* lhs,}
\]

\[
\text{const basic_string<charT,traits,Allocator>& rhs);}
\]

Returns: basic_string<charT,traits,Allocator>(lhs) + rhs.

Remarks: Uses traits::length().
const basic_string<charT,traits,Allocator>& rhs);

Returns: basic_string<charT,traits,Allocator>(1,lhs) + rhs.

template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+(charT lhs,
  basic_string<charT,traits,Allocator>&& rhs);

Returns: std::move(rhs.insert(0, 1, lhs)).

template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+(const basic_string<charT,traits,Allocator>& lhs, const charT* rhs);

Returns: lhs + basic_string<charT,traits,Allocator>(rhs).

Remarks: Uses traits::length().

template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+(const basic_string<charT,traits,Allocator>& lhs, charT* rhs);

Returns: std::move(lhs.append(rhs)).

Remarks: Uses traits::length().

template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+(const basic_string<charT,traits,Allocator>& lhs, const charT* rhs);

Returns: std::move(lhs.append(1, rhs)).

21.4.8.2 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.

template<class charT, class traits, class Allocator>
bool operator==(const charT* lhs, const basic_string<charT,traits,Allocator>& rhs);

Returns: rhs == lhs.

template<class charT, class traits, class Allocator>
bool operator==(const basic_string<charT,traits,Allocator>& lhs, const charT* rhs);

§ 21.4.8.2
Requires: rhs points to an array of at least traits::length(rhs) + 1 elements of charT.

Returns: lhs.compare(rhs) == 0.

21.4.8.3 operator!=

    template<class charT, class traits, class Allocator>
    bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
                    const basic_string<charT,traits,Allocator>& rhs);

    Returns: !(lhs == rhs).

    template<class charT, class traits, class Allocator>
    bool operator!=(const charT* lhs,
                    const basic_string<charT,traits,Allocator>& rhs);

    Returns: rhs != lhs.

    template<class charT, class traits, class Allocator>
    bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
                    const charT* rhs);

    Requires: rhs points to an array of at least traits::length(rhs) + 1 elements of charT.
    Returns: lhs.compare(rhs) != 0.

21.4.8.4 operator<

    template<class charT, class traits, class Allocator>
    bool operator< (const basic_string<charT,traits,Allocator>& lhs,
                    const basic_string<charT,traits,Allocator>& rhs);

    Returns: lhs.compare(rhs) < 0.

    template<class charT, class traits, class Allocator>
    bool operator< (const charT* lhs,
                    const basic_string<charT,traits,Allocator>& rhs);

    Returns: basic_string<charT,traits,Allocator>(lhs) < rhs.

    template<class charT, class traits, class Allocator>
    bool operator< (const basic_string<charT,traits,Allocator>& lhs,
                    const charT* rhs);

    Returns: lhs < basic_string<charT,traits,Allocator>(rhs).

21.4.8.5 operator>

    template<class charT, class traits, class Allocator>
    bool operator> (const basic_string<charT,traits,Allocator>& lhs,
                    const basic_string<charT,traits,Allocator>& rhs);

    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).

21.4.8.6 operator<=

template<class charT, class traits, class Allocator>
bool operator<=(const basic_string<charT, traits, Allocator>& lhs,
const basic_string<charT, traits, Allocator>& rhs);

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).

21.4.8.7 operator>=

template<class charT, class traits, class Allocator>
bool operator>=(const basic_string<charT, traits, Allocator>& lhs,
const basic_string<charT, traits, Allocator>& rhs);

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).

21.4.8.8 swap

template<class charT, class traits, class Allocator>
void swap(basic_string<charT, traits, Allocator>& lhs,
basic_string<charT, traits, Allocator>& rhs);

1 Effects: lhs.swap(rhs);
21.4.8.9 Inserters and extractors

```
template<class charT, class traits, class Allocator>
    basic_istream<charT,traits>&
    operator>>(basic_istream<charT,traits>& is,
               basic_string<charT,traits,Allocator>& str);

1 Effects: Behaves as a formatted input function (27.7.1.2.1). 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.

2 After the last character (if any) is extracted, is.width(0) is called and the sentry object k is
destroyed.

3 If the function extracts no characters, it calls is.setstate(ios::failbit), which may throw ios_-
base::failure (27.5.4.3).

Returns: is
```

```
Returns: os
```

```
template<class charT, class traits, class Allocator>
    basic_istream<charT,traits>&
    getline(basic_istream<charT,traits>& is,
            basic_string<charT,traits,Allocator>& str,
            charT delim);

template<class charT, class traits, class Allocator>
    basic_istream<charT,traits>&
    getline(basic_istream<charT,traits>&& is,
            basic_string<charT,traits,Allocator>& str,
            charT delim);

7 Effects: Behaves as an unformatted input function (27.7.1.3), 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) (27.5.4.3)
— str.max_size() characters are stored (in which case, the function calls is.setstate(ios_base::failbit)) (27.5.4.3)

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 (27.5.4.3).

Returns: is.

10

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(‘\n’))

21.5 Numeric Conversions [string.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 `strtol` reports that no conversion could be performed. Throws out_of_range if `strtod` or `strtol` sets `errno` to ERANGE.

```plaintext
string to_string(int val);
string to_string(unsigned val);
string to_string(long val);
string to_string(unsigned long val);
string to_string(long long val);
string to_string(unsigned long long val);
string to_string(float val);
string to_string(double val);
string to_string(long double val);
```

Returns: each function returns a string object holding the character representation of the value of its argument that would be generated by calling `sprintf(buf, fmt, val)` with a format specifier of "%d", "%u", "%ld", "%lu", "%lld", "%llu", "%f", "%f", or "%Lf", respectively, where `buf` designates an internal character buffer of sufficient size.

```plaintext
int stoi(const wstring& str, size_t *idx = 0, int base = 10);
long stol(const wstring& str, size_t *idx = 0, int base = 10);
unsigned long stoul(const wstring& str, size_t *idx = 0, int base = 10);
long long stoll(const wstring& str, size_t *idx = 0, int base = 10);
unsigned long long stoull(const wstring& str, size_t *idx = 0, int base = 10);
```

Effects: the first two functions call `wcstol(str.c_str(), ptr, base)`, and the last three functions call `wcstoul(str.c_str(), ptr, base), wcstoll(str.c_str(), ptr, base), and wcstoull(str.c_str(), ptr, base)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx` != 0, the function stores in `*idx` the index of the first unconverted element of `str`.

Returns: the converted result.

Throws: invalid_argument if `wcstol`, `wcstoul`, `wcstoll`, or `wcstoull` reports that no conversion could be performed. Throws out_of_range if the converted value is outside the range of representable values for the return type.

```plaintext
float stof(const wstring& str, size_t *idx = 0);
double stod(const wstring& str, size_t *idx = 0);
long double stold(const wstring& str, size_t *idx = 0);
```

Effects: the first two functions call `wcstod(str.c_str(), ptr)` and the third function calls `wcstold(str.c_str(), ptr)`. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx` != 0, the function stores in `*idx` the index of the first unconverted element of `str`.

Returns: the converted result.

Throws: invalid_argument if `wcstod` or `wcstold` reports that no conversion could be performed. Throws out_of_range if `wcstod` or `wcstold` sets `errno` to ERANGE.

```plaintext
wstring to_wstring(int val);
wstring to_wstring(unsigned val);
wstring to_wstring(long val);
wstring to_wstring(unsigned long val);
wstring to_wstring(long long val);
```
wstring to_wstring(unsigned long long val);
wstring to_wstring(float val);
wstring to_wstring(double val);
wstring to_wstring(long double val);

Returns: Each function returns a wstring object holding the character representation of the value of its argument that would be generated by calling swprintf(buf, buffsz, fmt, val) with a format specifier of L"%d", L"%u", L"%ld", L"%lu", L"%lld", L"%llu", L"%f", L"%f", or L"%Lf", respectively, where buf designates an internal character buffer of sufficient size buffsz.

21.6 Hash support

template <> struct hash<string>;
template <> struct hash<u16string>;
template <> struct hash<u32string>;
template <> struct hash<wstring>;

Requires: the template specializations shall meet the requirements of class template hash (20.8.15).

21.7 Null-terminated sequence utilities

Tables 71, 72, 73, 74, 75, and 76 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>, <cwchar.h>, and <cstdlib.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 (2.12).
2. The function signature strchr(const char*, int) shall be replaced by the two declarations:
   
   const char* strchr(const char* s, int c);
   char* strchr(char* s, int c);

   both of which shall have the same behavior as the original declaration.
3. The function signature strpbrk(const char*, const char*) shall be replaced by the two declarations:
   
   const char* strpbrk(const char* s1, const char* s2);
   char* strpbrk(char* s1, const char* s2);

   both of which shall have the same behavior as the original declaration.
4. The function signature strrchr(const char*, int) shall be replaced by the two declarations:
   
   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.
5. The function signature strstr(const char*, const char*) shall be replaced by the two declarations:
   
   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.
6. The function signature memchr(const void*, int, size_t) shall be replaced by the two declarations:

§ 21.7
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:

    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:

    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:

    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:

    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:

    const wchar_t* wmemchr(const wchar_t* s, wchar_t c, size_t n);
    wchar_t* wmemchr( wchar_t* s, wchar_t c, size_t n);

both of which shall have the same behavior as the original declaration.

14 The functions strerror and strtok are not required to avoid data races (17.6.4.8).

15 Calling the functions listed in Table 70 with an mbstate_t* argument of NULL may introduce a data race (17.6.4.8) with other calls to these functions with an mbstate_t* argument of NULL.

Table 70 — Potential mbstate_t data races

<table>
<thead>
<tr>
<th>mbrlen</th>
<th>mbtowc</th>
<th>mbsrtowc</th>
<th>mbtowc</th>
<th>wcrtomb</th>
</tr>
</thead>
<tbody>
<tr>
<td>wcsrtomb</td>
<td>wcrtomb</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See also: ISO C 7.3, 7.10.7, 7.10.8, and 7.11. Amendment 1 4.4, 4.5, and 4.6.
Table 71 — Header `<cctype>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Functions:</strong></td>
</tr>
<tr>
<td></td>
<td>isalnum</td>
</tr>
<tr>
<td></td>
<td>isblank</td>
</tr>
<tr>
<td></td>
<td>isdigit</td>
</tr>
<tr>
<td></td>
<td>isprint</td>
</tr>
<tr>
<td></td>
<td>isupper</td>
</tr>
<tr>
<td></td>
<td>tolower</td>
</tr>
<tr>
<td></td>
<td>isalpha</td>
</tr>
<tr>
<td></td>
<td>isgraph</td>
</tr>
<tr>
<td></td>
<td>ispunct</td>
</tr>
<tr>
<td></td>
<td>isxdigit</td>
</tr>
<tr>
<td></td>
<td>toupper</td>
</tr>
<tr>
<td></td>
<td>iscntrl</td>
</tr>
<tr>
<td></td>
<td>islower</td>
</tr>
<tr>
<td></td>
<td>isspace</td>
</tr>
</tbody>
</table>

Table 72 — Header `<cwctype>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Macro:</strong> WEOF <code>&lt;cwctype&gt;</code></td>
</tr>
<tr>
<td></td>
<td><strong>Types:</strong> wctrans_t wctype_t wint_t <code>&lt;cwctype&gt;</code></td>
</tr>
<tr>
<td></td>
<td><strong>Functions:</strong></td>
</tr>
<tr>
<td></td>
<td>iswalnum</td>
</tr>
<tr>
<td></td>
<td>iswcntrl</td>
</tr>
<tr>
<td></td>
<td>iswgraph</td>
</tr>
<tr>
<td></td>
<td>iswpunct</td>
</tr>
<tr>
<td></td>
<td>iswxdigit</td>
</tr>
<tr>
<td></td>
<td>towupper</td>
</tr>
<tr>
<td></td>
<td>iswalpha</td>
</tr>
<tr>
<td></td>
<td>iswctype</td>
</tr>
<tr>
<td></td>
<td>iswlower</td>
</tr>
<tr>
<td></td>
<td>iswspace</td>
</tr>
<tr>
<td></td>
<td>towctrans</td>
</tr>
<tr>
<td></td>
<td>wctrans</td>
</tr>
<tr>
<td></td>
<td>iswblank</td>
</tr>
<tr>
<td></td>
<td>iswdigit</td>
</tr>
<tr>
<td></td>
<td>iswprint</td>
</tr>
<tr>
<td></td>
<td>iswupper</td>
</tr>
<tr>
<td></td>
<td>towlower</td>
</tr>
<tr>
<td></td>
<td>wctype</td>
</tr>
</tbody>
</table>

Table 73 — Header `<cstring>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Macro:</strong> NULL <code>&lt;cstring&gt;</code></td>
</tr>
<tr>
<td></td>
<td><strong>Type:</strong> size_t <code>&lt;cstring&gt;</code></td>
</tr>
<tr>
<td></td>
<td><strong>Functions:</strong></td>
</tr>
<tr>
<td></td>
<td>memchr</td>
</tr>
<tr>
<td></td>
<td>strcat</td>
</tr>
<tr>
<td></td>
<td>strcspn</td>
</tr>
<tr>
<td></td>
<td>strncpy</td>
</tr>
<tr>
<td></td>
<td>strtok</td>
</tr>
<tr>
<td></td>
<td>memcmp</td>
</tr>
<tr>
<td></td>
<td>strchr</td>
</tr>
<tr>
<td></td>
<td>strerror</td>
</tr>
<tr>
<td></td>
<td>strpbrk</td>
</tr>
<tr>
<td></td>
<td>strxfrm</td>
</tr>
<tr>
<td></td>
<td>memcpy</td>
</tr>
<tr>
<td></td>
<td>strcmp</td>
</tr>
<tr>
<td></td>
<td>strlen</td>
</tr>
<tr>
<td></td>
<td>strchr</td>
</tr>
<tr>
<td></td>
<td>strpbrk</td>
</tr>
<tr>
<td></td>
<td>strxfrm</td>
</tr>
<tr>
<td></td>
<td>memmove</td>
</tr>
<tr>
<td></td>
<td>strcoll</td>
</tr>
<tr>
<td></td>
<td>strncat</td>
</tr>
<tr>
<td></td>
<td>strspn</td>
</tr>
<tr>
<td></td>
<td>memset</td>
</tr>
<tr>
<td></td>
<td>strcpy</td>
</tr>
<tr>
<td></td>
<td>strncmp</td>
</tr>
<tr>
<td></td>
<td>strstr</td>
</tr>
</tbody>
</table>

Table 74 — Header `<cwchar>` synopsis

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Macros:</strong> NULL <code>&lt;cwchar&gt;</code> WCHAR_MAX WCHAR_MIN WEOF <code>&lt;cwchar&gt;</code></td>
</tr>
<tr>
<td></td>
<td><strong>Types:</strong> mbstate_t wint_t <code>&lt;cwchar&gt;</code> size_t <code>&lt;cwchar&gt;</code> tm</td>
</tr>
<tr>
<td></td>
<td><strong>Functions:</strong></td>
</tr>
<tr>
<td></td>
<td>btowc</td>
</tr>
<tr>
<td></td>
<td>mbrlen</td>
</tr>
<tr>
<td></td>
<td>vfwscanf</td>
</tr>
<tr>
<td></td>
<td>wcscpy</td>
</tr>
<tr>
<td></td>
<td>wcsspn</td>
</tr>
<tr>
<td></td>
<td>wcslxfrm</td>
</tr>
<tr>
<td></td>
<td>fgetwc</td>
</tr>
<tr>
<td></td>
<td>mbtowc</td>
</tr>
<tr>
<td></td>
<td>vswchar</td>
</tr>
<tr>
<td></td>
<td>wcsanspn</td>
</tr>
<tr>
<td></td>
<td>wcslstr</td>
</tr>
<tr>
<td></td>
<td>wctob</td>
</tr>
<tr>
<td></td>
<td>fgetws</td>
</tr>
<tr>
<td></td>
<td>mbsinit</td>
</tr>
<tr>
<td></td>
<td>vsprintf</td>
</tr>
<tr>
<td></td>
<td>wcsftime</td>
</tr>
<tr>
<td></td>
<td>wcstod</td>
</tr>
<tr>
<td></td>
<td>wmemchr</td>
</tr>
<tr>
<td></td>
<td>fputwc</td>
</tr>
<tr>
<td></td>
<td>mbsrtowcs</td>
</tr>
<tr>
<td></td>
<td>vwchar</td>
</tr>
<tr>
<td></td>
<td>wcslent</td>
</tr>
<tr>
<td></td>
<td>wcstof</td>
</tr>
<tr>
<td></td>
<td>wmemcmp</td>
</tr>
<tr>
<td></td>
<td>fputws</td>
</tr>
<tr>
<td></td>
<td>putwc</td>
</tr>
<tr>
<td></td>
<td>vsscanf</td>
</tr>
<tr>
<td></td>
<td>wcsncat</td>
</tr>
<tr>
<td></td>
<td>wcstok</td>
</tr>
<tr>
<td></td>
<td>wmemcpy</td>
</tr>
<tr>
<td></td>
<td>fwide</td>
</tr>
<tr>
<td></td>
<td>putwchar</td>
</tr>
<tr>
<td></td>
<td>wcrtomb</td>
</tr>
<tr>
<td></td>
<td>wcsncmp</td>
</tr>
<tr>
<td></td>
<td>wcstol</td>
</tr>
<tr>
<td></td>
<td>wmemmove</td>
</tr>
<tr>
<td></td>
<td>fprintf</td>
</tr>
<tr>
<td></td>
<td>swprintf</td>
</tr>
<tr>
<td></td>
<td>wcscat</td>
</tr>
<tr>
<td></td>
<td>wcsncpy</td>
</tr>
<tr>
<td></td>
<td>wcstold</td>
</tr>
<tr>
<td></td>
<td>wmemset</td>
</tr>
<tr>
<td></td>
<td>fswcanf</td>
</tr>
<tr>
<td></td>
<td>swscanf</td>
</tr>
<tr>
<td></td>
<td>wcscat</td>
</tr>
<tr>
<td></td>
<td>wcscaten</td>
</tr>
<tr>
<td></td>
<td>wcstok</td>
</tr>
<tr>
<td></td>
<td>wprintf</td>
</tr>
<tr>
<td></td>
<td>ungetwc</td>
</tr>
<tr>
<td></td>
<td>wcscmp</td>
</tr>
<tr>
<td></td>
<td>wcscat</td>
</tr>
<tr>
<td></td>
<td>wscanf</td>
</tr>
<tr>
<td></td>
<td>getwc</td>
</tr>
<tr>
<td></td>
<td>wcscmp</td>
</tr>
<tr>
<td></td>
<td>wcscat</td>
</tr>
<tr>
<td></td>
<td>wcstoul</td>
</tr>
<tr>
<td></td>
<td>wscanf</td>
</tr>
</tbody>
</table>

§ 21.7
Table 75 — 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></td>
</tr>
<tr>
<td>atof</td>
<td>mblen</td>
</tr>
<tr>
<td>atoi</td>
<td>mbtowc</td>
</tr>
<tr>
<td>atol</td>
<td>mbstowcs</td>
</tr>
<tr>
<td>atoll</td>
<td>strtol</td>
</tr>
<tr>
<td></td>
<td>strtold</td>
</tr>
<tr>
<td></td>
<td>wcstombs</td>
</tr>
<tr>
<td></td>
<td>strtof</td>
</tr>
<tr>
<td></td>
<td>strtoul</td>
</tr>
<tr>
<td></td>
<td>strtof</td>
</tr>
<tr>
<td></td>
<td>strtoull</td>
</tr>
<tr>
<td></td>
<td>wctomb</td>
</tr>
<tr>
<td></td>
<td>wcstombs</td>
</tr>
</tbody>
</table>

Table 76 — 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></td>
</tr>
<tr>
<td>mbtowc16</td>
<td>c16rtomb</td>
</tr>
<tr>
<td>mbtowc32</td>
<td>c32rtomb</td>
</tr>
</tbody>
</table>
22 Localization library

22.1 General

This Clause describes components that C++ programs may use to encapsulate (and therefore be more portable when confronting) cultural differences. The locale facility includes internationalization support for character classification and string collation, numeric, monetary, and date/time formatting and parsing, and message retrieval.

The following subclauses describe components for locales themselves, the standard facets, and facilities from the ISO C library, as summarized in Table 77.

Table 77 — Localization library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.3 Locales</td>
<td>&lt;locale&gt;</td>
</tr>
<tr>
<td>22.4 Standard locale Categories</td>
<td></td>
</tr>
<tr>
<td>22.5 Standard code conversion facets</td>
<td>&lt;codecvt&gt;</td>
</tr>
<tr>
<td>22.6 C library locales</td>
<td>&lt;clocale&gt;</td>
</tr>
</tbody>
</table>

22.2 Header <locale> synopsis

```
namespace std {
    // 22.3.1, locale:
    class locale;
    template <class Facet> const Facet& use_facet(const locale&);
    template <class Facet> bool has_facet(const locale&) throw();

    // 22.3.3, convenience interfaces:
    template <class charT> bool isspace (charT c, const locale& loc);
    template <class charT> bool isprint (charT c, const locale& loc);
    template <class charT> bool iscntrl (charT c, const locale& loc);
    template <class charT> bool isupper (charT c, const locale& loc);
    template <class charT> bool islower (charT c, const locale& loc);
    template <class charT> bool isalpha (charT c, const locale& loc);
    template <class charT> bool isdigit (charT c, const locale& loc);
    template <class charT> bool ispunct (charT c, const locale& loc);
    template <class charT> bool isxdigit(charT c, const locale& loc);
    template <class charT> bool isgraph (charT c, const locale& loc);
    template <class charT> charT toupper(charT c, const locale& loc);
    template <class charT> charT tolower(charT c, const locale& loc);
    template <class Codecvt, class Elem = wchar_t,
               class Wide_alloc = std::allocator<Elem>,
               class Byte_alloc = std::allocator<char> > class wstring_convert;
    template <class Codecvt, class Elem = wchar_t,
              class Tr = char_traits<Elem>> class wbuffer_convert;

    // 22.4.1 and 22.4.1.3, ctype:
```
class ctype_base;
template <class charT> class ctype;
template <> class ctype<char>; // specialization
template <class charT> class ctype_byname;
class codecvt_base;
template <class internT, class externT, class stateT> class codecvt;
template <class internT, class externT, class stateT> class codecvt_byname;

// 22.4.2 and 22.4.3, numeric:
template <class charT, class InputIterator = istreambuf_iterator<charT>> class num_get;
template <class charT, class OutputIterator = ostreambuf_iterator<charT>> class num_put;
template <class charT> class numpunct;
template <class charT> class numpunct_byname;

// 22.4.4, collation:
template <class charT> class collate;
template <class charT> class collate_byname;

// 22.4.5, date and time:
class time_base;
template <class charT, class InputIterator = istreambuf_iterator<charT>> class time_get;
template <class charT, class InputIterator = istreambuf_iterator<charT>> class time_get_byname;
template <class charT, class OutputIterator = ostreambuf_iterator<charT>> class time_put;
template <class charT, class OutputIterator = ostreambuf_iterator<charT>> class time_put_byname;

// 22.4.6, money:
class money_base;
template <class charT, class InputIterator = istreambuf_iterator<charT>> class money_get;
template <class charT, class OutputIterator = ostreambuf_iterator<charT>> class money_put;
template <class charT, bool Intl = false> class moneypunct;
template <class charT, bool Intl = false> class moneypunct_byname;

// 22.4.7, 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{241}

## 22.3 Locales

### 22.3.1 Class locale

```cpp	namespace std {
    class locale {
        public:
            // types:
            class facet;
            class id;
            typedef int category; // values assigned here are for exposition only

        } // locale
    } // std
}
```

\footnote{241) In this subclause, the type name \texttt{struct tm} is an incomplete type that is defined in \texttt{<ctime>}.}
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();  // not 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();
};

1 Class locale implements a type-safe polymorphic set of facets, indexed by facet type. In other words, a facet has a dual role: in one sense, it’s just a class interface; at the same time, it’s an index into a locale’s set of facets.

2 Access to the facets of a locale is via two function templates, use_facet<> and has_facet<>.

3 [Example: An iostream operator<< might be implemented as:\textsuperscript{242}]

\begin{verbatim}
  template <class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& s, const Date& d) {
    typename basic_ostream<charT,traits>::sentry cerberos(s);
    if (cerberos) {
      ios_base::iostate err = ios_base::iostate::goodbit;
      tm tmbuf; d.extract(tmbuf);
      use_facet< time_put<charT,ostreambuf_iterator<charT,traits> > >()(s.getloc()).put(s, s, s.fill(), err, &tmbuf, 'x');
      s.setstate(err);  // might throw
    }
\end{verbatim}

\textsuperscript{242} Notice that, in the call to put, the stream is implicitly converted to an ostreambuf_iterator<charT,traits>.

\textsuperscript{656}
In the call to use_facet<Facet>(loc), the type argument chooses a facet, making available all members of the named type. If Facet is not present in a locale, it throws the standard exception bad_cast. A C++ program can check if a locale implements a particular facet with the function template has_facet<Facet>(). User-defined facets may be installed in a locale, and used identically as may standard facets (22.4.8).

[Note: All locale semantics are accessed via use_facet<> and has_facet<>], except that:

— A member operator template operator()(const basic_string<C,T,A>&, const basic_string<C,T,A>&) is provided so that a locale may be used as a predicate argument to the standard collections, to collate strings.

— Convenient global interfaces are provided for traditional ctype functions such as isdigit() and isspace(), so that given a locale object loc a C++ program can call isspace(c,loc). (This eases upgrading existing extractors (27.7.1.2).) — 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 "*".

Whether there is one global locale object for the entire program or one global locale object per thread is implementation-defined. Implementations should provide one global locale object per thread. If there is a single global locale object for the entire program, implementations are not required to avoid data races on it (17.6.4.8).

22.3.1.1 locale types

22.3.1.1.1 Type locale::category

typedef int category;

Valid category values include the locale member bitmask elements collate, ctype, monetary, numeric, time, and messages, each of which represents a single locale category. In addition, locale member bitmask constant none is defined as zero and represents no category. And locale member bitmask constant all is defined such that the expression

(collate | ctype | monetary | numeric | time | messages | all) == all

is true, and represents the union of all categories. Further, the expression (X | Y), where X and Y each represent a single category, represents the union of the two categories.
Table 78 — Locale category facets

<table>
<thead>
<tr>
<th>Category</th>
<th>Includes facets</th>
</tr>
</thead>
<tbody>
<tr>
<td>collate</td>
<td>collate&lt;char&gt;, collate&lt;wchar_t&gt;</td>
</tr>
<tr>
<td>ctype</td>
<td>ctype&lt;char&gt;, ctype&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt&lt;char, char, mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt&lt;char16_t, char, mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt&lt;char32_t, char, mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt&lt;wchar_t, char, mbstate_t&gt;</td>
</tr>
<tr>
<td>monetary</td>
<td>moneypunct&lt;char&gt;, moneypunct&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>moneypunct&lt;char, true&gt;, moneypunct&lt;wchar_t, true&gt;</td>
</tr>
<tr>
<td></td>
<td>money_get&lt;char&gt;, money_get&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>money_put&lt;char&gt;, money_put&lt;wchar_t&gt;</td>
</tr>
<tr>
<td>numeric</td>
<td>numpunct&lt;char&gt;, numpunct&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>num_get&lt;char&gt;, num_get&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>num_put&lt;char&gt;, num_put&lt;wchar_t&gt;</td>
</tr>
<tr>
<td>time</td>
<td>time_get&lt;char&gt;, time_get&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put&lt;char&gt;, time_put&lt;wchar_t&gt;</td>
</tr>
<tr>
<td>messages</td>
<td>messages&lt;char&gt;, messages&lt;wchar_t&gt;</td>
</tr>
</tbody>
</table>

2 locale member functions expecting a category argument require one of the category values defined above, or the union of two or more such values. Such a category value identifies a set of locale categories. Each locale category, in turn, identifies a set of locale facets, including at least those shown in Table 78.

3 For any locale loc either constructed, or returned by locale::classic(), and any facet Facet shown in Table 78, has_facet<Facet>(loc) is true. Each locale member function which takes a locale::category argument operates on the corresponding set of facets.

4 An implementation is required to provide those specializations for facet templates identified as members of a category, and for those shown in Table 79.

5 The provided implementation of members of facets num_get<charT> and num_put<charT> calls use_facet<F>(l) only for facet F of types numpunct<charT> and ctype<charT>, and for locale l the value obtained by calling member getloc() on the ios_base& argument to these functions.

6 In declarations of facets, a template 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 (24.2). A template formal parameter with name C represents the set of types containing char, wchar_t, and any other implementation-defined character types that satisfy the requirements for a character on which any of the iostream components can be instantiated. A template formal parameter with name International represents the set of all possible specializations on a bool parameter.

22.3.1.1.2 Class locale::facet

```cpp
namespace std {
    class locale::facet {
        protected:
            explicit facet(size_t refs = 0);
            virtual ~facet();
            facet(const facet&) = delete;
            void operator=(const facet&) = delete;
    };
}
```

§ 22.3.1.1.2 658
Table 79 — Required specializations

<table>
<thead>
<tr>
<th>Category</th>
<th>Includes facets</th>
</tr>
</thead>
<tbody>
<tr>
<td>collate</td>
<td>collate_byname&lt;char&gt;, collate_byname&lt;wchar_t&gt;</td>
</tr>
<tr>
<td>ctype</td>
<td>ctype_byname&lt;char&gt;, ctype_byname&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt_byname&lt;char, char, mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt_byname&lt;char16_t, char, mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt_byname&lt;char32_t, char, mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt_byname&lt;wchar_t, char, mbstate_t&gt;</td>
</tr>
<tr>
<td>monetary</td>
<td>moneypunct_byname&lt;char, International&gt;</td>
</tr>
<tr>
<td></td>
<td>moneypunct_byname&lt;wchar_t, International&gt;</td>
</tr>
<tr>
<td></td>
<td>money_get&lt;C, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>money_put&lt;C, OutputIterator&gt;</td>
</tr>
<tr>
<td>numeric</td>
<td>numpunct_byname&lt;char&gt;, numpunct_byname&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>num_get&lt;C, InputIterator&gt;, num_put&lt;C, OutputIterator&gt;</td>
</tr>
<tr>
<td>time</td>
<td>time_get&lt;char, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_get_byname&lt;char, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_get&lt;wchar_t, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_get_byname&lt;wchar_t, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put&lt;char, OutputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put_byname&lt;char, OutputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put&lt;wchar_t, OutputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put_byname&lt;wchar_t, OutputIterator&gt;</td>
</tr>
<tr>
<td>messages</td>
<td>messages_byname&lt;char&gt;, messages_byname&lt;wchar_t&gt;</td>
</tr>
</tbody>
</table>

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.

22.3.1.1.3 Class locale::id

```cpp
namespace std {
    class locale::id {
```
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]

22.3.1.2 locale constructors and destructor

locale() throw();
1. Default constructor: a snapshot of the current global locale.
2. Effects: Constructs a copy of the argument last passed to locale::global(locale&), if it has been called; else, the resulting facets have virtual function semantics identical to those of locale::classic().
[Note: This constructor is commonly used as the default value for arguments of functions that take a const locale& argument. — end note]

locale(const locale& other) throw();
3. Effects: Constructs a locale which is a copy of other.

const locale& operator=(const locale& other) throw();
4. Effects: Creates a copy of other, replacing the current value.
5. Returns: *this

explicit locale(const char* std_name);
6. Effects: Constructs a locale using standard C locale names, e.g., "POSIX". The resulting locale implements semantics defined to be associated with that name.
7. Throws: runtime_error if the argument is not valid, or is null.
8. Remarks: The set of valid string argument values is "C", "", and any implementation-defined values.

explicit locale(const string& std_name);
9. Effects: The same as locale(std_name.c_str()).

locale(const locale& other, const char* std_name, category);
10. 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).
11. Throws: runtime_error if the argument is not valid, or is null.
12. Remarks: The locale has a name if and only if other has a name.

locale(const locale& other, const string& std_name, category cat);
13. Effects: The same as locale(other, std_name.c_str(), cat).
template <class Facet> locale(const locale& other, Facet* f);

Effects: Constructs a locale incorporating all facets from the first argument except that of type Facet, and installs the second argument as the remaining facet. If f is null, the resulting object is a copy of other.

Remarks: The resulting locale has no name.

locale(const locale& other, const locale& one, category cats);

Effects: Constructs a locale incorporating all facets from the first argument except those that implement cats, which are instead incorporated from the second argument.

Remarks: The resulting locale has a name if and only if the first two arguments have names.

~locale() throw();

A non-virtual destructor that throws no exceptions.

22.3.1.3 locale members

template <class Facet> locale combine(const locale& other) const;

Effects: Constructs a locale incorporating all facets from *this except for that one facet of other that is identified by Facet.

Returns: The newly created locale.

Throws: runtime_error if has_facet<Facet>(other) is false.

Remarks: The resulting locale has no name.

basic_string<char> name() const;

Returns: The name of *this, if it has one; otherwise, the string ".". If *this has a name, then locale(name().c_str()) is equivalent to *this. Details of the contents of the resulting string are otherwise implementation-defined return value of locale::name.

22.3.1.4 locale operators

bool operator==(const locale& other) const;

Returns: true if both arguments are the same locale, or one is a copy of the other, or each has a name and the names are identical; false otherwise.

bool operator!=(const locale& other) const;

Returns: The result of the expression: !(this == other).

template <class charT, class Traits, class Allocator>
bool operator()(const basic_string<charT,Traits,Allocator>& s1,
const basic_string<charT,Traits,Allocator>& s2) const;

Effects: Compares two strings according to the collate<charT> facet.

Remarks: This member operator template (and therefore locale itself) satisfies requirements for a comparator predicate template argument (Clause 25) applied to strings.

Returns: The result of the following expression:
use_facet< collate<charT> >(*this).compare
  (s1.data(), s1.data()+s1.size(), s2.data(), s2.data()+s2.size()) < 0;

[Example: A vector of strings \textit{v} can be collated according to collation rules in locale \textit{loc} simply by (25.4.1, 23.3.6):
\begin{verbatim}
std::sort(v.begin(), v.end(), loc);
\end{verbatim}
— end example]

\subsection*{22.3.1.5 locale static members \hfill [locale.statics]}

\begin{itemize}
\item static locale \texttt{global(const locale& \textit{loc});}
\end{itemize}

\begin{itemize}
\item Sets the global locale to its argument.
\item \textit{Effects:} Causes future calls to the constructor \texttt{locale()} to return a copy of the argument. If the argument has a name, does
\begin{verbatim}
  std::setlocale(LC_ALL, \textit{loc}.name().c_str());
\end{verbatim}

  otherwise, the effect on the C locale, if any, is implementation-defined. No library function other than \texttt{locale::global()} shall affect the value returned by \texttt{locale()}. [\textit{Note: see 22.6 for data race considerations when setlocale is invoked}. — end note]
\item \textit{Returns:} The previous value of \texttt{locale()}. 
\end{itemize}

\begin{itemize}
\item static \texttt{const locale& classic();}
\end{itemize}

\begin{itemize}
\item The "C" locale.
\item \textit{Returns:} A locale that implements the classic "C" locale semantics, equivalent to the value \texttt{locale("C")}.
\item \textit{Remarks:} This locale, its facets, and their member functions, do not change with time.
\end{itemize}

\subsection*{22.3.2 locale globals \hfill [locale.global.templates]}

\begin{itemize}
\item \texttt{template <class Facet> const Facet& use_facet(const locale& \textit{loc});}
\end{itemize}

\begin{itemize}
\item \textit{Requires:} \texttt{Facet} is a facet class whose definition contains the public static member \texttt{id} as defined in \ref{22.3.1.1.2}.
\item \textit{Returns:} a reference to the corresponding facet of \textit{loc}, if present.
\item \textit{Throws:} \texttt{bad_cast} if \texttt{has_facet<Facet>(\textit{loc})} is \texttt{false}.
\item \textit{Remarks:} The reference returned remains valid at least as long as any copy of \textit{loc} exists.
\end{itemize}

\begin{itemize}
\item \texttt{template <class Facet> bool has_facet(const locale& \textit{loc}) throw();}
\end{itemize}

\begin{itemize}
\item \textit{Returns:} true if the facet requested is present in \textit{loc}; otherwise false.
\end{itemize}

\subsection*{22.3.3 Convenience interfaces \hfill [locale.convenience]}

\begin{itemize}
\item \texttt{22.3.3.1 Character classification \hfill [classification]}
\end{itemize}
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);

each of these functions is F returns the result of the expression:

use_facet< ctype<charT> >(loc).is(ctype_base::F, c)

where F is the ctype_base::mask value corresponding to that function (22.4.1).

22.3.3.2 Conversions

22.3.3.2.1 Character conversions

template <class charT> charT toupper (charT c, const locale& loc);

1 Returns: use_facet<ctype<charT> >(loc).toupper(c).

template <class charT> charT tolower (charT c, const locale& loc);

2 Returns: use_facet<ctype<charT> >(loc).tolower(c).

22.3.3.2.2 String conversions

1 Class template wstring_convert performs conversions between a wide string and a byte string. It lets you specify a code conversion facet (like class template codecvt) to perform the conversions, without affecting any streams or locales. [Example: 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:

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 Wide_alloc = std::allocator<Elem>,
             class Byte_alloc = std::allocator<char> > class wstring_convert {
        public:
            typedef std::basic_string<char, char_traits<char>, Byte_alloc> byte_string;
            typedef std::basic_string<Elem, char_traits<Elem>, Wide_alloc> wide_string;
            typedef typename Codecvt::state_type state_type;
            typedef typename wide_string::traits_type::int_type int_type;

        }
The class template describes an object that controls conversions between wide string objects of class
`std::basic_string<Elem, char_traits<Elem>, Wide_alloc>` and byte string objects of class `std::basic_string<char, char_traits<char>, Byte_alloc>`. The class template defines the types `wstring` and `byte_string` as synonyms for these two types. Conversion between a sequence of `Elem` values (stored in a `wstring` 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>`.

An object of this class template stores:
- `byte_err_string` — a byte string to display on errors
- `wide_err_string` — a wide string to display on errors
- `cvtptr` — a pointer to the allocated conversion object (which is freed when the `wstring_convert` object is destroyed)
- `cvtstate` — a conversion state object
- `cvtcount` — a conversion count

```cpp
typedef std::basic_string<char> byte_string;
```

The type shall be a synonym for `std::basic_string<char>`

```cpp
size_t converted() const;
```

*Returns:* `cvtcount`.

```cpp
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();
```

```cpp
void 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);
```

```cpp
byte_string to_bytes(Elem wchar);
byte_string to_bytes(const Elem *wptr);
byte_string to_bytes(const wide_string& wstr);
byte_string to_bytes(const Elem *first, const Elem *last);
```

```cpp
size_t converted() const;
state_type state() const;
```

```cpp
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
};
```
Effects: The first member function shall convert the single-element sequence `byte` to a wide string. The second member function shall convert the null-terminated sequence beginning at `ptr` to a wide string. The third member function shall convert the sequence stored in `str` to a wide string. The fourth member function shall convert the sequence defined by the range `[first, last)` to a wide string.

In all cases:

— 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`.

Effects: The first member function shall convert the single-element sequence `wchar` to a byte string. The second member function shall convert the null-terminated sequence beginning at `wptr` to a byte string. The third member function shall convert the sequence stored in `wstr` to a byte string. The fourth member function shall convert the sequence defined by the range `[first, last)` to a byte string.

In all cases:

— If the `cvtstate` object was not constructed with an explicit value, it shall be set to its default value (the initial conversion state) before the conversion begins. Otherwise it shall be left unchanged.

— The number of input elements successfully converted shall be stored in `cvtcount`.

Returns: If no conversion error occurs, the member function shall return the converted byte string. Otherwise, if the object was constructed with a byte-error string, the member function shall return the byte-error string. Otherwise, the member function shall throw an object of class `std::range_error`.

typedef std::basic_string<Elem> wide_string;

The type shall be a synonym for `std::basic_string<Elem>`.

Effects: The `wstring_convert` constructor shall construct a new `Codecvt` object.

Effects: The `wstring_convert` constructor shall construct a new `Codecvt` object, using the specified state.

Effects: The `wstring_convert` constructor shall construct a new `Codecvt` object, using the specified byte-error and wide-error strings.

§ 22.3.3.2.2
Effects: The first constructor shall store `pcvt` in `cvtptr` and default values in `cvtstate`, `byte_err_string`, and `wide_err_string`. The second constructor shall store `pcvt` in `cvtptr`, `state` in `cvtstate`, and default values in `byte_err_string` and `wide_err_string`; moreover the stored state shall be retained between calls to `from_bytes` and `to_bytes`. The third constructor shall store new `Codecvt` in `cvtptr`, `state_type()` in `cvtstate`, `byte_err` in `byte_err_string`, and `wide_err` in `wide_err_string`.

`~wstring_convert();`

Effects: The destructor shall delete `cvtptr`.

22.3.3.2.3 Buffer conversions

Class template `wbuffer_convert` looks like a wide stream buffer, but performs all its I/O through an underlying byte stream buffer that you specify when you construct it. Like class template `wstring_convert`, it lets you specify a code conversion facet to perform the conversions, without affecting any streams or locales.

Class template `wbuffer_convert` synopsis

```cpp
namespace std {
    template<class Codecvt,
             class Elem = wchar_t,
             class Tr = std::char_traits<Elem> >
    class wbuffer_convert
        : public std::basic_streambuf<Elem, Tr> {
    public:
        typedef typename Tr::state_type state_type;
        wbuffer_convert(std::streambuf *bytebuf = 0,
                        Codecvt *pcvt = new Codecvt,
                        state_type state = state_type());
        std::streambuf *rdbuf() const;
        std::streambuf *rdbuf(std::streambuf *bytebuf);
        state_type state() const;
    private:
        std::streambuf *bufptr;       // exposition only
        Codecvt *cvtptr;              // exposition only
        state_type cvtstate;          // exposition only
    }
}
```

The class template describes a stream buffer that controls the transmission of elements of type `Elem`, whose character traits are described by the class `Tr`, to and from a byte stream buffer of type `std::streambuf`. Conversion between a sequence of `Elem` values and multibyte sequences is performed by an object of class `Codecvt<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

```cpp
state_type state() const;
```

Returns: cvtstate.

```cpp
std::streambuf *rdbuf() const;
```

Returns: bufptr.

```cpp
std::streambuf *rdbuf(std::streambuf *bytebuf);
```

Effects: stores bytebuf in bufptr.

Returns: the previous value of bufptr.

```cpp
typedef typename Codecvt::state_type state_type;
```

The type shall be a synonym for Codecvt::state_type.

```cpp
wbuffer_convert(std::streambuf *bytebuf = 0,
    Codecvt *pcvt = new Codecvt, state_type state = state_type());
```

Effects: The constructor constructs a stream buffer object, initializes bufptr to bytebuf, initializes cvtptr to pcvt, and initializes cvtstate to state.

```cpp
~wbuffer_convert();
```

Effects: The destructor shall delete cvtptr.

### 22.4 Standard locale categories

Each of the standard categories includes a family of facets. Some of these implement formatting or parsing of a datum, for use by standard or users’ iostream operators `<<` and `>>`, as members `put()` and `get()`, respectively. Each such member function takes an `ios_base&` argument whose members `flags()`, `precision()`, and `width()`, specify the format of the corresponding datum (27.5.2). Those functions which need to use other facets call its member `getloc()` to retrieve the locale imbued there. Formatting facets use the character argument `fill()` to fill out the specified width where necessary.

The `put()` members make no provision for error reporting. (Any failures of the `OutputIterator` argument must be extracted from the returned iterator.) The `get()` members take an `ios_base::iostate&` argument whose value they ignore, but set to `ios_base::failbit` in case of a parse error.

Within this clause it is unspecified whether one virtual function calls another virtual function.

#### 22.4.1 The ctype category

```cpp
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 (17.5.2.1.3).

22.4.1.1 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;

§ 22.4.1.1 668
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 78 (22.3.1.1.1), namely ctype<char> and ctype<wchar_t>, implement character classing appropriate to the implementation’s native character set.

22.4.1.1.1 ctype members

    bool is(mask m, charT c) const;
    const charT* is(const charT* low, const charT* high, mask* vec) const;

1 Returns: do_is(m,c) or do_is(low,high,vec)

    const charT* scan_is(mask m,
            const charT* low, const charT* high) const;

2 Returns: do_scan_is(m,low,high)

    const charT* scan_not(mask m,
            const charT* low, const charT* high) const;

3 Returns: do_scan_not(m,low,high)

    charT toupper(charT c) const;
    const charT* toupper(const charT* low, const charT* high) const;

4 Returns: do_toupper(c) or do_toupper(low,high)

    charT tolower(charT c) const;
    const charT* tolower(const charT* low, const charT* high) const;

5 Returns: do_tolower(c) or do_tolower(low,high)

    char widen(char c) const;
    const char* widen(const char* low, const charT* high, charT* to) const;

6 Returns: do_widen(c) or do_widen(low,high,to)

    char narrow(charT c, char dfault) const;
    const charT* narrow(const charT* low, const charT*, char dfault,
            char* to) const;

7 Returns: do_narrow(c,dfault) or do_narrow(low,high,dfault,to)

22.4.1.1.2 ctype virtual functions

    bool do_is(mask m, charT c) const;
    const charT* do_is(const charT* low, const charT* high,
            mask* vec) const;

1 Effects: Classifies a character or sequence of characters. For each argument character, identifies a value M of type ctype_base::mask. The second form identifies a value M of type ctype_base::mask for each *p where (low<=p && p<high), and places it into vec[p-low].
Returns: The first form returns the result of the expression \((M & m) \neq 0\); i.e., \texttt{true} if the character has the characteristics specified. The second form returns \texttt{high}.

\begin{verbatim}
const charT* do_scan_is(mask m,
                       const charT* low, const charT* high) const;
\end{verbatim}

Effects: Locates a character in a buffer that conforms to a classification \(m\).

Returns: The smallest pointer \(p\) in the range \([\text{low, high})\) such that \(\text{is}(m,*p)\) would return \texttt{true}; otherwise, returns \texttt{high}.

\begin{verbatim}
const charT* do_scan_not(mask m,
                        const charT* low, const charT* high) const;
\end{verbatim}

Effects: Locates a character in a buffer that fails to conform to a classification \(m\).

Returns: The smallest pointer \(p\), if any, in the range \([\text{low,high})\) such that \(\text{is}(m,*p)\) would return \texttt{false}; otherwise, returns \texttt{high}.

\begin{verbatim}
charT do_toupper(charT c) const;
const charT* do_toupper(charT* low, const charT* high) const;
\end{verbatim}

Effects: Converts a character or characters to upper case. The second form replaces each character \(*p\) in the range \([\text{low,high})\) for which a corresponding upper-case character exists, with that character.

Returns: The first form returns the corresponding upper-case character if it is known to exist, or its argument if not. The second form returns \texttt{high}.

\begin{verbatim}
charT do_tolower(charT c) const;
const charT* do_tolower(charT* low, const charT* high) const;
\end{verbatim}

Effects: Converts a character or characters to lower case. The second form replaces each character \(*p\) in the range \([\text{low,high})\) and for which a corresponding lower-case character exists, with that character.

Returns: The first form returns the corresponding lower-case character if it is known to exist, or its argument if not. The second form returns \texttt{high}.

\begin{verbatim}
charT do_widen(char c) const;
const char* do_widen(const char* low, const char* high,
                     charT* dest) const;
\end{verbatim}

Effects: Applies the simplest reasonable transformation from a \texttt{char} value or sequence of \texttt{char} values to the corresponding \texttt{charT} value or values.\(^{244}\) The only characters for which unique transformations are required are those in the basic source character set (2.3).

For any named \texttt{ctype} category with a \texttt{ctype<
\texttt{charT}> facet} \texttt{ctc} and valid \texttt{ctype_base::mask} value \(M\), \((\texttt{ctc.is}(M, c) || \neg\neg\texttt{is}(M, \texttt{do_widen}(c)))\) is \texttt{true}.\(^{245}\)

The second form transforms each character \(*p\) in the range \([\text{low,high})\), placing the result in \texttt{dest}[p-low].

Returns: The first form returns the transformed value. The second form returns \texttt{high}.

\begin{verbatim}
char do_narrow(charT c, char dfault) const;
const charT* do_narrow(const charT* low, const charT* high,
                       char dfault, char* dest) const;
\end{verbatim}

\(^{244}\) The \texttt{char} argument of \texttt{do_widen} is intended to accept values derived from character literals for conversion to the locale’s encoding.

\(^{245}\) In other words, the transformed character is not a member of any character classification that \(c\) is not also a member of.
Effects: Applies the simplest reasonable transformation from a charT value or sequence of charT values to the corresponding char value or values.

For any character c in the basic source character set (2.3) the transformation is such that

\[
do\_widen(do\_narrow(c, 0)) == c
\]

For any named ctype category with a ctype<char> facet ctc however, and ctype_base::mask value M,

\[
is(M, c) || !ctc.is(M, do\_narrow(c, dfault))
\]

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.

22.4.1.2 Class template ctype_byname

```cpp
namespace std {
    template <class charT>
    class ctype_byname : public ctype<charT> {
    public:
        typedef typename ctype<charT>::mask mask;
        explicit ctype_byname(const char*, size_t refs = 0);
        explicit ctype_byname(const string&, size_t refs = 0);
    protected:
        ~ctype_byname();
    }
}
```

22.4.1.3 ctype specializations

```cpp
namespace std {
    template <> class ctype<char>
        : public locale::facet, public ctype_base {
    public:
        typedef char char_type;

        explicit ctype(const mask* tab = 0, bool del = false,
                       size_t refs = 0);

        bool is(mask m, char c) const;
        const char* is(const char* low, const char* high, mask* vec) const;
        const char* scan_is(mask m,
                            const char* low, const char* high) const;
        const char* scan_not(mask m,
                             const char* low, const char* high) const;

        char toupper(char c) const;
        const char* toupper(char* low, const char* high) const;
        char tolower(char c) const;
        const char* tolower(char* low, const char* high) const;
    }
}
```
char widen(char c) const;
const char* widen(const char* low, const char* high, char* to) const;
char narrow(char c, char dfault) const;
const char* narrow(const char* low, const char* high, char dfault,
        char* to) const;

static locale::id id;
static const size_t table_size = implementation-defined;
const mask* table() const throw();
static const mask* classic_table() throw();

protected:
    ~ctype();
    virtual char do_toupper(char c) const;
    virtual const char* do_toupper(char* low, const char* high) const;
    virtual char do_tolower(char c) const;
    virtual const char* do_tolower(char* low, const char* high) const;
    virtual char do_widen(char c) const;
    virtual const char* do_widen(const char* low,
        const char* high, char* to) const;
    virtual char do_narrow(char c, char dfault) const;
    virtual const char* do_narrow(const char* low,
        const char* high, char dfault, char* to) const;
};

1 A specialization ctype<char> is provided so that the member functions on type char can be implemented inline.\textsuperscript{246} The implementation-defined value of member table_size is at least 256.

22.4.1.3.1 ctype<char> destructor [facet.ctype.char.dtor]

~ctype();
1 \textit{Effects:} If the constructor’s first argument was nonzero, and its second argument was true, does \texttt{delete} [] table().

22.4.1.3.2 ctype<char> members [facet.ctype.char.members]

1 In the following member descriptions, for unsigned char values \( v \) where \( v \geq \text{table\_size}, \text{table()}[v] \) is assumed to have an implementation-specific value (possibly different for each such value \( v \)) without performing the array lookup.

    explicit ctype(const mask* tbl = 0, bool del = false,
            size_t refs = 0);

2 \textit{Requires:} tbl either 0 or an array of at least table\_size elements.

3 \textit{Effects:} Passes its \texttt{refs} argument to its base class constructor.

\textsuperscript{246} 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>.
bool is(mask m, char c) const;
const char* is(const char* low, const char* high,
               mask* vec) const;

Effects: The second form, for all *p in the range [low, high), assigns into vec[p-low] the value table()[((unsigned char)*p] & m; the second form returns high.

Returns: The first form returns table()[((unsigned char)c] & m; the second form returns high.

const char* scan_is(mask m,  
                   const char* low, const char* high) const;

Returns: The smallest p in the range [low, high) such that table()[((unsigned char) *p] & m

is true.

const char* scan_not(mask m,  
                     const char* low, const char* high) const;

Returns: The smallest p in the range [low, high) such that table()[((unsigned char) *p] & m

is false.

char toupper(char c) const;
const char* toupper(const char* low, const char* high) const;

Returns: do_toupper(c) or do_toupper(low, high), respectively.

char tolower(char c) const;
const char* tolower(const char* low, const char* high) const;

Returns: do_tolower(c) or do_tolower(low, high), respectively.

char widen(char c) const;
const char* widen(const char* low, const char* high, char* to) const;

Returns: do_widen(c) or do_widen(low, high, to), respectively.

char narrow(char c, char dfault) const;
const char* narrow(const char* low, const char* high,  
                   char dfault, char* to) const;

Returns: do_narrow(c, dfault) or do_narrow(low, high, dfault, to), respectively.

const mask* table() const throw();

Returns: The first constructor argument, if it was non-zero, otherwise classic_table().

22.4.1.3.3 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.
22.4.1.3.4 ctype<char> virtual functions

```
char do_toupper(char) const;
const char* do_toupper(char* low, const char* high) const;
char do_tolower(char) const;
const char* do_tolower(char* low, const char* high) const;

virtual char do_widen(char c) const;
virtual const char* do_widen(const char* low,
    const char* high,
    char* to) const;
virtual char do_narrow(char c, char dfault) const;
virtual const char* do_narrow(const char* low,
    const char* high,
    char dfault, char* to) const;
```

These functions are described identically as those members of the same name in the `ctype` class template (22.4.1.1.1).

22.4.1.4 Class template codecvt

```
namespace std {
    class codecvt_base {
        public:
            enum result { ok, partial, error, noconv private: }
    }

template <class internT, class externT, class stateT>
    class codecvt : public locale::facet, public codecvt_base {
        public:
            typedef internT intern_type;
            typedef externT extern_type;
            typedef stateT state_type;

            explicit codecvt(size_t refs = 0);

            result out(stateT& state,
                const internT* from, const internT* from_end, const internT*& from_next,
                externT* to, externT* to_end, externT*& to_next) const;
            result unshift(stateT& state,
                externT* to, externT* to_end, externT*& to_next) const;
            result in(stateT& state,
                const externT* from, const externT* from_end, const externT*& from_next,
                internT* to, internT* to_end, internT*& to_next) const;
            int encoding() const throw();
            bool always_noconv() const throw();
            int length(stateT& state, 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,
```
The class `codecvt<internT,externT,stateT>` is for use when converting from one character encoding to another, such as from wide characters to multibyte characters or between wide character encodings such as Unicode and EUC.

The `stateT` argument selects the pair of character encodings being mapped between.

The specializations required in Table 78 (22.3.1.1.1) convert the implementation-defined native character set. `codecvt<char, char, mbstate_t>` implements a degenerate conversion; it does not convert at all. The specialization `codecvt<char16_t, char, mbstate_t>` converts between the UTF-16 and UTF-8 encoding schemes, and the specialization `codecvt<char32_t, char, mbstate_t>` converts between the UTF-32 and UTF-8 encoding schemes. `codecvt<wchar_t, char, mbstate_t>` converts between the native character sets for narrow and wide characters. Specializations on `mbstate_t` perform conversion between encodings known to the library implementor. Other encodings can be converted by specializing on a user-defined `stateT` type.

The `stateT` object can contain any state that is useful to communicate to or from the specialized `do_in` or `do_out` members.

### 22.4.1.4.1 codecvt members

- `result out(stateT& state, const internT* from, const internT* from_end, const internT*& from_next, externT* to, externT* to_end, externT*& to_next) const;
  
  **Returns:** `do_out(state, from, from_end, from_next, to, to_end, to_next)`

- `result unshift(stateT& state, externT* to, externT* to_end, externT*& to_next) const;
  
  **Returns:** `do_unshift(state, to, to_end, to_next)`

- `result in(stateT& state, const externT* from, const externT* from_end, const externT*& from_next, internT* to, internT* to_end, internT*& to_next) const;
  
  **Returns:** `do_in(state, from, from_end, from_next, to, to_end, to_next)`

- `int encoding() const throw();
  
  **Returns:** `do_encoding()`

- `bool always_noconv() const throw();
  
  **Returns:** `do_always_noconv()`
size_t max) const;

Returns: do_length(state, from, from_end, max)

int max_length() const throw();

Returns: do_max_length()

### 22.4.1.4.2 codecvt virtual functions

[locale.codecvt.virtuals]

result do_out(stateT& state,  
const internT* from, const internT* from_end, const internT*& from_next,  
externT* to, externT* to_end, externT*& to_next) const;

result do_in(stateT& state,  
const externT* from, const externT* from_end, const externT*& from_next,  
internT* to, internT* to_end, internT*& to_next) const;

1 Requires: (from<=from_end && to<=to_end) well-defined and true; state initialized, if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.

2 Effects: Translates characters in the source range [from, from_end), placing the results in sequential positions starting at destination to. Converts no more than (from_end-from) source elements, and stores no more than (to_end-to) destination elements.

   Stops if it encounters a character it cannot convert. It always leaves the from_next and to_next pointers pointing one beyond the last element successfully converted. If returns noconv, internT and externT are the same type and the converted sequence is identical to the input sequence [from, from_next). to_next is set equal to to, the value of state is unchanged, and there are no changes to the values in [to, to_end).

3 A codecvt facet that is used by basic_filebuf (27.9) shall have the property that if  
   do_out(state, from, from_end, from_next, to, to_end, to_next)

   would return ok, where from != from_end, then  
   do_out(state, from, from + 1, from_next, to, to_end, to_next)

   shall also return ok, and that if  
   do_in(state, from, from_end, from_next, to, to_end, to_next)

   would return ok, where to != to_end, then  
   do_in(state, from, from_end, from_next, to, to + 1, to_next)

   shall also return ok. [247] 

   [Note: As a result of operations on state, it can return ok or partial and set from_next == from and to_next != to. — end note]

4 Remarks: Its operations on state are unspecified. [Note: This argument can be used, for example, to maintain shift state, to specify conversion options (such as count only), or to identify a cache of seek offsets. — end note]

5 Returns: An enumeration value, as summarized in Table 80.

---

247) 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.
Table 80 — do_in/do_out result values

<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 \(\text{from\_next} = \text{from\_end}\), 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.

```cpp
def result do_unshift(stateT& state, externT* to, externT* to_end, externT*& to_next) const;
```

6 Requires: \((to \leq to\_end)\) well defined and true; state initialized, if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.

7 Effects: Places characters starting at to that should be appended to terminate a sequence when the current stateT is given by state.\(^{248}\) Stores no more than \((to\_end-to)\) destination elements, and leaves the to\_next pointer pointing one beyond the last element successfully stored.

8 Returns: An enumeration value, as summarized in Table 81.

Table 81 — 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 ((to_end-to)) destination elements was needed to terminate a sequence given the value of state</td>
</tr>
<tr>
<td>error</td>
<td>an unspecified error has occurred</td>
</tr>
<tr>
<td>noconv</td>
<td>no termination is needed for this state_type</td>
</tr>
</tbody>
</table>

```cpp
int do_encoding() const throw();
```

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\(^{249}\).

```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> returns true.

```cpp
int do_length(stateT& state, const externT* from, const externT* from_end, size_t max) const;
```

11 Requires: \((from \leq from\_end)\) 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.

---

\(^{248}\) Typically these will be characters to return the state to stateT().

\(^{249}\) 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.
Effects: The effect on the state argument is “as if” it called do_in(state, from, from_end, from, to, to+max, to) for to pointing to a buffer of at least max elements.

Returns: (from_next-from) where from_next is the largest value in the range [from,from_end] such that the sequence of values in the range [from,from_next) represents max or fewer valid complete characters of type internT. The specialization codecvt<char, char, mbstate_t>, returns the lesser of max and (from_end-from).

int do_max_length() const 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.

22.4.1.5 Class template codecvt_byname

namespace std {
    template <class internT, class externT, class stateT>
    class codecvt_byname : public codecvt<internT, externT, stateT> {
        public:
            explicit codecvt_byname(const char*, size_t refs = 0);
            explicit codecvt_byname(const string&, size_t refs = 0);
        protected:
            codecvt_byname();
    };
}

22.4.2 The numeric category

1 The classes num_get<> and num_put<> handle numeric formatting and parsing. Virtual functions are provided for several numeric types. Implementations may (but are not required to) delegate extraction of smaller types to extractors for larger types.250

2 All specifications of member functions for num_put and num_get in the subclauses of 22.4.2 only apply to the specializations required in Tables 78 and 79 (22.3.1.1.1), namely num_get<char>, num_get<wchar_t>, num_get<C, InputIterator>, num_put<char>, num_put<wchar_t>, and num_put<C, OutputIterator>. These specializations refer to the ios_base& argument for formatting specifications (22.4), and to its imbued locale for the numpunct<> facet to identify all numeric punctuation preferences, and also for the ctype<> facet to perform character classification.

3 Extractor and inserter members of the standard iostreams use num_get<> and num_put<> member functions for formatting and parsing numeric values (27.7.1.2.1, 27.7.2.6.1).

22.4.2.1 Class template num_get

namespace std {
    template <class charT, class InputIterator = istreambuf_iterator<charT> >
    class num_get : public locale::facet {
        public:
            typedef charT char_type;
            typedef InputIterator iter_type;
            explicit num_get(size_t refs = 0);
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, bool& v) const;
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, long& v) const;
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, long long& v) const;
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, unsigned short& v) const;
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, unsigned int& v) const;
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, unsigned long& v) const;
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, unsigned long long& v) const;
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, float& v) const;
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, double& v) const;
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, long double& v) const;
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, void*& v) const;

static locale::id id;

protected:

    "num_get()";
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, bool& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, long& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, long long& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, unsigned short& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, unsigned int& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, unsigned long& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, unsigned long long& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, float& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, double& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, long double& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, void*& v) const;
The facet `num_get` is used to parse numeric values from an input sequence such as an istream.

### 22.4.2.1.1 num_get members

```
iter_type get(iter_type in, iter_type end, ios_base& str, 
    ios_base::iostate& err, bool& val) const;
iter_type get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, long& val) const;
iter_type get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, long long& val) const;
iter_type get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, unsigned short& val) const;
iter_type get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, unsigned int& val) const;
iter_type get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, unsigned long& val) const;
iter_type get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, unsigned long long& val) const;
iter_type get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, float& val) const;
iter_type get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, double& val) const;
iter_type get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, long double& val) const;
iter_type get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, void*& val) const;
```

Returns: `do_get(in, end, str, err, val)`.

### 22.4.2.1.2 num_get virtual functions

```
iter_type do_get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, long& val) const;
iter_type do_get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, long long& val) const;
iter_type do_get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, unsigned short& val) const;
iter_type do_get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, unsigned int& val) const;
iter_type do_get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, unsigned long& val) const;
iter_type do_get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, unsigned long long& val) const;
iter_type do_get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, float& val) const;
iter_type do_get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, double& val) const;
iter_type do_get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, long double& val) const;
iter_type do_get(iterator in, iterator end, ios_base& str, 
    ios_base::iostate& err, void*& val) const;
```

Effects: Reads characters from `in`, interpreting them according to `str.flags()`, `use_facet<ctype<charT>>>(loc)`, and `use_facet<numpunct<charT>>>(loc)`, where `loc` is `str.getloc()`.

The details of this operation occur in three stages
Stage 1: Determine a conversion specifier

Stage 2: Extract characters from in and determine a corresponding char value for the format expected by the conversion specification determined in stage 1.

Stage 3: Store results

The details of the stages are presented below.

Stage 1: The function initializes local variables via

```cpp
define fmtflags flags = str .flags();
def define fmtflags basefield = (flags & ios_base::basefield);
def define fmtflags uppercase = (flags & ios_base::uppercase);
def define fmtflags boolalpha = (flags & ios_base::boolalpha);
```

For conversion to an integral type, the function determines the integral conversion specifier as indicated in Table 82. The table is ordered. That is, the first line whose condition is true applies.

<table>
<thead>
<tr>
<th>State</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>basefield == oct</td>
<td>%o</td>
</tr>
<tr>
<td>basefield == hex</td>
<td>%X</td>
</tr>
<tr>
<td>basefield == 0</td>
<td>%i</td>
</tr>
<tr>
<td>signed integral type</td>
<td>%d</td>
</tr>
<tr>
<td>unsigned integral type</td>
<td>%u</td>
</tr>
</tbody>
</table>

For conversions to a floating type the specifier is %g.

For conversions to void* the specifier is %p.

A length modifier is added to the conversion specification, if needed, as indicated in Table 83.

<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 \texttt{src} and \texttt{atoms} are defined as if by:

\begin{verbatim}
static const char src[] = "0123456789abcdefABCDEF+-";
char_type atoms[sizeof(src)];
use_facet<ctype<charT> >(loc).widen(src, src + sizeof(src), atoms);
\end{verbatim}

for this value of \texttt{loc}.

If \texttt{discard} is true, then if \texttt{'. '} has not yet been accumulated, then the position of the character is remembered, but the character is otherwise ignored. Otherwise, if \texttt{'. '} has already been accumulated, the character is discarded and Stage 2 terminates.

If the character is either discarded or accumulated then \texttt{in} is advanced by ++\texttt{in} and processing returns to the beginning of stage 2.

\textbf{Stage 3:} The sequence of \texttt{chars} accumulated in stage 2 (the field) is converted to a numeric value by the rules of one of the functions declared in the header \texttt{<cstdlib>}:

- For a signed integer value, the function \texttt{strtol}.
- For an unsigned integer value, the function \texttt{strtoul}.
- For a floating-point value, the function \texttt{strtof}.

The numeric value to be stored can be one of:

- zero, if the conversion function fails to convert the entire field. \texttt{ios_base::failbit} is assigned to \texttt{err}.
- the most positive representable value, if the field represents a value too large positive to be represented in \texttt{val}. \texttt{ios_base::failbit} is assigned to \texttt{err}.
- the most negative representable value or zero for an unsigned integer type, if the field represents a value too large negative to be represented in \texttt{val}. \texttt{ios_base::failbit} is assigned to \texttt{err}.
- the converted value, otherwise.

The resultant numeric value is stored in \texttt{val}.

Digit grouping is checked. That is, the positions of discarded separators is examined for consistency with \texttt{use_facet<numpunct<charT> >(loc).grouping()}. If they are not consistent then \texttt{ios_base::failbit} is assigned to \texttt{err}.

In any case, if stage 2 processing was terminated by the test for \texttt{in==end} then \texttt{err | =ios_base::eofbit} is performed.

\begin{verbatim}
iter_type do_get(iter_type in, iter_type end, ios_base& str, 
    ios_base::iostate& err, bool& val) const;
\end{verbatim}

\textbf{Effects:} If \texttt{(str.flags()\&ios_base::boolalpha)==0} then input proceeds as it would for a \texttt{long} except that if a value is being stored into \texttt{val}, the value is determined according to the following: If the value to be stored is 0 then \texttt{false} is stored. If the value is 1 then \texttt{true} is stored. Otherwise \texttt{true} is stored and \texttt{ios_base::failbit} is assigned to \texttt{err}.

Otherwise target sequences are determined “as if” by calling the members \texttt{falsename()} and \texttt{truename()} of the facet obtained by \texttt{use_facet<numpunct<charT> >(str.getloc())}. Successive characters in the range \texttt{[in,end]} (see 23.2.3) are obtained and matched against corresponding positions in the target sequences only as necessary to identify a unique match. The input iterator \texttt{in} is compared to \texttt{end}.
only when necessary to obtain a character. If a target sequence is uniquely matched, val is set to the corresponding value. Otherwise false is stored and ios_base::failbit is assigned to err.

The in iterator is always left pointing one position beyond the last character successfully matched. If val is set, then err is set to str.goodbit; or to str.eofbit if, when seeking another character to match, it is found that (in == end). If val is not set, then err is set to str.failbit; or to (str.failbit|str.eofbit) if the reason for the failure was that (in == end). [Example: For targets true: "a" and false: "abb", the input sequence "a" yields val == true and err == str.eofbit; the input sequence "abc" yields err = str.failbit, with in ending at the 'c' element. For targets true: "1" and false: "0", the input sequence "1" yields val == true and err == str.goodbit. For empty targets (""), any input sequence yields err == str.failbit. — end example]

Returns: in.

22.4.2.2 Class template num_put

namespace std {

template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
class num_put : public locale::facet {

    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, ios_base&, char_type fill, bool v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill, long v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill, long long v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill, unsigned long) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill, unsigned long long) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill, double v) const;

§ 22.4.2.2
virtual iter_type do_put(iter_type, ios_base&, char_type fill, long double v) const;
virtual iter_type do_put(iter_type, ios_base&, char_type fill, const void* v) const;
}
}

The facet `num_put` is used to format numeric values to a character sequence such as an ostream.

### 22.4.2.2.1 num_put members

```c
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)`.

### 22.4.2.2.2 num_put virtual functions

```c
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` (27.9.2) given this conversion specifier for...
printf(spec, val)

assuming that the current locale is the "C" locale.

— Stage 2: Adjust the representation by converting each char determined by stage 1 to a charT using
a conversion and values returned by members of use_facet< numpunct<charT> >(str.getloc())
— Stage 3: Determine where padding is required.
— Stage 4: Insert the sequence into the out.

Detailed descriptions of each stage follow.

Returns: out.

Stage 1: The first action of stage 1 is to determine a conversion specifier. The tables that describe
this determination use the following local variables

```cpp
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 84.

<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 85.

<table>
<thead>
<tr>
<th>State</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>floatfield == ios_base::fixed</td>
<td>%f</td>
</tr>
<tr>
<td>Floatfield == ios_base::scientific &amp;&amp; !uppercase</td>
<td>%e</td>
</tr>
<tr>
<td>floatfield == ios_base::scientific</td>
<td>%E</td>
</tr>
<tr>
<td>floatfield == (ios_base::fixed</td>
<td>ios_base::scientific) &amp;&amp; !uppercase</td>
</tr>
<tr>
<td>floatfield == (ios_base::fixed</td>
<td>ios_base::scientific)</td>
</tr>
<tr>
<td>!uppercase</td>
<td>%g</td>
</tr>
<tr>
<td>otherwise</td>
<td>%G</td>
</tr>
</tbody>
</table>
For conversions from an integral or floating-point type a length modifier is added to the conversion specifier as indicated in Table 86.

Table 86 — Length modifier

<table>
<thead>
<tr>
<th>Type</th>
<th>Length modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>1</td>
</tr>
<tr>
<td>long long</td>
<td>ll</td>
</tr>
<tr>
<td>unsigned long</td>
<td>1</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>ll</td>
</tr>
<tr>
<td>long double</td>
<td>L</td>
</tr>
<tr>
<td>otherwise</td>
<td>none</td>
</tr>
</tbody>
</table>

The conversion specifier has the following optional additional qualifiers prepended as indicated in Table 87.

Table 87 — 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>

For conversion from a floating-point type, if `floatfield` != (ios_base::fixed | ios_base::scientific), `str.precision()` is specified as precision in the conversion specification. Otherwise, no precision is specified.

For conversion from `void*` the specifier is `%p`.

The representations at the end of stage 1 consists of the char’s that would be printed by a call of `printf(s, val)` where `s` is the conversion specifier determined above.

**Stage 2:** Any character `c` other than a decimal point(.) is converted to a `charT` via `use_facet<ctype<charT>>(loc).widen( c )`

A local variable `punct` is initialized via

```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 22.4.3.1.2

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\(^{251}\) is determined according to Table 88.

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()`.

\(^{251}\) The conversion specification `%o` generates a leading 0 which is not a padding character.
Table 88 — Fill padding

<table>
<thead>
<tr>
<th>State</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjustfield == ios_base::left</td>
<td>pad after</td>
</tr>
<tr>
<td>adjustfield == ios_base::right</td>
<td>pad before</td>
</tr>
<tr>
<td>adjustfield == internal and a sign occurs in the representation</td>
<td>pad after the sign</td>
</tr>
<tr>
<td>adjustfield == internal and representation after stage 1 began with 0x or 0X</td>
<td>pad after x or X</td>
</tr>
<tr>
<td>otherwise</td>
<td>pad before</td>
</tr>
</tbody>
</table>

str.width(0) is called.

Stage 4: The sequence of charT's at the end of stage 3 are output via

```c
*out++ = c
```

iter_type do_put(iter_type out, ios_base& str, char_type fill, bool val) const;

6 Returns: If (str.flags() & ios_base::boolalpha) == 0 returns do_put(out, str, fill, (int)val), otherwise obtains a string s as if by

```c
string_type s =
    val ? use_facet<ctype<charT>>(loc).truename()
        : use_facet<ctype<charT>>(loc).falsename();
```

and then inserts each character c of s into out via *out++ = c and returns out.

22.4.3 The numeric punctuation facet

22.4.3.1 Class template numpunct

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;

            
§ 22.4.3.1
virtual string do_grouping() const;
virtual string_type do_truename() const;       // for bool
virtual string_type do_falsename() const;     // for bool
};
}

numpunct<> specifies numeric punctuation. The specializations required in Table 78 (22.3.1.1.1), namely numpunct<wchar_t> and numpunct<char>, provide classic "C" numeric formats, i.e., they contain information equivalent to that contained in the "C" locale or their wide character counterparts as if obtained by a call to widen.

The syntax for number formats is as follows, where digit represents the radix set specified by the fmtflags argument value, and thousands-sep and decimal-point are the results of corresponding numpunct<charT> members. Integer values have the format:

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-sep is as specified by do_grouping(). For parsing, if the digits portion contains no thousands-separators, no grouping constraint is applied.

22.4.3.1.1 numpunct members

char_type decimal_point() const;
1    Returns: do_decimal_point()

char_type thousands_sep() const;
2    Returns: do_thousands_sep()

string grouping() const;
3    Returns: do_grouping()

string_type truename() const;
string_type falsename() const;
4    Returns: do_truename() or do_falsename(), respectively.

22.4.3.1.2 numpunct virtual functions

char_type do_decimal_point() const;
1    Returns: A character for use as the decimal radix separator. The required specializations return '.'
or L'\n'.'.

char_type do_thousands_sep() const;

§ 22.4.3.1.2       688
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() <= 1, the number is the same as group (i-1); if (i<0 || vec[i]<=0 || vec[i]==CHAR_MAX), the size of the digit group is unlimited.

The required specializations return the empty string, indicating no grouping.

string_type do_truename() const;
string_type do_falsename() const;

Returns: A string representing the name of the boolean value true or false, respectively.

In the base class implementation these names are "true" and "false", or L"true" and L"false".

### 22.4.3.2 Class template numpunct_byname

```cpp
namespace std {
    template <class charT>
    class numpunct_byname : public numpunct<charT> {
        // this class is specialized for char and wchar_t.
        public:
            typedef charT char_type;
            typedef basic_string<charT> string_type;
            explicit numpunct_byname(const char*, size_t refs = 0);
            explicit numpunct_byname(const string&, size_t refs = 0);
        protected:
            ~numpunct_byname();
    };
}
```

### 22.4.4 The collate category

#### 22.4.4.1 Class template collate

```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;
    };
}
```

252) Thus, the string "\003" specifies groups of 3 digits each, and "3" probably indicates groups of 51 (!) digits each, because 51 is the ASCII value of "3".
The class `collate<charT>` provides features for use in the collation (comparison) and hashing of strings. A locale member function template, `operator()`, uses the collate facet to allow a locale to act directly as the predicate argument for standard algorithms (Clause 25) and containers operating on strings. The specializations required in Table 78 (22.3.1.1.1), namely `collate<char>` and `collate<wchar_t>`, apply lexicographic ordering (25.4.8).

Each function compares a string of characters *p in the range [low,high).

### 22.4.4.1.1 collate members

```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)`

### 22.4.4.1.2 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 78 (22.3.1.1.1), namely `collate<char>` and `collate<wchar_t>`, implement a lexicographical comparison (25.4.8).

```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.\(^{253}\)

```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 \(1.0/\text{numeric\_limits<unsigned\_long>::max()}.\) \[- end note\]

\(^{253}\) This function is useful when one string is being compared to many other strings.

§ 22.4.4.1.2
22.4.4.2 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();
            
    }
}

22.4.5 The time category

Templates time_get<charT, InputIterator> and time_put<charT, OutputIterator> provide date and time formatting and parsing. All specifications of member functions for time_put and time_get in the subclauses of 22.4.5 only apply to the specializations required in Tables 78 and 79 (22.3.1.1.1). Their members use their ios_base&, ios_base::iostate&, and fill arguments as described in (22.4), and the ctype<> facet, to determine formatting details.

22.4.5.1 Class template time_get

namespace std {
    namespace { 
        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 {
    
        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;
  virtual iter_type do_get_weekday(iter_type s, iter_type end, ios_base&,
      ios_base::iostate& err, tm* t) const;
  virtual iter_type do_get_monthname(iter_type s, iter_type end, ios_base&,
      ios_base::iostate& err, tm* t) const;
  virtual iter_type do_get_year(iter_type s, iter_type end, ios_base&,
      ios_base::iostate& err, tm* t) const;
  virtual iter_type do_get(iter_type s, iter_type end, ios_base& f,
      ios_base::iostate& err, tm *t, char format, char modifier) const;
};
}

1  time_get is used to parse a character sequence, extracting components of a time or date into a struct
    tm record. Each get member parses a format as produced by a corresponding format specifier to time_put<>::put. If the sequence being parsed matches the correct format, the corresponding members of the
    struct tm argument are set to the values used to produce the sequence; otherwise either an error is reported
    or unspecified values are assigned.254

2  If the end iterator is reached during parsing by any of the get() member functions, the member sets ios_base::eofbit in err.

22.4.5.1.1  time_get members

    dateorder date_order() const;

1     Returns: do_date_order()

    iter_type get_time(iter_type s, iter_type end, ios_base& str,
        ios_base::iostate& err, tm* t) const;

2     Returns: do_get_time(s, end, str, err, t)

    iter_type get_date(iter_type s, iter_type end, ios_base& str,
        ios_base::iostate& err, tm* t) const;

3     Returns: do_get_date(s, end, str, err, t)

    iter_type get_weekday(iter_type s, iter_type end, ios_base& str,
        ios_base::iostate& err, tm* t) const;
    iter_type get_monthname(iter_type s, iter_type end, ios_base& str,
        ios_base::iostate& err, tm* t) const;

4     Returns: do_get_weekday(s, end, str, err, t) or do_get_monthname(s, end, str, err, t)

    iter_type get_year(iter_type s, iter_type end, ios_base& str,
        ios_base::iostate& err, tm* t) const;

5     Returns: do_get_year(s, end, str, err, t)

254) In other words, user confirmation is required for reliable parsing of user-entered dates and times, but machine-generated
    formats can be parsed reliably. This allows parsers to be aggressive about interpreting user variations on standard formats.
iter_type get(iter_type s, iter_type end, ios_base& f,
    ios_base::iostate& err, tm *t, char format, char modifier = 0) const;

Returns: do_get(s, end, f, err, t, format, modifier)

iter_type get(iter_type s, iter_type end, ios_base& f,
    ios_base::iostate& err, tm *t, const char_type *fmt, const char_type *fmtend) const;

Requires: [fmt,fmtend) shall be a valid range.

Effects: The function starts by evaluating err = ios_base::goodbit. It then enters a loop, reading
zero or more characters from s at each iteration. Unless otherwise specified below, the loop terminates
when the first of the following conditions holds:

— The expression fmt == fmtend 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,fmtend) is
not sufficient to unambiguously determine whether the conversion specification is complete and
valid, the function evaluates err = ios_base::failbit. Otherwise, the function evaluates s =
do_get(s, end, f, err, t, format, modifier), where the value of modifier is ‘0’ when
the optional modifier is absent from the conversion specification. If err == ios_base::goodbit
holds after the evaluation of the expression, the function increments fmt to point just past the
end of the conversion specification and continues looping.
— The expression isspace(*fmt, f.getloc()) evaluates to true, in which case the function first
increments fmt until fmt == fmtend || !isspace(*fmt, f.getloc()) evaluates to true, then
advances s until s == end || !isspace(*s, f.getloc()) is true, and finally resumes looping.
— The next character read from s matches the element pointed to by fmt in a case-insensitive
comparison, in which case the function evaluates ++fmt, ++s and continues looping. Otherwise,
the function evaluates err = ios_base::failbit.

[Note: The function uses the ctype<charT> facet installed in f’s locale to determine valid whitespace
characters. It is unspecified by what means the function performs case-insensitive comparison or
whether multi-character sequences are considered while doing so.

Returns: s

22.4.5.1.2 time_get virtual functions

dateorder do_date_order() const;

Returns: An enumeration value indicating the preferred order of components for those date formats
that are composed of day, month, and year.255 Returns no_order if the date format specified by 'x'
contains other variable components (e.g., Julian day, week number, week day).

iter_type do_get_time(iter_type s, iter_type end, ios_base& str,
    ios_base::iostate& err, tm* t) const;

255) This function is intended as a convenience only, for common formats, and may return no_order in valid locales.
Effects: Reads characters starting at \texttt{s} until it has extracted those \texttt{struct tm} members, and remaining format characters, used by \texttt{time\_put<>::put} to produce the format specified by "\texttt{%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.

\begin{verbatim}
iter_type do_get_date(iter_type s, iter_type end, ios_base& str,
    ios_base::iostate& err, tm* t) const;
\end{verbatim}

Effects: Reads characters starting at \texttt{s} until it has extracted those \texttt{struct tm} members and remaining format characters used by \texttt{time\_put<>::put} to produce one of the following formats, or until it encounters an error. The format depends on the value returned by \texttt{date\_order()} as shown in Table 89.

\begin{table}[h]
\centering
\begin{tabular}{|c|l|}
\hline
\textbf{date\_order()} & \textbf{Format} \\
\hline
no\_order & "\texttt{m%d%y}" \\
\hline
dmy & "\texttt{d%m%y}" \\
\hline
mdy & "\texttt{m%d%y}" \\
\hline
ymd & "\texttt{y%m%d}" \\
\hline
ydm & "\texttt{y%d%m}" \\
\hline
\end{tabular}
\caption{\texttt{do\_get\_date} effects}
\end{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.

\begin{verbatim}
iter_type do_get_weekday(iter_type s, iter_type end, ios_base& str,
    ios_base::iostate& err, tm* t) const;
iter_type do_get_monthname(iter_type s, iter_type end, ios_base& str,
    ios_base::iostate& err, tm* t) const;
\end{verbatim}

Effects: Reads characters starting at \texttt{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 \texttt{struct tm} member accordingly.

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid name.

\begin{verbatim}
iter_type do_get_year(iter_type s, iter_type end, ios_base& str,
    ios_base::iostate& err, tm* t) const;
\end{verbatim}

Effects: Reads characters starting at \texttt{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 \texttt{t->tm\_year} member accordingly.

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid year identifier.

\begin{verbatim}
iter_type do_get(iter_type s, iter_type end, ios_base& f,
    ios_base::iostate& err, tm *t, char format, char modifier) const;
\end{verbatim}

Requires: \texttt{t} shall be dereferenceable.
Effects: The function starts by evaluating \texttt{err = ios\_base::goodbit}. It then reads characters starting at \texttt{s} until it encounters an error, or until it has extracted and assigned those \texttt{struct tm} members, and any remaining format characters, corresponding to a conversion directive appropriate for the ISO/IEC 9945 function \texttt{strptime}, formed by concatenating '%', the \texttt{modifier} character, when non-NUL, and the \texttt{format} character. When the concatenation fails to yield a complete valid directive the function leaves the object pointed to by \texttt{t} unchanged and evaluates \texttt{err |= ios\_base::failbit}. When \texttt{s == end} evaluates to true after reading a character the function evaluates \texttt{err |= ios\_base::eofbit}.

For complex conversion directives such as \texttt{%c}, \texttt{%x}, or \texttt{%X}, or directives that involve the optional modifiers \texttt{E} or \texttt{O}, when the function is unable to unambiguously determine some or all \texttt{struct tm} members from the input sequence \texttt{[s,end)}, it evaluates \texttt{err |= ios\_base::eofbit}. In such cases the values of those \texttt{struct tm} members are unspecified and may be outside their valid range.

Remark: It is unspecified whether multiple calls to \texttt{do\_get()} with the address of the same \texttt{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 \texttt{format} and \texttt{modifier}.

22.4.5.2 Class template \texttt{time\_get\_byname}

```cpp
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();
    };
}
```

22.4.5.3 Class template \texttt{time\_put}

```cpp
namespace std {
    template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
    class time_put : public locale::facet {
    public:
        typedef charT char_type;
        typedef OutputIterator iter_type;

        explicit time_put(size_t refs = 0);

        // the following is implemented in terms of other member functions.
        iter_type put(iter_type s, ios_base& f, char_type fill, const tm* tmb,
                      const charT* pattern, const charT* pat_end) const;
    
    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;
};

22.4.5.3.1 time_put members

    iter_type put(iter_type s, ios_base& str, char_type fill, const tm* t,
        const charT* pattern, const charT* pat_end) const;
    iter_type put(iter_type s, ios_base& str, char_type fill, const tm* t,
        char format, char modifier = 0) const;

1 Effects: The first form steps through the sequence from pattern to pat_end, identifying characters that are part of a format sequence. Each character that is not part of a format sequence is written to s immediately, and each format sequence, as it is identified, results in a call to do_put; thus, format elements and other characters are interleaved in the output in the order in which they appear in the pattern. Format sequences are identified by converting each character c to a char value as if by ct.narrow(c,0), where ct is a reference to ctype<charT> obtained from str.getloc(). The first character of each sequence is equal to '%', followed by an optional modifier character mod and a format specifier character spec as defined for the function strftime. If no modifier character is present, mod is zero. For each valid format sequence identified, calls do_put(s, str, fill, t, spec, mod).

2 The second form calls do_put(s, str, fill, t, format, modifier).

[Note: The fill argument may be used in the implementation-defined formats or by derivations. A space character is a reasonable default for this argument. — end note]

4 Returns: An iterator pointing immediately after the last character produced.

22.4.5.3.2 time_put virtual functions

    iter_type do_put(iter_type s, ios_base& str, char_type fill, const tm* t,
        char format, char modifier) const;

1 Effects: Formats the contents of the parameter t into characters placed on the output sequence s. Formatting is controlled by the parameters format and modifier, interpreted identically as the format specifiers in the string argument to the standard library function strftime(). except that the sequence of characters produced for those specifiers that are described as depending on the C locale are instead implementation-defined.

2 Returns: An iterator pointing immediately after the last character produced. [Note: The fill argument may be used in the implementation-defined formats or by derivations. A space character is a reasonable default for this argument. — end note]

22.4.5.4 Class template time_put_byname

256) Although the C programming language defines no modifiers, most vendors do.
257) Interpretation of the modifier argument is implementation-defined, but should follow POSIX conventions.
258) Implementations are encouraged to refer to other standards (such as POSIX) for these definitions.
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();
    }
}

22.4.6 The monetary category
[category.monetary]

1 These templates handle monetary formats. A template parameter indicates whether local or international monetary formats are to be used.

2 All specifications of member functions for money_put and money_get in the subclauses of 22.4.6 only apply to the specializations required in Tables 78 and 79 (22.3.1.1.1). Their members use their ios_base&, ios_base::iostate&, and fill arguments as described in (22.4), and the moneypunct<> and ctype<> facets, to determine formatting details.

22.4.6.1 Class template money_get
[locale.money.get]

namespace std {
    template <class charT, class InputIterator = istreambuf_iterator<charT> >
    class money_get : public locale::facet {
    public:
        typedef charT char_type;
        typedef InputIterator iter_type;
        typedef basic_string<charT> string_type;

        explicit money_get(size_t refs = 0);

        iter_type get(iter_type s, iter_type end, bool intl,
                      ios_base& f, ios_base::iostate& err,
                      long double& units) const;
        iter_type get(iter_type s, iter_type end, bool intl,
                      ios_base& f, ios_base::iostate& err,
                      string_type& digits) const;

        static locale::id id;

    protected:
        ~money_get();
        virtual iter_type do_get(iter_type, iter_type, bool, ios_base &,
                                 ios_base::iostate & err, long double & units) const;
        virtual iter_type do_get(iter_type, iter_type, bool, ios_base &,
                                 ios_base::iostate & err, string_type & digits) const;
    }
}
22.4.6.1.1 money_get members

```cpp
ter_type get(iter_type s, iter_type end, bool intl,
    ios_base& f, ios_base::iostate& err,
    long double& quant) const;
```

Returns: do_get(s, end, intl, f, err, quant)

22.4.6.1.2 money_get virtual functions

```cpp
ter_type do_get(iter_type s, iter_type end, bool intl,
    ios_base& str, ios_base::iostate& err,
    long double& units) 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 `money_base::space` or `money_base::none` appears as the last element in the format pattern, no white space is consumed. Otherwise, where `money_base::space` appears in any of the initial elements of the format pattern, at least one white space character is required. Where `money_base::none` appears in any of the initial elements of the format pattern, white space is allowed but not required. If `(str.flags() & str.showbase)` is false, the currency symbol is optional and is consumed only if other characters are needed to complete the format; otherwise, the currency symbol is required.

If the first character (if any) in the string pos returned by `mp.positive_sign()` or the string neg returned by `mp.negative_sign()` is recognized in the position indicated by `sign` in the format pattern, it is consumed and any remaining characters in the string are required after all the other format components. [Example: If showbase is off, then for a neg value of "()" and a currency symbol of "L", in "(100 L)" the "L" is consumed; but if neg is "-", the "L" in "-100 L" is not consumed. — end example] If pos or neg is empty, the sign component is optional, and if no sign is detected, the result is given the sign that corresponds to the source of the empty string. Otherwise, the character in the indicated position must match the first character of pos or neg, and the result is given the corresponding sign. If the first character of pos is equal to the first character of neg, or if both strings are empty, the result is given a positive sign.

Digits in the numeric monetary component are extracted and placed in digits, or into a character buffer buf1 for conversion to produce a value for units, in the order in which they appear, preceded
by a minus sign if and only if the result is negative. The value \texttt{units} is produced as if by\textsuperscript{259}

\begin{verbatim}
for (int i = 0; i < n; ++i)
    buf2[i] = src[find(atoms, atoms+sizeof(src), buf1[i]) - atoms];
buf2[n] = 0;
scanf(buf2, "%Lf", \&units);
\end{verbatim}

where \texttt{n} is the number of characters placed in \texttt{buf1}, \texttt{buf2} is a character buffer, and the values \texttt{src} and \texttt{atoms} are defined as if by

\begin{verbatim}
static const char src[] = "0123456789-";
charT atoms[sizeof(src)];
ct.widen(src, src + sizeof(src) - 1, atoms);
\end{verbatim}

\textit{Returns:} An iterator pointing immediately beyond the last character recognized as part of a valid monetary quantity.

\subsection*{22.4.6.2 Class template \texttt{money_put}}

\begin{verbatim}
namespace std {
    template <class charT,
        class OutputIterator = ostreambuf_iterator<charT> >
    class money_put : public locale::facet {
        public:
            typedef charT char_type;
            typedef OutputIterator iter_type;
            typedef basic_string<charT> string_type;

            explicit money_put(size_t refs = 0);

            iter_type put(iter_type s, bool intl, ios_base& f,
                char_type fill, long double units) const;
            iter_type put(iter_type s, bool intl, ios_base& f,
                char_type fill, const string_type& digits) const;
        static locale::id id;
        protected:
            "money_put()";
            virtual iter_type do_put(iter_type, bool, ios_base&, char_type fill,
                long double units) const;
            virtual iter_type do_put(iter_type, bool, ios_base&, char_type fill,
                const string_type& digits) const;
    };
}
\end{verbatim}

\subsubsection*{22.4.6.2.1 \texttt{money_put} members}

\begin{verbatim}
    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;
\end{verbatim}

\textit{Returns:} \texttt{do_put(s, intl, f, loc, quant)}

\textsuperscript{259} The semantics here are different from \texttt{ct.narrow}.

\section*{§ 22.4.6.2.1}

699
money_put virtual functions

```cpp
iter_type do_put(iter_type s, bool intl, ios_base& str,
    char_type fill, long double units) const;
iter_type do_put(iter_type s, bool intl, ios_base& str,
    char_type fill, const string_type& digits) const;
```

**Effects:** Writes characters to `s` according to the format specified by a `moneypunct<charT,Intl>` facet reference `mp` and the character mapping specified by a `ctype<charT>` facet reference `ct` obtained from the locale returned by `str.getloc()`, and `str.flags()`. The argument `units` is transformed into a sequence of wide characters as if by

```cpp
cast.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 `ct`) are used; any trailing characters (including digits appearing after a non-digit character) are ignored. Calls `str.width(0)`.

**Remarks:** The currency symbol is generated if and only if `(str.flags() & str.showbase)` is nonzero. If the number of characters generated for the specified format is less than the value returned by `str.width()` on entry to the function, then copies of `fill` are inserted as necessary to pad to the specified width. For the value `af` equal to `(str.flags() & str.adjustfield), if `af == str.internal` is true, the fill characters are placed where `none` or `space` appears in the formatting pattern; otherwise if `af == str.left` is true, they are placed after the other characters; otherwise, they are placed before the other characters.

**Returns:** An iterator pointing immediately after the last character produced.

---

### Class template moneypunct

```cpp
namespace std {
    class money_base { 
        public:
            enum part { none, space, symbol, sign, value}; 
            struct pattern { char field[4]; }; 
        ;
    ;

    template <class charT, bool International = false>
    class moneypunct : public locale::facet, public money_base { 
        public:
            typedef charT char_type;
            typedef basic_string<charT> string_type;

            explicit moneypunct(size_t refs = 0);

            charT decimal_point() const;
            charT thousands_sep() const;
            string grouping() const;
            string_type curr_symbol() const;
            string_type positive_sign() const;
        }
    }
}
```
The `moneypunct<>` facet defines monetary formatting parameters used by `money_get<>` and `money_put<>`. A monetary format is a sequence of four components, specified by a `pattern` value `p`, such that the `part` value `static_cast<part>(p.field[i])` determines the `i`th component of the format. In the `field` member of a `pattern` object, each value `symbol`, `sign`, `value`, and either `space` or `none` appears exactly once. The value `none`, if present, is not first; the value `space`, if present, is neither first nor last.

Where `none` or `space` appears, white space is permitted in the format, except where `none` appears at the end, in which case no white space is permitted. The value `space` indicates that at least one space is required at that position. Where `symbol` appears, the sequence of characters returned by `curr_symbol()` is permitted, and can be required. Where `sign` appears, the first (if any) of the sequence of characters returned by `positive_sign()` or `negative_sign()` (respectively as the monetary value is non-negative or negative) is required. Any remaining characters of the sign sequence are required after all other format components. Where `value` appears, the absolute numeric monetary value is required.

The format of the numeric monetary value is a decimal number:

\[
\text{value ::= units \[ decimal-point [ digits ]\] | 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 \[ thousands-sep units \]} \\
\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 ctype<charT>&` obtained as described in the definitions of `money_get<>` and `money_put<>`. The symbol `thousands-sep` is the character returned by

\[\text{\cite{ISO/IEC N3092 § 22.4.6.3}}\]
thousands_sep(). The space character used is the value `ct.widen(' ')`. White space characters are those characters `c` for which `ci.is(space,c)` returns true. The number of digits required after the decimal point (if any) is exactly the value returned by `frac_digits()`.

The placement of thousands-separator characters (if any) is determined by the value returned by `grouping()`, defined identically as the member `numpunct<>::do_grouping()`.

### 22.4.6.3.1 moneypunct members

```cpp
    charT decimal_point() const;
    charT thousands_sep() const;
    string grouping() const;
    string_type curr_symbol() const;
    string_type positive_sign() const;
    string_type negative_sign() const;
    int frac_digits() const;
    pattern pos_format() const;
    pattern neg_format() const;
```

1. Each of these functions `F` returns the result of calling the corresponding virtual member function `do_F()`.

### 22.4.6.3.2 moneypunct virtual functions

```cpp
    charT do_decimal_point() const;
    charT do_thousands_sep() const;
    string do_grouping() const;
    string_type do_curr_symbol() const;
    string_type do_positive_sign() const;
    string_type do_negative_sign() const;
    int do_frac_digits() const;
    pattern do_pos_format() const;
    pattern do_neg_format() const;
```

1. **Returns:** The radix separator to use in case `do_frac_digits()` is greater than zero.

2. **Returns:** The digit group separator to use in case `do_grouping()` specifies a digit grouping pattern.

3. **Returns:** A pattern defined identically as, but not necessarily equal to, the result of `numpunct<>::do_grouping()`.

4. **Returns:** A string to use as the currency identifier symbol.

5. **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.

6. **Returns:** The number of digits after the decimal radix separator, if any.

---

261) In common U.S. locales this is `. `. 
262) In common U.S. locales this is `. ,`. 
263) To specify grouping by 3s, the value is "\003" not "3". 
264) For international specializations (second template parameter `true`) this is typically four characters long, usually three letters and a space. 
265) This is usually the empty string. 
266) In common U.S. locales, this is 2.
Returns: The specializations required in Table 79 (22.3.1.1.1), namely `moneypunct<char>`, `moneypunct<wchar_t>`, `moneypunct<char,true>`, and `moneypunct<wchar_t,true>`, return an object of type `pattern` initialized to `{ symbol, sign, none, value }`.267

### 22.4.6.4 Class template `moneypunct_byname`

```cpp
namespace std {
    template <class charT, bool Intl = false>
    class moneypunct_byname : public moneypunct<charT, Intl> {
    public:
        typedef money_base::pattern pattern;
        typedef basic_string<charT> string_type;

        explicit moneypunct_byname(const char*, size_t refs = 0);
        explicit moneypunct_byname(const string&, size_t refs = 0);
    protected:
        "moneypunct_byname();"
    };
}
```

### 22.4.7 The message retrieval category

Class `messages<charT>` implements retrieval of strings from message catalogs.

#### 22.4.7.1 Class template `messages`

```cpp
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>& fn, const locale&) const;
        virtual string_type do_get(catalog, int set, int msgid,
                                    const string_type& dfault) const;
        virtual void do_close(catalog) const;
    };
}
```

267) Note that the international symbol returned by `do_curr_sym()` usually contains a space, itself; for example, "USD ".

§ 22.4.7.1
Values of type `messages_base::catalog` usable as arguments to members `get` and `close` can be obtained only by calling member `open`.

### 22.4.7.1.1 messages members

```cpp
catalog open(const basic_string<char>& name, const locale& loc) const;

Returns: `do_open(name, loc)`.
```

```cpp
string_type get(catalog cat, int set, int msgid,
                const string_type& dfault) const;

Returns: `do_get(cat, set, msgid, dfault)`.
```

```cpp
void close(catalog cat) const;

Effects: Calls `do_close(cat)`.
```

### 22.4.7.1.2 messages virtual functions

```cpp
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.

```cpp
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`.

```cpp
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.
```

### 22.4.7.2 Class template messages_byname

```cpp
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:
    "messages_byname();
};

22.4.8 Program-defined facets

A C++ program may define facets to be added to a locale and used identically as the built-in facets. To create a new facet interface, C++ programs simply derive from `locale::facet` a class containing a static member: `static locale::id id`.

[Note: The locale member function templates verify its type and storage class. — end note]

Example: Traditional global localization is still easy:

```c++
#include <iostream>
#include <locale>

int main(int argc, char** argv) {
    using namespace std;
    locale::global(locale("")); // set the global locale
    locale::classic(); // imbue it on all the std streams
    cin.imbue(locale());
    cout.imbue(locale());
    cerr.imbue(locale());
    wcin.imbue(locale());
    wcout.imbue(locale());
    wchar_t* wargv = reinterpret_cast<wchar_t*>(argv);
    wcerr.imbue(locale());

    return MyObject(argc, argv).doit();
}
```

— end example]

Example: Greater flexibility is possible:

```c++
#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.

// file: Date.h
#include <iosfwd>
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:

```cpp
#include <locale>
namespace My {
  using namespace std;
  class JCategory : 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;
typeof 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;
typeof numpunctbyname<char> cnumpunct;
class BoolNames : public cnumpunct {
protected:
    string do_truename() const { return "Oui Oui!"; }
    string do_falsename() const { return "Mais Non!"; }
    ~BoolNames() {} 
public:
    BoolNames(const char* name) : cnumpunct(name) {} 
    
};

int main(int argc, char** argv) {
    using namespace std;
    // make the user's preferred locale, except for...
    locale loc(locale(""), new My::BoolNames(""));
cout.imbue(loc);
cout << boolalpha << "Any arguments today? " << (argc > 1) << endl;
return 0;
}
22.5 Standard code conversion facets

The header `<codecvt>` provides code conversion facets for various character encodings.

Header `<codecvt>` synopsis

```cpp
namespace std {
    enum codecvt_mode {
        consume_header = 4,
        generate_header = 2,
        little_endian = 1
    };

    template<class Elem, unsigned long Maxcode = 0x10ffff,
        codecvt_mode Mode = (codecvt_mode)0>
    class codecvt_utf8
        : public codecvt<Elem, char, mbstate_t> {
        // 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`.
- `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.

For the facet `codecvt_utf8`:

- The facet shall convert between UTF-8 multibyte sequences and UCS2 or UCS4 (depending on the size of `Elem`) 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 **Elem**) 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.

### 22.6 C Library Locales

1 Table 90 describes header `<clocale>`.

<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>

2 The contents are the same as the Standard C library header `<locale.h>`.

3 Calls to the function **setlocale** may introduce a data race (17.6.4.8) with other calls to **setlocale** or with calls to the functions listed in Table 91.

<table>
<thead>
<tr>
<th>Function(s)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>fprintf</td>
<td>isprint, iswctype, localeconv, tolower</td>
</tr>
<tr>
<td>fscanf</td>
<td>ispunct, iswdigit, mblen, toupper</td>
</tr>
<tr>
<td>isalnum</td>
<td>ispunct, iswgraph, mbstowcs, tolower</td>
</tr>
<tr>
<td>isalpha</td>
<td>isspace, islower, mbtowc, towupper</td>
</tr>
<tr>
<td>isblank</td>
<td>isupper, iswprint, setlocale, wcscoll</td>
</tr>
<tr>
<td>iscntrl</td>
<td>iswalnum, iswspace, strcoll, wcstod</td>
</tr>
<tr>
<td>isdigit</td>
<td>iswalpha, iswupper, strerror, wcstombs</td>
</tr>
<tr>
<td>isgraph</td>
<td>iswblank, iswxdigit, strtod, wcxfrm</td>
</tr>
<tr>
<td>islower</td>
<td>iswcnctr1, isxdigit, strxfrm, wctomb</td>
</tr>
</tbody>
</table>

**See also:** ISO C Clause 7.4.

§ 22.6    709
23 Containers library

23.1 General

This Clause describes components that C++ programs may use to organize collections of information.

The following subclauses describe container requirements, and components for sequence containers and associative containers, as summarized in Table 92.

Table 92 — Containers library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.2 Requirements</td>
<td>&lt;array&gt;</td>
</tr>
<tr>
<td>23.3 Sequence containers</td>
<td>&lt;deque&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;forwardlist&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>23.4 Associative containers</td>
<td>&lt;map&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;set&gt;</td>
</tr>
<tr>
<td>20.5 bitset</td>
<td>&lt;bitset&gt;</td>
</tr>
<tr>
<td>23.5 Unordered associative containers</td>
<td>&lt;unordered_map&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;unordered_set&gt;</td>
</tr>
</tbody>
</table>

23.2 Container requirements

23.2.1 General container requirements

Containers are objects that store other objects. They control allocation and deallocation of these objects through constructors, destructors, insert and erase operations.

All of the complexity requirements in this Clause are stated solely in terms of the number of operations on the contained objects. [Example: the copy constructor of type `vector <vector<int>>` has linear complexity, even though the complexity of copying each contained `vector<int>` is itself linear. — end example]

For the components affected by this subclause that declare an `allocator_type`, objects stored in these components shall be constructed using the `allocator_traits<allocator_type>::construct` function and destroyed using the `allocator_traits<allocator_type>::destroy` function (20.9.4.2). These functions are called only for the container’s element type, not for internal types used by the container. [Note: this means, for example, that a node-based container might need to construct nodes containing aligned buffers and call `construct` to place the element into the buffer. — end note]

In Tables 93 and 94, X denotes a container class containing objects of type T, a and b denote values of type X, u denotes an identifier, r denotes a non-const value of type X, and rv denotes a non-const rvalue of type X.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::value_<code>-</code>_type</td>
<td>T</td>
<td>Requires: T is Destructible</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>X::reference</td>
<td>lvalue of T</td>
<td></td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>X::const_<code>-</code>_reference</td>
<td>const lvalue 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::const_iterator</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>X::const_<code>-</code>_iterator</td>
<td>constant iterator type whose value type is T</td>
<td>any iterator category except output iterator</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>X::difference_type</td>
<td>signed integral type</td>
<td>is identical to the difference type of X::iterator and X::const_iterator</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>X::size_type</td>
<td>unsigned integral type</td>
<td>size_type can represent any non-negative value of difference_type</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>X u;</td>
<td></td>
<td>post: u.empty() returns true</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>X()</td>
<td></td>
<td>post: X().empty() returns true</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>X(a)</td>
<td></td>
<td>Requires: T is CopyConstructible. post: a == X(a).</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>X u(a)</td>
<td></td>
<td>Requires: T is CopyConstructible.</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>X u = a;</td>
<td></td>
<td>post: u == a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X u(rv)</td>
<td></td>
<td>Requires: T is MoveConstructible. (Note B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X u = rv</td>
<td></td>
<td>Requires: T is MoveConstructible. post: u shall be equal to the value that rv had before this construction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a = rv</td>
<td>X&amp;</td>
<td>All existing elements of a are either move assigned to or destroyed</td>
<td>a shall be equal to the value that rv had before this assignment</td>
<td>linear</td>
</tr>
</tbody>
</table>
Table 93 — Container requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&amp;a)-&gt;~X()</td>
<td>void</td>
<td>note: the destructor is applied to every element of a; all the memory is deallocated.</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>a.begin()</td>
<td>iterator;</td>
<td>constant</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>const_iterator for constant a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a.end()</td>
<td>iterator;</td>
<td>constant</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>const_iterator for constant a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a.cbegin()</td>
<td>const_iterator</td>
<td>const_cast&lt;X const&amp;&gt;(a).begin();</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>a.cend()</td>
<td>const_iterator</td>
<td>const_cast&lt;X const&amp;&gt;(a).end();</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>a == b</td>
<td>convertible to bool</td>
<td>Requires: T is EqualityComparable</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>a != b</td>
<td>convertible to bool</td>
<td>Equivalent to: !(a == b)</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>a.swap(b)</td>
<td>void</td>
<td>exchanges the contents of a and b</td>
<td>(Note A)</td>
<td></td>
</tr>
<tr>
<td>swap(a, b)</td>
<td>void</td>
<td>a.swap(b)</td>
<td>(Note A)</td>
<td></td>
</tr>
<tr>
<td>r = a</td>
<td>X&amp;</td>
<td>post: r == a.</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>a.size()</td>
<td>size_type</td>
<td>distance(a.begin(), a.end())</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>a.max_size()</td>
<td>size_type</td>
<td>distance(begin(), end()) for the largest possible container</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>a.empty()</td>
<td>convertible to bool</td>
<td>a.begin() == a.end()</td>
<td>constant</td>
<td></td>
</tr>
</tbody>
</table>

Notes: the algorithms equal() and lexicographical_compare() are defined in Clause 25. Those entries marked “(Note A)” or “(Note B)” have linear complexity for array and have constant complexity for all other standard containers.

5 The member function size() returns the number of elements in the container. The number of elements is
defined by the rules of constructors, inserts, and erases.

6 begin() returns an iterator referring to the first element in the container. end() returns an iterator which is the past-the-end value for the container. If the container is empty, then begin() == end();

7 In the expressions
   i == j
   i != j
   i < j
   i <= j
   i >= j
   i > j
   i - j

where i and j denote objects of a container’s iterator type, either or both may be replaced by an object of the container’s const_iterator type referring to the same element with no change in semantics.

8 Unless otherwise specified, all containers defined in this clause obtain memory using an allocator (see 20.2.5). Copy constructors for these container types obtain an allocator by calling allocator_traits<allocator_type>::select_on_container_copy_construction on their first parameters. Move constructors obtain an allocator by move construction of the allocator belonging to the container being moved. Such move construction of the allocator shall not exit via an exception. All other constructors for these container types take an Allocator argument (20.2.5), an allocator whose value type is the same as the container’s value type. [Note: if an invocation of a constructor uses the default value of an optional allocator argument, then the Allocator type must support value initialization. — end note] A copy of this allocator is used for any memory allocation performed, by these constructors and by all member functions, during the lifetime of each container object or until the allocator is replaced. The allocator may be replaced only via assignment or swap(). Allocator replacement is performed by copy assignment, move assignment, or swapping of the allocator only if allocator_traits<allocator_type>::propagate_on_container_copy_assignment::value, allocator_traits<allocator_type>::propagate_on_container_move_assignment::value, or allocator_traits<allocator_type>::propagate_on_container_swap::value is true within the implementation of the corresponding container operation. The behavior of a call to a container’s swap function is undefined unless the objects being swapped have allocators that compare equal or allocator_traits<allocator_type>::propagate_on_container_swap::value is true. In all container types defined in this Clause, the member get_allocator() returns a copy of the allocator used to construct the container or, if that allocator has been replaced, a copy of the most recent replacement.

9 The expression a.swap(b), for containers a and b of a standard container type other than array, shall exchange the values of a and b without invoking any move, copy, or swap operations on the individual container elements. Any Compare, Pred, or Hash objects belonging to a and b shall be swappable and shall be exchanged by unqualified calls to non-member swap. If allocator_traits<allocator_type>::propagate_on_container_swap::value is true, then the allocators of a and b shall also be exchanged using an unqualified call to non-member swap. Otherwise, they shall not be swapped, and the behavior is undefined unless a.get_allocator() == b.get_allocator(). Every iterator referring to an element in one container before the swap shall refer to the same element in the other container after the swap. It is unspecified whether an iterator with value a.end() before the swap will have value b.end() after the swap.

10 If the iterator type of a container belongs to the bidirectional or random access iterator categories (24.2), the container is called reversible and satisfies the additional requirements in Table 94.
Table 94 — Reversible container requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::reverse_iterator</td>
<td>iterator type whose value type is T</td>
<td>reverse_iterator&lt;iterator&gt;</td>
<td>compile time</td>
</tr>
<tr>
<td>X::const_reverse_iterator</td>
<td>iterator type whose value type is const T</td>
<td>reverse_iterator&lt;const_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(); const_reverse_iterator</td>
<td>const_cast&lt;X const&amp;&gt;(a).rbegin();</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>a.crend(); const_reverse_iterator</td>
<td>const_cast&lt;X const&amp;&gt;(a).rend();</td>
<td>constant</td>
<td></td>
</tr>
</tbody>
</table>

11 Unless otherwise specified (see 23.2.4.1, 23.2.5.1, 23.3.2.3, and 23.3.6.4) all container types defined in this Clause meet the following additional requirements:

— if an exception is thrown by an `insert()` function while inserting a single element, that function has no effects.
— if an exception is thrown by a `push_back()` or `push_front()` function, that function has no effects.
— no `erase()`, `clear()`, `pop_back()` or `pop_front()` function throws an exception.
— no copy constructor or assignment operator of a returned iterator throws an exception.
— no `swap()` function throws an exception.
— no `swap()` function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped. [Note: the `end()` iterator does not refer to any element, so it may be invalidated. — end note]

12 Unless otherwise specified (either explicitly or by defining a function in terms of other functions), invoking a container member function or passing a container as an argument to a library function shall not invalidate iterators to, or change the values of, objects within that container.

13 Table 95 lists operations that are provided for some types of containers but not others. Those containers for which the listed operations are provided shall implement the semantics described in Table 95 unless otherwise stated.
Table 95 — Optional container operations

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt; b</td>
<td>convertible to bool</td>
<td>lexicographical_compare(a.begin(), a.end(), b.begin(), b.end())</td>
<td>pre: &lt; is defined for values of T. &lt; is a total ordering relationship.</td>
<td>linear</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>convertible to bool</td>
<td>b &lt; a</td>
<td></td>
<td>linear</td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>convertible to bool</td>
<td>!(a &gt; b)</td>
<td></td>
<td>linear</td>
</tr>
<tr>
<td>a &gt;= b</td>
<td>convertible to bool</td>
<td>!(a &lt; b)</td>
<td></td>
<td>linear</td>
</tr>
</tbody>
</table>

14 All of the containers defined in this Clause and in (21.4) except array meet the additional requirements of an allocator-aware container, as described in Table 96.

15 The descriptions of the requirements of the type T in this section use the terms CopyConstructible, MoveConstructible, constructible from *i, and constructible from args. These terms are equivalent to the following expression using the appropriate arguments:

\[
\text{allocator_traits<allocator_type>::construct(x.get_allocator(), q, args...);}
\]

where x is a non-const lvalue of some container type X and q has type X::value_type*. [Example: The container is going to move construct a T, so will call:

\[
\text{allocator_traits<allocator_type>::construct(x.get_allocator(), q, std::move(t));}
\]

The default implementation of construct will call:

\[
\text{::new (q) T(std::forward<T>(t)); // where forward is the same as move here, cast to rvalue}
\]

But the allocator author may override the above definition of construct and do the construction of T by some other means. — end example]

16 In Table 96, X denotes an allocator-aware container class with a value_type of T using allocator of type A. u denotes a variable, a and b denote non-const lvalues of type X, t denotes an lvalue or a const rvalue of type X, rv denotes a non-const rvalue of type X, m is a value of type A, and Q is an allocator type.

Table 96 — Allocator-aware container requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocator_ - A type</td>
<td>Requires:allocator_ type::value_type is the same as X::value_type.</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>get_ - A allocator()</td>
<td></td>
<td>constant</td>
<td></td>
</tr>
</tbody>
</table>
Table 96 — Allocator-aware container requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X()</td>
<td></td>
<td>Requires: A is DefaultConstructible.</td>
<td>constant</td>
</tr>
<tr>
<td>X u;</td>
<td></td>
<td>post: u.empty() returns true, u.get_allocator() == A()</td>
<td></td>
</tr>
<tr>
<td>X(m)</td>
<td></td>
<td>post: u.empty() returns true, u.get_allocator() == m</td>
<td>constant</td>
</tr>
<tr>
<td>X(t, m)</td>
<td></td>
<td>Requires: T is CopyConstructible. post: u == t, get_allocator() == m</td>
<td>linear</td>
</tr>
<tr>
<td>X(rv)</td>
<td></td>
<td>Requires: move construction of A shall not exit via an exception. post: u shall have the same elements as rv had before this construction; the value of get_allocator() shall be the same as the value of rv.get_allocator() before this construction.</td>
<td>constant</td>
</tr>
<tr>
<td>X(rv, m)</td>
<td></td>
<td>Requires: T is MoveConstructible.</td>
<td>constant if m == rv.get_allocator(); otherwise linear</td>
</tr>
<tr>
<td>X u(rv, m);</td>
<td></td>
<td>post: u shall have the same elements, or copies of the elements, that rv had before this construction, get_allocator() == m</td>
<td></td>
</tr>
<tr>
<td>a = t X&amp;</td>
<td></td>
<td>Requires: T is CopyConstructible and CopyAssignable. post: a == t</td>
<td>linear</td>
</tr>
<tr>
<td>a = rv X&amp;</td>
<td></td>
<td>Requires: If allocator_traits&lt;allocator_type&gt;::propagate_on_container_move_assignment::value is false, T is MoveConstructible and MoveAssignable. All existing elements of a are either move assigned to or destroyed. post: a shall be equal to the value that rv had before this assignment.</td>
<td>linear</td>
</tr>
</tbody>
</table>
Table 96 — Allocator-aware container requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.swap(b)</td>
<td>void</td>
<td>exchanges the contents of a and b</td>
<td>constant</td>
</tr>
</tbody>
</table>

23.2.2 Container data races

1 For purposes of avoiding data races (17.6.4.8), implementations shall consider the following functions to be const: begin, end, rbegin, rend, front, back, data, find, lower_bound, upper_bound, equal_range, at and, except in associative or unordered associative containers, operator[].

2 Notwithstanding (17.6.4.8), implementations are required to avoid data races when the contents of the contained object in different elements in the same sequence, excepting `vector<bool>`, are modified concurrently.

3 \[Note: For a `vector<int>` x with a size greater than one, x[1] = 5 and *x.begin() = 10 can be executed concurrently without a data race, but x[0] = 5 and *x.begin() = 10 executed concurrently may result in a data race. As an exception to the general rule, for a `vector<bool>` y, y[0] = true may race with y[1] = true. \] — end note

23.2.3 Sequence containers

1 A sequence container organizes a finite set of objects, all of the same type, into a strictly linear arrangement. The library provides four basic kinds of sequence containers: `vector`, `forward_list`, `list`, and `deque`. In addition, `array` is provided as a sequence container which provides limited sequence operations because it has a fixed number of elements. The library also provides container adaptors that make it easy to construct abstract data types, such as stacks or queues, out of the basic sequence container kinds (or out of other kinds of sequence containers that the user might define).

2 The sequence containers offer the programmer different complexity trade-offs and should be used accordingly. `vector` or `array` is the type of sequence container that should be used by default. `list` or `forward_list` should be used when there are frequent insertions and deletions from the middle of the sequence. `deque` is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence.

3 In Tables 97 and 98, 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, il designates an object of type initializer_list<value_type>, n denotes a value of X::size_type, p denotes a valid const iterator to a, q 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.

§ 23.2.3
Table 97 — Sequence container requirements (in addition to container)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X(n, t)$</td>
<td></td>
<td>$Requires$: $T$ shall be CopyConstructible. post: $distance(begin(), end()) == n$ Constructs a sequence container with $n$ copies of $t$.</td>
</tr>
<tr>
<td>$X a(n, t)$</td>
<td></td>
<td>$Requires$: $T$ shall be CopyConstructible and CopyAssignable. Inserting $n$ copies of $t$ before $p$.</td>
</tr>
<tr>
<td>$X(i, j)$</td>
<td></td>
<td>$Requires$: $T$ shall be constructible from $*i$. For vector, if the iterator does not meet the forward iterator requirements (24.2.5), $T$ shall also be MoveConstructible. Each iterator in the range $[i, j)$ shall be dereferenced exactly once. post: $distance(begin(), end()) == distance(i, j)$ Constructs a sequence container equal to the range $[i, j)$.</td>
</tr>
<tr>
<td>$X(il)$</td>
<td>$X&amp;$</td>
<td>$Requires$: $T$ is CopyConstructible and CopyAssignable. Assigns the range $[il.begin(), il.end())$ into a. All existing elements of a are either assigned to or destroyed. $Returns$: $*this$.</td>
</tr>
<tr>
<td>$a = il;$</td>
<td></td>
<td>$Requires$: $T$ is constructible from $args$. For vector and deque, $T$ is also MoveConstructible and MoveAssignable. $Effects$: Inserts an object of type $T$ constructed with $std::forward&lt;Args&gt;(args)...$ before $p$.</td>
</tr>
<tr>
<td>$a.emplace(p, args);$</td>
<td>iterator</td>
<td>$Requires$: $T$ shall be CopyConstructible. For vector and deque, $T$ shall also be CopyAssignable. $Effects$: Inserts a copy of $t$ before $p$.</td>
</tr>
<tr>
<td>$a.insert(p,t)$</td>
<td>iterator</td>
<td>$Requires$: $T$ shall be MoveConstructible. For vector and deque, $T$ shall also be MoveAssignable. $Effects$: Inserts a copy of $rv$ before $p$.</td>
</tr>
<tr>
<td>$a.insert(p,rv)$</td>
<td>iterator</td>
<td>$Requires$: $T$ shall be CopyConstructible and CopyAssignable. Inserts $n$ copies of $t$ before $p$.</td>
</tr>
</tbody>
</table>
Table 97 — Sequence container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.insert(p, i, j)</code></td>
<td>iterator</td>
<td>Requires: T shall be constructible from *i. For vector, if the iterator does not meet the forward iterator requirements (24.2.5), T shall also be MoveConstructible and MoveAssignable. Each iterator in the range [i, j) shall be dereferenced exactly once. Pre: i and j are not iterators into a. Inserts copies of elements in [i, j) before p.</td>
</tr>
<tr>
<td><code>a.insert(p, il);</code></td>
<td>iterator</td>
<td>a.insert(p, il.begin(), il.end()).</td>
</tr>
<tr>
<td><code>a.erase(q)</code></td>
<td>iterator</td>
<td>Requires: For vector and deque, T shall be MoveAssignable. Effects: Erases the element pointed to by q.</td>
</tr>
<tr>
<td><code>a.erase(q1, q2)</code></td>
<td>iterator</td>
<td>Requires: For vector and deque, T shall be MoveAssignable. Effects: Erases the elements in the range [q1, q2).</td>
</tr>
<tr>
<td><code>a.clear()</code></td>
<td>void</td>
<td>Destroys all elements in a. Invalidates all references, pointers, and iterators referring to the elements of a and may invalidate the past-the-end iterator. Pre: It returns true.</td>
</tr>
<tr>
<td><code>a.assign(i, j)</code></td>
<td>void</td>
<td>Requires: T shall be constructible from *i and assignable from *i. For vector, if the iterator does not meet the forward iterator requirements (24.2.5), T shall also be MoveConstructible. Each iterator in the range [i, j) shall be dereferenced exactly once. Pre: i, j are not iterators into a. Replaces elements in a with a copy of [i, j).</td>
</tr>
<tr>
<td><code>a.assign(il)</code></td>
<td>void</td>
<td>a.assign(il.begin(), il.end()).</td>
</tr>
<tr>
<td><code>a.assign(n, t)</code></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.insert(p, rv)` points to the copy of rv inserted into a.

8 The iterator returned from `a.insert(p, n, t)` points to the copy of the first element inserted into a, or p if n == 0.

9 The iterator returned from `a.insert(p, i, j)` points to the copy of the first element inserted into a, or p if i == j.
The iterator returned from `a.insert(p, i1)` points to the copy of the first element inserted into `a`, or `p` if `i1` is empty.

The iterator returned from `a.emplace(p, args)` points to the new element constructed from args into `a`.

The iterator returned from `a.erase(q)` points to the element immediately following `q` prior to the element being erased. If no such element exists, `a.end()` is returned.

The iterator returned by `a.erase(q1,q2)` points to the element pointed to by `q2` prior to any elements being erased. If no such element exists, `a.end()` is returned.

For every sequence container defined in this Clause and in Clause 21:

- If the constructor
  
  ```
  template <class InputIterator>
  X(InputIterator first, InputIterator last,
      const allocator_type& alloc = allocator_type())
  ```

  is called with a type `InputIterator` that does not qualify as an input iterator, then the constructor will behave as if the overloaded constructor:

  ```
  X(size_type, const value_type& = value_type(),
      const allocator_type& = allocator_type())
  ```

  were called instead, with the arguments `static_cast<size_type>(first), last and alloc`, respectively.

- If the member functions of the forms:
  
  ```
  template <class InputIterator>
  // such as insert()
  rt fx1(iterator p, InputIterator first, InputIterator last);
  ```

  ```
  template <class InputIterator>
  // such as append(), assign()
  rt fx2(InputIterator first, InputIterator last);
  ```

  ```
  template <class InputIterator>
  // such as replace()
  rt fx3(iterator i1, iterator i2, InputIterator first, InputIterator last);
  ```

  are called with a type `InputIterator` that does not qualify as an input iterator, then these functions will behave as if the overloaded member functions:

  ```
  rt fx1(iterator, size_type, const value_type&);
  ```

  ```
  rt fx2(size_type, const value_type&);
  ```

  ```
  rt fx3(iterator, iterator, size_type, const value_type&);
  ```

  were called instead, with the same arguments.

In the previous paragraph the alternative binding will fail if `first` is not implicitly convertible to `X::size_type` or if `last` is not implicitly convertible to `X::value_type`.

The extent to which an implementation determines that a type cannot be an input iterator is unspecified, except that a minimum integral types shall not qualify as input iterators.

Table 98 lists operations that are provided for some types of sequence containers but not others. An implementation shall provide these operations for all container types shown in the “container” column, and shall implement them so as to take amortized constant time.
Table 98 — Optional sequence container operations

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Container</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.front()</td>
<td>reference; const_reference</td>
<td>*a.begin()</td>
<td>basic_, string, array, deque, forward_, list, vector</td>
</tr>
<tr>
<td></td>
<td>for constant a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a.back()</td>
<td>reference; const_reference</td>
<td>{ auto tmp = a.end(); --tmp; return *tmp; }</td>
<td>basic_, string, array, deque, list, vector</td>
</tr>
<tr>
<td></td>
<td>for constant a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a.emplace_front(args)</td>
<td>void</td>
<td>Prepends an object of type T</td>
<td>deque, forward_, list, list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>constructed with std::forward&lt;Arg&gt;(args)....</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requires: T shall be constructible from args.</td>
<td></td>
</tr>
<tr>
<td>a.emplace_back(args)</td>
<td>void</td>
<td>Appends an object of type T</td>
<td>basic_, list, vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>constructed with std::forward&lt;Args&gt;(args)....</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requires: T shall be constructible from args.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For vector, T shall also be MoveConstructible.</td>
<td></td>
</tr>
<tr>
<td>a.push_front(t)</td>
<td>void</td>
<td>Prepends a copy of t.</td>
<td>deque,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requires: T shall be CopyConstructible.</td>
<td>forward_, list, list</td>
</tr>
<tr>
<td>a.push_front(rv)</td>
<td>void</td>
<td>Prepends a copy of rv.</td>
<td>deque,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requires: T shall be MoveConstructible.</td>
<td>forward_, list, list</td>
</tr>
<tr>
<td>a.push_back(t)</td>
<td>void</td>
<td>Appends a copy of t.</td>
<td>basic_, string, deque, list, vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requires: T shall be CopyConstructible.</td>
<td></td>
</tr>
<tr>
<td>a.push_back(rv)</td>
<td>void</td>
<td>Appends a copy of rv.</td>
<td>basic_, string, deque, list, vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requires: T shall be MoveConstructible.</td>
<td></td>
</tr>
<tr>
<td>a.pop_front()</td>
<td>void</td>
<td>Destroys the first element.</td>
<td>deque,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requires: a.empty() shall be false.</td>
<td>forward_, list, list</td>
</tr>
<tr>
<td>a.pop_back()</td>
<td>void</td>
<td>Destroys the last element.</td>
<td>basic_, string, deque, list, vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requires: a.empty() shall be false.</td>
<td></td>
</tr>
</tbody>
</table>
Table 98 — Optional sequence container operations (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Container</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[n]</td>
<td>reference; const_reference</td>
<td>*(a.begin() + n) for constant a</td>
<td>basic_string, array, deque, vector</td>
</tr>
<tr>
<td>a.at(n)</td>
<td>reference; const_reference</td>
<td>*(a.begin() + n) for constant a</td>
<td>basic_string, array, deque, vector</td>
</tr>
</tbody>
</table>

18 The member function at() provides bounds-checked access to container elements. at() throws out_of_range if \( n \geq a.size() \).

23.2.4 Associative containers

1 Associative containers provide fast retrieval of data based on keys. The library provides four basic kinds of associative containers: set, multiset, map and multimap.

2 Each associative container is parameterized on Key and an ordering relation Compare that induces a strict weak ordering (25.4) 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.

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 \( \text{comp} \), \( \text{comp}(k1, k2) = \text{false} \) \&\& \( \text{comp}(k2, k1) = \text{false} \). For any two keys \( k1 \) and \( k2 \) in the same container, calling \( \text{comp}(k1, k2) \) 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>.

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. [Note: iterator and const_iterator have identical semantics in this case, and iterator is convertible to const_iterator. Users can avoid violating the One Definition Rule by always using const_iterator in their function parameter lists. — end note]

7 The associative containers meet all the requirements of Allocator-aware containers (23.2.1), except that for map and multimap, the requirements placed on value_type in Table 93 apply instead to key_type and mapped_type. [Note: For example, key_type and mapped_type are sometimes required to be CopyAssignable even though the associated value_type, pair<const key_type, mapped_type>, is not CopyAssignable. — end note]

8 In Table 99, X denotes an associative container class, a denotes a value of X, a_uniq denotes a value of X when X supports unique keys, a_eq denotes a value of X when X supports multiple keys, u denotes an
identifier, 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, il designates an object of type initializer_list<value_type>, t denotes a value of X::value_type, k denotes a value of X::key_type and c denotes a value of type X::key_compare. 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 99 — Associative container requirements (in addition to container)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/Note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::key_type</td>
<td>Key</td>
<td>Requires: Key is Destructible.</td>
<td>compile time</td>
</tr>
<tr>
<td>mapped_type</td>
<td>T</td>
<td>Requires: T is Destructible.</td>
<td>compile time</td>
</tr>
<tr>
<td>X::key_compare</td>
<td>Compare</td>
<td>is the same as key_compare for set and multiset; is an ordering relation on pairs induced by the first component (i.e., Key) for map and multimap.</td>
<td>compile time</td>
</tr>
<tr>
<td>X()</td>
<td>X a;</td>
<td>Requires: key_compare is CopyConstructible.</td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td>X a;</td>
<td>Requires: key_compare is DefaultConstructible.</td>
<td>constant</td>
</tr>
<tr>
<td>X(i,j,c)</td>
<td>X a(i,j,c);</td>
<td>Requires: key_compare is CopyConstructible. value_type is constructible from *i. Effects: Constructs an empty container and inserts elements from the range [i, j) into it; uses c as a comparison object.</td>
<td>N log N in general (N has the value distance(i, j)); linear if [i, j) is sorted with value_comp()</td>
</tr>
</tbody>
</table>
### Table 99 — Associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Expression</strong></td>
<td><strong>Return type</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>pre-/post-condition</strong></td>
<td></td>
</tr>
<tr>
<td>X(i,j)</td>
<td></td>
<td><em>Requires:</em> key_compare is</td>
<td>same as above</td>
</tr>
<tr>
<td>X a(i,j);</td>
<td></td>
<td>DefaultConstructible.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>value_type is constructible from *i</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Effects:</em> Same as above, but</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>uses Compare() as a comparison object</td>
<td></td>
</tr>
<tr>
<td>X ila;</td>
<td></td>
<td>Same as X ila.begin(), ila.end()</td>
<td>same as X ila.begin(), ila.end()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Requires:</em> T is CopyConstructible and CopyAssignable.</td>
<td>NlogN in general (where $N$ has the value ila.size() + a.size()); linear if [ila.begin(),ila.end()] is sorted with value_comp().</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Effects:</em> Assigns the range [ila.begin(),ila.end()] into a. All existing elements of a are either assigned to or destroyed.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>returns the comparison object out of which a was constructed.</td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>returns an object of value_compare constructed out of the comparison object</td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Requires:</em> T shall be constructible from args.</td>
<td>logarithmic</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Effects:</em> 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></td>
</tr>
</tbody>
</table>
Table 99 — Associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_eq.emplace(args)</td>
<td>iterator</td>
<td>Requires: T shall be constructible from args. Effects: 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>a.emplace_hint(p, args)</td>
<td>iterator</td>
<td>equivalent to aemplace( 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)bool&gt;</td>
<td>pair&lt;iterator, bool&gt;</td>
<td>Requires: If t is a non-const value expression, T shall be MoveConstructible; otherwise, T shall be CopyConstructible. Effects: Inserts t if and only if there is no element in the container with key equivalent to the key of t. The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of t.</td>
<td>logarithmic</td>
</tr>
</tbody>
</table>
Table 99 — Associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_.eq.insert(t)</td>
<td>iterator</td>
<td>Returns: If ( t ) is a non-const rvalue expression, ( T ) shall be <code>MoveConstructible</code>; otherwise, ( T ) shall be <code>CopyConstructible</code>. Effects: Inserts ( t ) and returns the iterator pointing to the newly inserted element. If a range containing elements equivalent to ( t ) exists in a.eq, ( t ) is inserted at the end of that range.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>a.insert(p, t)</td>
<td>iterator</td>
<td>Returns: If ( t ) is a non-const rvalue expression, ( T ) shall be <code>MoveConstructible</code>; otherwise, ( T ) shall be <code>CopyConstructible</code>. Effects: Inserts ( t ) if and only if there is no element with key equivalent to the key of ( t ) in containers with unique keys; always inserts ( t ) in containers with equivalent keys. always returns the iterator pointing to the element with key equivalent to the key of ( t ). ( t ) is inserted as close as possible to the position just prior to ( p ).</td>
<td>logarithmic in general, but amortized constant if ( t ) is inserted right before ( p ).</td>
</tr>
<tr>
<td>a.insert(i, j)</td>
<td>void</td>
<td>Requires: ( T ) shall be constructible from (*i). pre: ( i, j ) are not iterators into a. inserts each element from the range ([i,j)) if and only if there is no element with key equivalent to the key of that element in containers with unique keys; always inserts that element in containers with equivalent keys.</td>
<td>(N \log(a.size() + N) (N \text{ has the value } distance(i, j)))</td>
</tr>
<tr>
<td>a.insert(il)</td>
<td>void</td>
<td>Equivalent to a.insert(il.begin(), il.end()).</td>
<td></td>
</tr>
</tbody>
</table>
### Table 99 — Associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>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(a.size()) + a.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(a.size()) + N</code> where <code>N</code> has the value <code>distance(q1, q2)</code></td>
</tr>
<tr>
<td><code>a.clear()</code></td>
<td><code>void</code></td>
<td><code>a.erase(a.begin(),a.end())</code> post: <code>a.empty()</code> returns <code>true</code></td>
<td>linear in <code>a.size()</code></td>
</tr>
<tr>
<td><code>a.find(k)</code></td>
<td><code>iterator;</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>const_iterator for constant a.</code></td>
<td></td>
<td></td>
<td></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(a.size()) + a.count(k)</code></td>
</tr>
<tr>
<td><code>a.lower_bound(k)</code></td>
<td><code>iterator;</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>const_iterator for constant a.</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a.upper_bound(k)</code></td>
<td><code>iterator;</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>const_iterator for constant a.</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a.equal_range(k)</code></td>
<td><code>pair&lt;iterator, iterator&gt;;</code></td>
<td>equivalent to <code>make_pair(a.lower_bound(k), a.upper_bound(k))</code>.</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>pair&lt;const_iterator, const_iterator&gt; for constant a.</code></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

9 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.

10 The fundamental property of iterators of associative containers is that they iterate through the containers in the non-descending order of keys where non-descending is defined by the comparison that was used to construct them. For any two dereferenceable iterators `i` and `j` such that `distance from i to j` is positive,
value_comp(*j, *i) == false

11 For associative containers with unique keys the stronger condition holds,
value_comp(*i, *j) != false.

12 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.

23.2.4.1 Exception safety guarantees

1 For associative containers, no clear() function throws an exception. erase(k) does not throw an exception
unless that exception is thrown by the container’s Compare object (if any).

2 For associative containers, if an exception is thrown by any operation from within an insert() function
inserting a single element, the insert() function has no effect.

3 For associative containers, no swap function throws an exception unless that exception is thrown by the
swap of the container’s Compare object (if any).

23.2.5 Unordered associative containers

1 Unordered associative containers provide an ability for fast retrieval of data based on keys. The worst-case
complexity for most operations is linear, but the average case is much faster. The library provides four
unordered associative containers: unordered_set, unordered_map, unordered_multiset, and unordered_multimap.

2 Unordered associative containers conform to the requirements for Containers (23.2), except that the expres-
sions a == b and a != b have different semantics than for the other container types.

3 Each unordered associative container is parameterized by Key, by a function object type Hash that meets
the Hash requirements (20.2.4) and acts as a hash function for argument values of type Key, and by a binary
predicate Pred that induces an equivalence relation on values of type Key. Additionally, unordered_map
and unordered_multimap associate an arbitrary mapped type T with the Key.

4 A hash function is a function object that takes a single argument of type Key and returns a value of type
std::size_t.

5 Two values k1 and k2 of type Key are considered equivalent if the container’s key_equal function object
returns true when passed those values. If k1 and k2 are equivalent, the hash function shall return the same
value for both. [Note: thus, when an unordered associative container is instantiated with a non-default Pred
parameter it usually needs a non-default Hash parameter as well. — end note]

6 An unordered associative container supports unique keys if it may contain at most one element for each
key. Otherwise, it supports equivalent keys. unordered_set and unordered_map support unique keys.
unordered_multiset and unordered_multimap support equivalent keys. In containers that support equivalent
keys, elements with equivalent keys are adjacent to each other in the iteration order of the container.
Thus, although the absolute order of elements in an unordered container is not specified, its elements are
grouped into equivalent-key groups such that all elements of each group have equivalent keys. Mutating
operations on unordered containers shall preserve the relative order of elements within each equivalent-key
group unless otherwise specified.

7 For unordered_set and unordered_multiset the value type is the same as the key type. For unordered_map
and unordered_multimap it is std::pair<const Key, T>.
The elements of an unordered associative container are organized into buckets. Keys with the same hash code appear in the same bucket. The number of buckets is automatically increased as elements are added to an unordered associative container, so that the average number of elements per bucket is kept below a bound. Rehashing invalidates iterators, changes ordering between elements, and changes which buckets elements appear in, but does not invalidate pointers or references to elements. For **unordered_multiset** and **unordered_multimap**, rehashing preserves the relative ordering of equivalent elements.

The unordered associative containers meet all the requirements of Allocator-aware containers (23.2.1), except that for **unordered_map** and **unordered_multimap**, the requirements placed on **value_type** in Table 93 apply instead to **key_type** and **mapped_type**. [Note: For example, **key_type** and **mapped_type** are sometimes required to be **CopyAssignable** even though the associated **value_type**.pair<**const key_type**, **mapped_type**>, is not **CopyAssignable**. — end note]

In Table 100: **X** is an unordered associative container class, **a** is an object of type **X**, **b** is a possibly const object of type **X**, **a_uniq** is an object of type **X** when **X** supports unique keys, **a_eq** is an object of type **X** when **X** supports equivalent keys, **i** and **j** are input iterators that refer to **value_type**, **[i, j)** is a valid range, **p** and **q2** are valid const iterators to **a**, **q** and **q1** are valid dereferenceable const iterators to **a**, **[q1, q2)** is a valid range in **a**, **il** designates an object of type **initializer_list<value_type>**, **t** is a value of type **X::value_type**, **k** is a value of type **key_type**, **hf** is a possibly const value of type **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>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>X::key_type</strong></td>
<td>Key</td>
<td><strong>Requires</strong>: <strong>Key</strong> shall be <strong>Destructible</strong>.</td>
<td>compile time</td>
</tr>
<tr>
<td><strong>X::mapped_type</strong> (unordered_map and unordered_multimap only)</td>
<td><strong>T</strong></td>
<td><strong>Requires</strong>: <strong>T</strong> is <strong>Destructible</strong>.</td>
<td>compile time</td>
</tr>
<tr>
<td><strong>X::hasher</strong></td>
<td>Hash</td>
<td><strong>Hash</strong> shall be a unary function object type such that the expression <strong>hf(k)</strong> has type <strong>std::size_t</strong>.</td>
<td>compile time</td>
</tr>
<tr>
<td><strong>X::key_equal</strong></td>
<td>Pred</td>
<td><strong>Pred</strong> shall be a binary predicate that takes two arguments of type <strong>Key</strong>. <strong>Pred</strong> is an equivalence relation.</td>
<td>compile time</td>
</tr>
<tr>
<td><strong>X::local_iterator</strong></td>
<td>An iterator type whose category, value type, difference type, and pointer and reference types are the same as <strong>X::iterator</strong>'s.</td>
<td>A <strong>local_iterator</strong> 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><strong>X::const_local_iterator</strong></td>
<td>An iterator type whose category, value type, difference type, and pointer and reference types are the same as <strong>X::const_iterator</strong>'s.</td>
<td>A <strong>const_local_iterator</strong> 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>
</tbody>
</table>
### Table 100 — Unordered associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X(n, hf, eq)$</td>
<td>$X$</td>
<td>Requires: $\text{hasher}$ and $\text{key_equal}$ are CopyConstructible. Effects: Constructs an empty container with at least $n$ buckets, using $hf$ as the hash function and $eq$ as the key equality predicate.</td>
<td>$\mathcal{O}(n)$</td>
</tr>
<tr>
<td>$X a(n, hf, eq)$</td>
<td>$X$</td>
<td>Requires: $\text{copy_constructible}(\text{hasher})$ and $\text{copy_constructible}(\text{key_equal})$. Effects: Constructs an empty container with at least $n$ buckets, using $hf$ as the hash function and $eq$ as the key equality predicate.</td>
<td>$\mathcal{O}(n)$</td>
</tr>
<tr>
<td>$X(n, hf)$</td>
<td>$X$</td>
<td>Requires: $\text{hasher}$ is CopyConstructible and $\text{key_equal}$ is DefaultConstructible. Effects: Constructs an empty container with at least $n$ buckets, using $hf$ as the hash function and $\text{key_equal}()$ as the key equality predicate.</td>
<td>$\mathcal{O}(n)$</td>
</tr>
<tr>
<td>$X a(n, hf)$</td>
<td>$X$</td>
<td>Requires: $\text{copy_constructible}(\text{hasher})$ and $\text{copy_constructible}(\text{key_equal})$. Effects: Constructs an empty container with at least $n$ buckets, using $hf$ as the hash function and $\text{key_equal}()$ as the key equality predicate.</td>
<td>$\mathcal{O}(n)$</td>
</tr>
<tr>
<td>$X(n)$</td>
<td>$X$</td>
<td>Requires: $\text{hasher}$ and $\text{key_equal}$ are DefaultConstructible. Effects: Constructs an empty container with at least $n$ buckets, using $\text{hasher}()$ as the hash function and $\text{key_equal}()$ as the key equality predicate.</td>
<td>$\mathcal{O}(n)$</td>
</tr>
<tr>
<td>$X a(n)$</td>
<td>$X$</td>
<td>Requires: $\text{hasher}$ and $\text{key_equal}$ are DefaultConstructible. Effects: Constructs an empty container with an unspecified number of buckets, using $\text{hasher}()$ as the hash function and $\text{key_equal}()$ as the key equality predicate.</td>
<td>constant</td>
</tr>
<tr>
<td>$X()$</td>
<td>$X$</td>
<td>Requires: $\text{hasher}$ and $\text{key_equal}$ are DefaultConstructible. Effects: Constructs an empty container with an unspecified number of buckets, using $\text{hasher}()$ as the hash function and $\text{key_equal}()$ as the key equality predicate.</td>
<td>$\mathcal{O}(N)$ (Average case) $\mathcal{O}(N^2)$ (Worst case)</td>
</tr>
<tr>
<td>$X(i, j, n, hf, eq)$</td>
<td>$X$</td>
<td>Requires: $\text{hasher}$ and $\text{key_equal}$ are CopyConstructible. value_type is constructible from *i. Effects: Constructs an empty container with at least $n$ buckets, using $hf$ as the hash function and $eq$ as the key equality predicate, and inserts elements from $[i, j)$ into it.</td>
<td>$\mathcal{O}(N) (N \text{ is distance}(i, j))$, worst case $\mathcal{O}(N^2)$</td>
</tr>
</tbody>
</table>

§ 23.2.5
Table 100 — Unordered associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X(i, j, n, hf)) (Xa(i, j, n, hf))</td>
<td>(X)</td>
<td>\textit{Requires:} (\text{hasher}) is (\text{CopyConstructible}) and (\text{key_equal}) is (\text{DefaultConstructible}). (\text{value_type}) is constructible from (*i). \text{Effects:} Constructs an empty container with at least (n) buckets, using (hf) as the hash function and (\text{key_equal()}) as the key equality predicate, and inserts elements from ([i, j)) into it.</td>
<td>Average case: (O(N)) ((N) is (\text{distance}(i, j))), worst case: (O(N^2))</td>
</tr>
<tr>
<td>(X(i, j, n)) (Xa(i, j, n))</td>
<td>(X)</td>
<td>\textit{Requires:} (\text{hasher}) and (\text{key_equal}) are (\text{DefaultConstructible}). (\text{value_type}) is constructible from (*i). \text{Effects:} Constructs an empty container with at least (n) buckets, using (\text{hasher()}) as the hash function and (\text{key_equal()}) as the key equality predicate, and inserts elements from ([i, j)) into it.</td>
<td>Average case: (O(N)) ((N) is (\text{distance}(i, j))), worst case: (O(N^2))</td>
</tr>
<tr>
<td>(X(i, j)) (Xa(i, j))</td>
<td>(X)</td>
<td>\textit{Requires:} (\text{hasher}) and (\text{key_equal}) are (\text{DefaultConstructible}). (\text{value_type}) is constructible from (*i). \text{Effects:} Constructs an empty container with an unspecified number of buckets, using (\text{hasher()}) as the hash function and (\text{key_equal()}) as the key equality predicate, and inserts elements from ([i, j)) into it.</td>
<td>Average case: (O(N)) ((N) is (\text{distance}(i, j))), worst case: (O(N^2))</td>
</tr>
<tr>
<td>(X(il))</td>
<td>(X)</td>
<td>Same as (X(il\text{.begin()}, il\text{.end}())).</td>
<td>Same as (X(il\text{.begin()}, il\text{.end}())).</td>
</tr>
<tr>
<td>(X(b)) (Xa(b))</td>
<td>(X)</td>
<td>Copy constructor. In addition to the requirements of Table 93, copies the hash function, predicate, and maximum load factor.</td>
<td>Average case linear in (b\text{.size()}), worst case quadratic.</td>
</tr>
</tbody>
</table>
Table 100 — Unordered associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = b</code></td>
<td><code>X&amp;</code></td>
<td>Copy assignment operator. In addition to the requirements of Table 93, copies the hash function, predicate, and maximum load factor.</td>
<td>Average case linear in <code>b.size()</code>, worst case quadratic.</td>
</tr>
<tr>
<td><code>a = il</code></td>
<td><code>X&amp;</code></td>
<td>Requires: <code>T</code> is CopyConstructible and CopyAssignable. Effects: Assigns the range <code>[il.begin(),il.end())</code> into <code>a</code>. All existing elements of <code>a</code> are either assigned to or destroyed.</td>
<td>Same as <code>a = X(il)</code>.</td>
</tr>
<tr>
<td><code>b.hash_function()</code></td>
<td>hasher</td>
<td>Returns <code>b</code>’s hash function.</td>
<td>constant</td>
</tr>
<tr>
<td><code>b.key_eq()</code></td>
<td>key_equal</td>
<td>Returns <code>b</code>’s key equality predicate.</td>
<td>constant</td>
</tr>
<tr>
<td><code>a_uniq.emplace(args)</code></td>
<td>pair&lt;iterator, bool&gt;</td>
<td>Requires: <code>T</code> shall be constructible from <code>args</code>. Effects: 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>iterator</td>
<td>Requires: <code>T</code> shall be constructible from <code>args</code>. Effects: 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>
</tbody>
</table>
Table 100 — Unordered associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
</table>
| aemplace_hint(p, args)       | iterator    | Requires: T shall be constructible from args.  
Effects: Equivalent to aemplace(
std::forward<Args>(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. | Average case $O(1)$, worst case $O(a.size())$. |
| a_uniq.insert(t)             | pair<iterator, bool> | Requires: If t is a non-const rvalue expression, T shall be MoveConstructible; otherwise, T shall be CopyConstructible.  
Effects: Inserts t if and only if there is no element in the container with key equivalent to the key of t. The bool component of the returned pair indicates whether the insertion takes place, and the iterator component points to the element with key equivalent to the key of t. | Average case $O(1)$, worst case $O(a_uniq.size())$. |
| a_eq.insert(t)               | iterator    | Requires: If t is a non-const rvalue expression, T shall be MoveConstructible; otherwise, T shall be CopyConstructible.  
Effects: Inserts t, and returns an iterator pointing to the newly inserted element. | Average case $O(1)$, worst case $O(a_eq.size())$. |
Table 100 — Unordered associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.insert(q, t)</td>
<td>iterator</td>
<td>Requires: If t is a non-const value expression, T shall be MoveConstructible; otherwise, T shall be CopyConstructible. Effects: Equivalent to a.insert(t). Return value is an iterator pointing to the element with the key equivalent to that of t. The iterator q is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.</td>
<td>Average case $O(1)$, worst case $O(a.size())$.</td>
</tr>
<tr>
<td>a.insert(i, j)</td>
<td>void</td>
<td>Requires: T shall be constructible from *i. Pre: i and j are not iterators in a. Equivalent to a.insert(t) for each element in [i,j].</td>
<td>Average case $O(N)$, where N is distance(i, j). Worst case $O(N * (a.size()) + N)$.</td>
</tr>
<tr>
<td>a.insert(il)</td>
<td>void</td>
<td>Same as a.insert(il.begin(), il.end()).</td>
<td>Same as a.insert-il.begin(), il.end()).</td>
</tr>
<tr>
<td>a.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 $O(a.count(k))$. Worst case $O(a.size())$.</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 $O(1)$, worst case $O(a.size())$.</td>
</tr>
<tr>
<td>a.erase(q1, q2)</td>
<td>iterator</td>
<td>Erases all elements in the range [q1, q2). Return value is the iterator immediately following the erased elements prior to the erasure.</td>
<td>Average case linear in distance(q1, q2), worst case $O(a.size())$.</td>
</tr>
<tr>
<td>a.clear()</td>
<td>void</td>
<td>Erases all elements in the container. Post: a.empty() returns true</td>
<td>Linear.</td>
</tr>
<tr>
<td>b.find(k)</td>
<td>iterator; const_iterator for const b.</td>
<td>Returns an iterator pointing to an element with key equivalent to k, or b.end() if no such element exists.</td>
<td>Average case $O(1)$, worst case $O(b.size())$.</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>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 \text{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>Pre: ( b.bucket_count() &gt; 0 ). Returns the index of the bucket in which elements with keys equivalent to ( k ) would be found, if any such element existed. Post: the return value shall be in the range ([0, b.bucket_count())).</td>
<td>Constant</td>
</tr>
<tr>
<td>b.bucket_size(n)</td>
<td>size_type</td>
<td>Pre: ( n ) shall be in the range ([0, b.bucket_count())). Returns the number of elements in the ( n )th bucket.</td>
<td>( \mathcal{O}(b.bucket_size(n)) )</td>
</tr>
<tr>
<td>b.begin(n)</td>
<td>local_iterator; const_local_iterator for const b.</td>
<td>Pre: ( n ) shall be in the range ([0, b.bucket_count())). ( b.begin(n) ) returns an iterator referring to the first element in the bucket. If the bucket is empty, then ( b.begin(n) == b.end(n) ).</td>
<td>Constant</td>
</tr>
<tr>
<td>b.end(n)</td>
<td>local_iterator; const_local_iterator for const b.</td>
<td>Pre: ( n ) shall be in the range ([0, b.bucket_count())). ( b.end(n) ) returns an iterator which is the past-the-end value for the bucket.</td>
<td>Constant</td>
</tr>
<tr>
<td>b.cbegin(n)</td>
<td>const_local_iterator</td>
<td>Pre: ( n ) shall be in the range ([0, b.bucket_count())). Note: ([b.cbegin(n), b.cend(n)]) is a valid range containing all of the elements in the ( n )th bucket.</td>
<td>Constant</td>
</tr>
<tr>
<td>b.cend(n)</td>
<td>const_local_iterator</td>
<td>Pre: ( n ) shall be in the range ([0, b.bucket_count())).</td>
<td>Constant</td>
</tr>
</tbody>
</table>
Table 100 — Unordered associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b.load_factor()</code></td>
<td>float</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>float</td>
<td>Returns a positive number that the container attempts to keep the load factor less than or equal to. The container automatically increases the number of buckets as necessary to keep the load factor below this number.</td>
<td>Constant</td>
</tr>
<tr>
<td><code>a.max_load_factor(z)</code></td>
<td>void</td>
<td>Pre: <code>z</code> shall be positive. May change the container’s maximum load load factor, using <code>z</code> as a hint.</td>
<td>Constant</td>
</tr>
<tr>
<td><code>a.reserve(n)</code></td>
<td>void</td>
<td>Same as <code>a.rehash(ceil(n / a.max_load_factor()))</code>.</td>
<td>Average case linear in <code>a.size()</code>, worst case quadratic.</td>
</tr>
</tbody>
</table>

11 Two unordered containers `a` and `b` compare equal if `a.size() == b.size()` and, for every equivalent-key group `[Ea1,Ea2)` obtained from `a.equal_range(Ea1)`, there exists an equivalent-key group `[Eb1,Eb2)` obtained from `b.equal_range(Ea1)`, such that `distance(Ea1, Ea2) == distance(Eb1, Eb2)` and `is_permutation(Ea1, Ea2, Eb1)` returns true. For unordered_set and unordered_map, the complexity of `operator==` (i.e., the number of calls to the `==` operator of the `value_type`, to the predicate returned by `key_equal()`, and to the hasher returned by `hash_function()`) is proportional to `N` in the average case and to `N^2` in the worst case, where `N` is `a.size()`. For unordered_multiset and unordered_multimap, the complexity of `operator==` is proportional to \(\sum E_i^2\) in the average case and to \(N^2\) in the worst case, where `N` is `a.size()`, and `E_i` is the size of the `i`th equivalent-key group in `a`. However, if the respective elements of each corresponding pair of equivalent-key groups `Ea_i` and `Eb_i` are arranged in the same order (as is commonly the case, e.g., if `a` and `b` are unmodified copies of the same container), then the average-case complexity for unordered_multiset and unordered_multimap becomes proportional to `N` (but worst-case complexity remains \(O(N^2)\), e.g., for a pathologically bad hash function). The behavior of a program that uses `operator==` or `operator!=` on unordered containers is undefined unless the `Hash` and `Pred` function objects respectively have the same behavior for both containers and the equality comparison operator for `Key` is a refinement\(^\text{268}\) of the partition into equivalent-key groups produced by `Pred`.

12 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.

---

\(^{268}\) Equality comparison is a refinement of partitioning if no two objects that compare equal fall into different partitions.
The insert members shall not affect the validity of references to container elements, but may invalidate all
iterators to the container. The erase members shall invalidate only iterators and references to the erased
elements.

The insert members shall not affect the validity of iterators if \((N+n) < z \times B\), 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.

### 23.2.5.1 Exception safety guarantees

For unordered associative containers, no `clear()` function throws an exception. `erase(k)` does not throw
an exception unless that exception is thrown by the container’s `Hash` or `Pred` object (if any).

For unordered associative containers, if an exception is thrown by any operation other than the container’s
hash function from within an `insert()` 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 swap of the container’s `Hash` or `Pred` object (if any).

For unordered associative containers, if an exception is thrown from within a `rehash()` function other than
by the container’s hash function or comparison function, the `rehash()` function has no effect.

### 23.3 Sequence containers

1 Headers `<array>`, `<deque>`, `<forward_list>`, `<list>`, `<queue>`, `<stack>`, and `<vector>`.

**Header `<array>` synopsis**

```cpp
namespace std {
    #include <initializer_list>

    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 <size_t I, class T> class tuple_element;
    template <class T, size_t N> struct tuple_size<array<T,N> >;
    template <size_t I, class T, size_t N> struct tuple_element<I, array<T,N> >;
    template <size_t I, class T, size_t N> T& get(array<T,N>&); // Get element
    template <size_t I, class T, size_t N> const T& get(const array<T,N>&); // Get element
```

§ 23.3 737
const T& get(const array<T, N>&);
}

Header <deque> synopsis

namespace std {
    #include <initializer_list>

    template <class T, class Allocator = allocator<T> > class deque;
    template <class T, class Allocator>
        bool operator==(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
    template <class T, class Allocator>
        bool operator<(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
    template <class T, class Allocator>
        bool operator!=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
    template <class T, class Allocator>
        bool operator>(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
    template <class T, class Allocator>
        bool operator>=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
    template <class T, class Allocator>
        bool operator<=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
    template <class T, class Allocator>
        void swap(deque<T,Allocator>& x, deque<T,Allocator>& y);
}

Header <forward_list> synopsis

namespace std {
    #include <initializer_list>

    template <class T, class Allocator = allocator<T> > class forward_list;
    template <class T, class Allocator>
        bool operator==(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
    template <class T, class Allocator>
        bool operator<(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
    template <class T, class Allocator>
        bool operator!=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
    template <class T, class Allocator>
        bool operator>(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
    template <class T, class Allocator>
        bool operator>=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
    template <class T, class Allocator>
        bool operator<=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
    template <class T, class Allocator>
        void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>& y);
}

Header <list> synopsis

namespace std {
    #include <initializer_list>

    template <class T, class Allocator = allocator<T> > class list;
    template <class T, class Allocator>
        bool operator==(const list<T,Allocator>& x, const list<T,Allocator>& y);
    template <class T, class Allocator>
        bool operator<(const list<T,Allocator>& x, const list<T,Allocator>& y);
    template <class T, class Allocator>
        bool operator!=(const list<T,Allocator>& x, const list<T,Allocator>& y);
    template <class T, class Allocator>
        bool operator>(const list<T,Allocator>& x, const list<T,Allocator>& y);
    template <class T, class Allocator>
        bool operator>=(const list<T,Allocator>& x, const list<T,Allocator>& y);
    template <class T, class Allocator>
        bool operator<=(const list<T,Allocator>& x, const list<T,Allocator>& y);
    template <class T, class Allocator>
        void swap(list<T,Allocator>& x, list<T,Allocator>& y);
}
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, list<T,Allocator>& y);

} // namespace std

Header <queue> synopsis

namespace std {

#include <initializer_list>

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, queue<T, Container>& y);

template <class T, class Container = vector<T>,
class Compare = less<typename Container::value_type> >
class priority_queue;

template <class T, class Container, class Compare>
void swap(priority_queue<T, Container, Compare>& x, priority_queue<T, Container, Compare>& y);

} // namespace std

Header <stack> synopsis

namespace std {

#include <initializer_list>

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, stack<T, Container>& y);

} // namespace std

§ 23.3
bool operator>=(const stack<T, Container>& x, const stack<T, Container>& y);

template <class T, class Container>
bool operator<=(const stack<T, Container>& x, const stack<T, Container>& y);

template <class T, class Container>
void swap(stack<T, Container>& x, stack<T, Container>& y);

} // Header <vector> synopsis

namespace std {
    #include <initializer_list>

    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 Allocator> class vector<bool, Allocator>;

    // Hash support
    template <class T> struct hash;
    template <class Allocator> struct hash<vector<bool, Allocator> >;

} // namespace std

23.3.1 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

    array a<T, N> = { initializer-list };

where initializer-list is a comma separated list of up to N elements whose types are convertible to T.

An array satisfies all of the requirements of a container and of a reversible container (23.2), except that a default constructed array object is not empty and that swap does not have constant complexity. An array satisfies some of the requirements of a sequence container (23.2.3). Descriptions are provided here only for operations on array that are not described in one of these tables and for operations where there is additional semantic information.

namespace std {
    template <class T, size_t N>
    struct array {

§ 23.3.1 740
// types:
typedef T& reference;
typedef const T& const_reference;
typedef implementation-defined iterator;
typedef implementation-defined const_iterator;
typedef size_t size_type;
typedef ptrdiff_t difference_type;
typedef T value_type;
typedef T* pointer;
typedef const T* const_pointer;
typedef reverse_iterator<iterator> reverse_iterator;
typedef reverse_iterator<const_iterator> const_reverse_iterator;

T elems[N]; // exposition only

// No explicit construct/copy/destroy for aggregate type

void fill(const T& u);
void swap(array<T, N>&);

// iterators:
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// capacity:
constexpr size_type size();
constexpr size_type max_size();
constexpr bool empty();

// element access:
reference operator[](size_type n);
const_reference operator[](size_type n) const;
const_reference at(size_type n) const;
reference at(size_type n);
reference front();
const_reference front() const;
reference back();
const_reference back() const;

T * data();
const T * data() const;
};
[Note: The member variable `elems` is shown for exposition only, to emphasize that `array` is a class aggregate. The name `elems` is not part of `array`'s interface. — end note]

### 23.3.1.1 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 23.2.

### 23.3.1.2 array specialized algorithms

```
template <class T, size_t N> void swap(array<T,N>& x, array<T,N>& y);
```

1 **Effects:**

\[
x . \text{swap}(y);
\]

2 **Complexity:** linear in \(N\).

### 23.3.1.3 array::size

```
template <class T, size_t N> constexpr size_type array<T,N>::size();
```

1 **Returns:** \(N\)

### 23.3.1.4 array::data

```
T *data();
const T *data() const;
```

1 **Returns:** `elems`.

### 23.3.1.5 array::fill

```
void fill(const T& u);
```

1 **Effects:** `fill_n(begin(), N, u)`

### 23.3.1.6 array::swap

```
void swap(array& y);
```

1 **Effects:** `swap_ranges(begin(), end(), y.begin())`

2 **Throws:** Nothing unless one of the element-wise swap calls throws an exception.

3 **Note:** Unlike the `swap` function for other containers, `array::swap` takes linear time, may exit via an exception, and does not cause iterators to become associated with the other container.

### 23.3.1.7 Zero sized arrays

1 `array` shall provide support for the special case \(N == 0\).

2 In the case that \(N == 0\), `begin()` == `end()` == unique value. The return value of `data()` is unspecified.

3 The effect of calling `front()` or `back()` for a zero-sized array is implementation-defined.

### 23.3.1.8 Tuple interface to class template array

§ 23.3.1.8
tuple_size<array<T, N> >::value
1    Return type: integral constant expression.
2    Value: N

tuple_element<I, array<T, N> >::type
3    Requires: I < N. The program is ill-formed if I is out of bounds.
4    Value: The type T.

template <size_t I, class T, size_t N> T& get(array<T, N>& a);
5    Requires: I < N. The program is ill-formed if I is out of bounds.
6    Returns: A reference to the Ith element of a, where indexing is zero-based.
7    Throws: nothing.

template <size_t I, class T, size_t N> const T& get(const array<T, N>& a);
8    Requires: I < N. The program is ill-formed if I is out of bounds.
9    Returns: A const reference to the Ith element of a, where indexing is zero-based.
10   Throws: nothing.

23.3.2 Class template deque [deque]

1 A deque is a sequence container that, like a vector (23.3.6), supports random access iterators. In addition, it supports constant time insert and erase operations at the beginning or the end; insert and erase in the middle take linear time. That is, a deque is especially optimized for pushing and popping elements at the beginning and end. As with vectors, storage management is handled automatically.

2 A deque satisfies all of the requirements of a container, of a reversible container (given in tables in 23.2), of a sequence container, including the optional sequence container requirements (23.2.3), and of an allocator-aware container (Table 96). Descriptions are provided here only for operations on deque that are not described in one of these tables or for operations where there is additional semantic information.

namespace std {
    template <class T, class Allocator = allocator<T> >
    class deque {
        public:
            // types:
            typedef value_type& reference;
            typedef const value_type& const_reference;
            typedef implementation-defined iterator; // See 23.2
            typedef implementation-defined const_iterator; // See 23.2
            typedef implementation-defined size_type; // See 23.2
            typedef implementation-defined difference_type; // See 23.2
            typedef T value_type;
            typedef Allocator allocator_type;
            typedef typename allocator_traits<Allocator>::pointer pointer;
            typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
            typedef std::reverse_iterator<iterator> reverse_iterator;
            typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

            // 23.3.2.1 construct/copy/destroy:

§ 23.3.2
explicit deque(const Allocator& = Allocator());
explicit deque(size_type n);
deque(size_type n, const T& value, const Allocator& = Allocator());
template <class InputIterator>
deque(InputIterator first, InputIterator last, const Allocator& = Allocator());
deque(const deque<T, Allocator>& x);
deque(deque&&);
deque(const deque&, const Allocator&);
deque(deque&&, const Allocator&);
deque(initializer_list<T>, const Allocator& = Allocator());
~deque();
deque<T, Allocator>& operator=(const deque<T, Allocator>& x);
deque<T, Allocator>& operator=(deque<T, Allocator>&& x);
deque& operator=(initializer_list<T>);
template <class InputIterator>
void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& t);
void assign(initializer_list<T>);
allocator_type get_allocator() const;

// 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;

// 23.3.2.2 capacity:
size_type size() const;
size_type max_size() const;
void resize(size_type sz);
void resize(size_type sz, const T& c);
void shrink_to_fit();
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;

// 23.3.2.3 modifiers:
template <class... Args> void emplace_front(Args&&... args);
template <class... Args> void emplace_back(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);

void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);

iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
iterator insert (const_iterator position, InputIterator first, InputIterator last);
iterator insert(const_iterator position, initializer_list<T>);

void pop_front();
void pop_back();

iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void swap(deque<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);

23.3.2.1 deque constructors, copy, and assignment

explicit deque(const Allocator& = Allocator());

Effects: Constructs an empty deque, using the specified allocator.

Complexity: Constant.
explicit deque(size_type n);

Effects: Constructs a deque with n default constructed elements.
Requires: T shall be DefaultConstructible.
Complexity: Linear in $n$.

```cpp
deqe(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>
deqe(InputIterator first, InputIterator last,
     const Allocator& = Allocator());
```

Effects: Constructs a `deque` equal to the the range `[first,last)`, using the specified allocator.

Complexity: $\text{distance}(\text{first}, \text{last})$.

```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);`

### 23.3.2.2 deque capacity

```cpp
void resize(size_type sz);
```

Effects: If $sz < \text{size}()$, equivalent to `erase(begin() + sz, end())`; If $\text{size}() < sz$, appends $sz - \text{size}()$ default constructed elements to the sequence.

Requires: $T$ shall be `DefaultConstructible`.

```cpp
void resize(size_type sz, const T& c);
```

Effects:
- if ($sz > \text{size}()$)
  - `insert(end(), sz-size(), c);`
- else if ($sz < \text{size}()$
  - `erase(begin()+sz, end());`
- else
  - `// do nothing`

Requires: $T$ shall be `CopyConstructible`.

```cpp
void shrink_to_fit();
```

Remarks: `shrink_to_fit` is a non-binding request to reduce memory use. [Note: The request is non-binding to allow latitude for implementation-specific optimizations. — end note]
23.3.2.3 deque modifiers

```
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
    iterator insert(const_iterator position,
        InputIterator first, InputIterator last);
iterator insert(const_iterator position, initializer_list<T>);
template <class... Args> void emplace_front(Args&&... args);
template <class... Args> void emplace_back(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);

1  Effects: An insertion in the middle of the deque invalidates all the iterators and references to elements of the deque. An insertion at either end of the deque invalidates all the iterators to the deque, but has no effect on the validity of references to elements of the deque.

2  Remarks: If an exception is thrown other than by the copy constructor, move constructor, assignment operator, or move assignment operator of T there are no effects. If an exception is thrown by the move constructor of a non-CopyConstructible T, the effects are unspecified.

3  Complexity: The complexity is linear in the number of elements inserted plus the lesser of the distances to the beginning and end of the deque. Inserting a single element either at the beginning or end of a deque always takes constant time and causes a single call to a constructor of T.

iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);

4  Effects: An erase operation that erases the last element of a deque invalidates only the past-the-end iterator and all iterators and references to the erased elements. An erase operation that erases the first element of a deque but not the last element invalidates only the erased elements. An erase operation that erases neither the first element nor the last element of a deque invalidates the past-the-end iterator and all iterators and references to all the elements of the deque.

5  Complexity: The number of calls to the destructor is the same as the number of elements erased, but the number of calls to the assignment operator is no more than the lesser of the number of elements before the erased elements and the number of elements after the erased elements.

6  Throws: Nothing unless an exception is thrown by the copy constructor, move constructor, assignment operator, or move assignment operator of T.
```

23.3.2.4 deque specialized algorithms

```
template <class T, class Allocator>
    void swap(deque<T,Allocator>& x, deque<T,Allocator>& y);

1  Effects:
    x.swap(y);
```
23.3.3 Class template `forward_list`  

1 A `forward_list` is a container that supports forward iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Fast random access to list elements is not supported.  

[Note: It is intended that `forward_list` have zero space or time overhead relative to a hand-written C-style singly linked list. Features that would conflict with that goal have been omitted. — end note]

2 A `forward_list` satisfies all of the requirements of a container (table 93), except that the `size()` member function is not provided. A `forward_list` also satisfies all of the requirements for an allocator-aware container (Table 96). In addition, a `forward_list` provides the `assign` member functions (Table 97) and several of the optional container requirements (Table 98). Descriptions are provided here only for operations on `forward_list` that are not described in that table or for operations where there is additional semantic information.

3 [Note: modifying any list requires access to the element preceding the first element of interest, but in a `forward_list` there is no constant-time way to access a preceding element. For this reason, ranges that are modified, such as those supplied to `erase` and `splice`, must be open at the beginning. — end note]

```cpp
namespace std {
    template <class T, class Allocator = allocator<T> >
    class forward_list {
public:
    // types:
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef implementation-defined iterator;  // See 23.2
    typedef implementation-defined const_iterator; // See 23.2
    typedef implementation-defined size_type;  // See 23.2
    typedef implementation-defined difference_type; // See 23.2
    typedef T value_type;
    typedef Allocator allocator_type;
    typedef typename allocator_traits<Allocator>::pointer pointer;
    typedef typename allocator_traits<Allocator>::const_pointer const_pointer;

    // 23.3.3.1 construct/copy/destroy:
    explicit forward_list(const Allocator& = Allocator());
    explicit forward_list(size_type n);
    forward_list(size_type n, const T& value,
                 const Allocator& = Allocator());
    template <class InputIterator>
    forward_list(InputIterator first, InputIterator last,
                 const Allocator& = Allocator());
    forward_list(const forward_list& x);
    forward_list(forward_list&& x);
    forward_list(const forward_list& x, const Allocator&);
    forward_list(forward_list&& x, const Allocator&);
    forward_list(initializer_list<T>, const Allocator& = Allocator());
    forward_list(T);
    forward_list<T,Allocator>& operator=(const forward_list<T,Allocator>& x);
    forward_list<T,Allocator>& operator=(forward_list<T,Allocator>&& x);
    forward_list& operator=(initializer_list<T>);
    template <class InputIterator>
    void assign(InputIterator first, InputIterator last);
    void assign(size_type n, const T& t);
    void assign(initializer_list<T>);
}
```
allocator_type get_allocator() const;

// 23.3.3.2 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;
const_iterator cbefore_begin() const;
const_iterator cend() const;

// capacity:
bool empty() const;
size_type max_size() const;

// 23.3.3.3 element access:
reference front();
const_reference front() const;

// 23.3.3.4 modifiers:
template <class... Args> void emplace_front(Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void pop_front();

template <class... Args> iterator emplace_after(const_iterator position, Args&&... args);
iterator insert_after(const_iterator position, const T& x);
iterator insert_after(const_iterator position, T&& x);

iterator insert_after(const_iterator position, size_type n, const T& x);
template <class InputIterator>
    iterator insert_after(const_iterator position, InputIterator first, InputIterator last);
iterator insert_after(const_iterator position, initializer_list<T> il);

void erase_after(const_iterator position);
void 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();

// 23.3.3.5 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);

// 23.3.3.6 specialized algorithms:
    template <class T, class Allocator>
        void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>& y);
}

23.3.3.1 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).

```cpp
template <class InputIterator>
void assign(InputIterator first, InputIterator last);
```

Effects: clear(); insert_after(before_begin(), first, last);

```cpp
void assign(size_type n, const T& t);
```

Effects: clear(); insert_after(before_begin(), n, t);

### 23.3.3.2 forward_list iterators

```cpp
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()`.

### 23.3.3.3 forward_list element access

```cpp
reference front();
const_reference front() const;
```

Returns: *begin()

### 23.3.3.4 forward_list modifiers

None of the overloads of `insert_after` shall affect the validity of iterators and references, and `erase_after` shall invalidate only iterators and references to the erased elements. If an exception is thrown during `insert_after` there shall be no effect. Inserting \( n \) elements into a `forward_list` is linear in \( n \), and the number of calls to the copy or move constructor of \( T \) is exactly equal to \( n \). Erasing \( n \) elements from a `forward_list` is linear in \( n \) and the number of calls to the destructor of type \( T \) is exactly equal to \( n \).

```cpp
template <class... Args> void emplace_front(Args&&... args);
```

Effects: Inserts an object of type `value_type` constructed with `value_type(std::forward<Args>(args)...)` at the beginning of the list.

```cpp
void push_front(const T& x);
void push_front(T&& x);
```

Effects: Inserts a copy of \( x \) at the beginning of the list.

```cpp
void pop_front();
```

Effects: `erase_after(before_begin())`

```cpp
iterator insert_after(const_iterator position, const T& x);
iterator insert_after(const_iterator position, T&& x);
```

Requires: `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(), end())`.

Effects: Inserts a copy of \( x \) after `position`.

Returns: An iterator pointing to the copy of \( x \).
iterator insert_after(const_iterator position, size_type n, const T& x);

Requires: position is before_begin() or is a dereferenceable iterator in the range [begin(),end()).
Effects: Inserts n copies of x after position.
Returns: position.

template <class InputIterator>
iterator insert_after(const_iterator position, InputIterator first, InputIterator last);

Requires: position is before_begin() or is a dereferenceable iterator in the range [begin(),end()).
first and last are not iterators in *this.
Effects: Inserts copies of elements in [first,last) after position.
Returns: position.

iterator insert_after(const_iterator position, initializer_list<T> il);

Effects: insert_after(p, il.begin(), il.end()).
Returns: position.

template <class... Args>
iterator emplace_after(const_iterator position, Args&&... args);

Requires: position is before_begin() or is a dereferenceable iterator in the range [begin(),end()).
Effects: Inserts an object of type value_type constructed with value_type(std::forward<Args>(args)...)
after position.

void erase_after(const_iterator position);

Requires: The iterator following position is dereferenceable.
Effects: Erases the element pointed to by the iterator following position.

void 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).

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()).

Remarks: Does not invalidate past-the-end iterators.

23.3.3.5 forward_list operations [forwardlist.ops]

void splice_after(const_iterator position, forward_list<T,Allocator>&& x);
Requires: position is before_begin() or is a dereferenceable iterator in the range [begin(), end()).

Effects: Inserts the contents of x after position, and x becomes empty. Pointers and references to the moved elements of x now refer to those same elements but as members of *this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this, not into x.

Throws: Nothing.

Complexity: \( \Theta(\text{distance}(x\.begin(), x\.end())) \)

```cpp
void splice_after(const_iterator position, forward_list<T, Allocator>&& x, const_iterator i);
```

Requires: position is before_begin() or is a dereferenceable iterator in the range [begin(), end()). The iterator following i is a dereferenceable iterator in x.

Effects: Inserts the element following i into *this, following position, and removes it from x. The result is unchanged if position == i or position == ++i. Pointers and references to *i continue to refer to the same element but as a member of *this. Iterators to *i (including i itself) continue to refer to the same element, but now behave as iterators into *this, not into x.

Throws: Nothing.

Complexity: \( \Theta(1) \)

```cpp
void splice_after(const_iterator position, forward_list<T, Allocator>&& x,
                 const_iterator first, const_iterator last);
```

Requires: position is before_begin() or is a dereferenceable iterator in the range [begin(), end()). (first, last) is a valid range in x, and all iterators in the range (first, last) are dereferenceable. position is not an iterator in the range (first, last).

Effects: Inserts elements in the range (first, last) after position and removes the elements from x. Pointers and references to the moved elements of x now refer to those same elements but as members of *this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this, not into x.

Complexity: \( \Theta(\text{distance}(\text{first}, \text{last})) \)

```cpp
void remove(const T& value);
template <class Predicate> void remove_if(Predicate pred);
```

Effects: Erases all the elements in the list referred by a list iterator i for which the following conditions hold: *i == value (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.

```cpp
void unique();
template <class BinaryPredicate> void unique(BinaryPredicate pred);
```

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 \([\text{first, last})\) is not empty, exactly \((\text{last} - \text{first}) - 1\) applications of the corresponding predicate, otherwise no applications of the predicate.

\[
\text{void merge(forward_list<T, Allocator>&& x)}; \\
\text{template <class Compare> void merge(forward_list<T, Allocator>&& x, Compare comp)}
\]

Requires: \(\text{comp}\) defines a strict weak ordering (25.4), and \(*\text{this}\) and \(x\) are both sorted according to this ordering.

Effects: Merges \(x\) into \(*\text{this}\). This operation shall be stable: for equivalent elements in the two lists, the elements from \(*\text{this}\) 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 distance(begin(), end()) + distance(x.begin(), x.end()) - 1 comparisons.

\[
\text{void sort();} \\
\text{template <class Compare> void sort(Compare comp)}; \\
\text{Requires: operator< (for the version with no arguments) or \text{comp} (for the version with a comparison argument) defines a strict weak ordering (25.4).}
\]

Effects: Sorts the list according to the operator< or the \text{comp} function object. 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 \(*\text{this}\) is unspecified.

Complexity: Approximately \(N \log N\) comparisons, where \(N\) is distance(begin(), end()).

\[
\text{void reverse();} \\
\text{Effects: Reverses the order of the elements in the list.}
\]

Throws: Nothing.

Complexity: Linear time.

23.3.3.6 forward_list specialized algorithms [forwardlist.spec]

\[
\text{template <class T, class Allocator>} \\
\text{void swap(forward_list<T, Allocator>& x, forward_list<T, Allocator>& y)}; \\
\text{Effects: \(x\).swap(y)}
\]

23.3.4 Class template list [list]

1 A list is a sequence container that supports bidirectional iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Unlike vectors (23.3.6) and deques (23.3.2), fast random access to list elements is not supported, but many algorithms only need sequential access anyway.

2 A list satisfies all of the requirements of a container, of a reversible container (given in two tables in 23.2), of a sequence container, including most of the the optional sequence container requirements (23.2.3), and of an allocator-aware container (Table 96). The exceptions are the operator[] and at member functions, which are not provided.269 Descriptions are provided here only for operations on list that are not described in one of these tables or for operations where there is additional semantic information.

269) These member functions are only provided by containers whose iterators are random access iterators.
namespace std {

template <class T, class Allocator = allocator<T> >
class list {
public:

   // types:
   typedef value_type& reference;
   typedef const value_type& const_reference;
   typedef implementation-defined iterator;  // See 23.2
   typedef implementation-defined const_iterator;  // See 23.2
   typedef implementation-defined size_type;  // See 23.2
   typedef implementation-defined difference_type;  // See 23.2
   typedef T value_type;
   typedef Allocator allocator_type;
   typedef typename allocator_traits<Allocator>::pointer pointer;
   typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
   typedef std::reverse_iterator<iterator> reverse_iterator;
   typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

   // 23.3.4.1 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(initializer_list<T>, const Allocator& = Allocator());
   list();
   list<T, Allocator>& operator=(const list<T, Allocator>& x);
   list<T, Allocator>& operator=(list<T, Allocator>&& x);
   list& operator=(initializer_list<T>);
   template <class InputIterator>
   void assign(InputIterator first, InputIterator last);
   void assign(size_type n, const T& t);
   void assign(initializer_list<T>);
   allocator_type get_allocator() const;

   // 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;

   // 23.3.4.2 capacity:

§ 23.3.4
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;

// 23.3.4.3 modifiers:
template <class... Args> void emplace_front(Args&&... args);
void pop_front();
template <class... Args> void emplace_back(Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
void pop_back();

template <class... Args> iterator emplace(const_iterator position, Args&&... args);
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
    iterator insert(const_iterator position, InputIterator first, InputIterator last);
iterator insert(const_iterator position, initializer_list<T> il);

iterator erase(const_iterator position);
iterator erase(const_iterator position, const_iterator last);
void    swap(list<T,Allocator>&);
void    clear();

// 23.3.4.4 list operations:
void splice(const_iterator position, list<T,Allocator>& x);
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 i);
void splice(const_iterator position, list<T,Allocator>& x, const_iterator first, const_iterator last);
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);
void merge(list<T,Allocator>&& x);
template <class Compare> void merge(list<T,Allocator>& x, Compare comp);
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);

23.3.4.1 list constructors, copy, and assignment

explicit list(const Allocator& = Allocator());

    Effects: Constructs an empty list, using the specified allocator.

    Complexity: Constant.

explicit list(size_type n);

    Effects: Constructs a list with n default constructed elements.

    Requires: T shall be DefaultConstructible.

    Complexity: Linear in n.

list(size_type n, const T& value,
    const Allocator& = Allocator());

    Effects: Constructs a list with n copies of value, using the specified allocator.

    Requires: T shall be CopyConstructible.

    Complexity: Linear in n.

template <class InputIterator>
list(InputIterator first, InputIterator last,
    const Allocator& = Allocator());

    Effects: Constructs a list equal to the range [first, last).

    Complexity: Linear in distance(first, last).
template <class InputIterator>
void assign(InputIterator first, InputIterator last);

Effects: Replaces the contents of the list with the range \([first, last)\).

```c
erase(begin(), end());
insert(begin(), n, t);
```

void assign(size_type n, const T& t);

Effects: Replaces the contents of the list with \(n\) copies of \(t\).

### 23.3.4.2 list capacity

```c
void resize(size_type sz);
```

Effects: If \(sz < \text{size()\)},\) equivalent to \(\text{list<T>::iterator it = begin(); advance(it, sz); erase(it, end());}\). If \(\text{size()} < sz\), appends \(sz - \text{size()\)} default constructed elements to the sequence.

Requires: \(T\) shall be \text{DefaultConstructible\).}

```c
void resize(size_type sz, const T& c);
```

Effects:

```
if (sz > size())
    insert(end(), sz-size(), c);
else if (sz < size()) {
    iterator i = begin();
    advance(i, sz);
    erase(i, end());
} else
    // do nothing
```

Requires: \(T\) shall be \text{CopyConstructible\).}

### 23.3.4.3 list modifiers

```c
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
iterator insert(const_iterator position, InputIterator first, InputIterator last);
iterator insert(const_iterator position, initializer_list<T>);
```

template <class... Args> void emplace_front(Args&&... args);
template <class... Args> void emplace_back(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);

Remarks: Does not affect the validity of iterators and references. If an exception is thrown there are no effects.
**Complexity:** Insertion of a single element into a list takes constant time and exactly one call to a constructor of \( T \). Insertion of multiple elements into a list is linear in the number of elements inserted, and the number of calls to the copy constructor or move constructor of \( T \) is exactly equal to the number of elements inserted.

```cpp
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
```

```cpp
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.

### 23.3.4.4 list operations

Since lists allow fast insertion and erasing from the middle of a list, certain operations are provided specifically for them.\(^{270}\)

`list` provides three `splice` operations that destructively move elements from one list to another. The behavior of `splice` operations is undefined if \( \text{get_allocator}() \neq x.\text{get_allocator}() \).

```cpp
void splice(const_iterator position, list<T,Allocator>& x);
void splice(const_iterator position, list<T,Allocator>&& x);
```

**Requires:** \&\( x \neq \text{this} \).

**Effects:** Inserts the contents of \( x \) before \( \text{position} \) and \( x \) becomes empty. Pointers and references to the moved elements of \( x \) now refer to those same elements but as members of \(*\text{this}\). Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into \(*\text{this}\), not into \( x \).

**Throws:** Nothing

**Complexity:** Constant time.

```cpp
void splice(const_iterator position, list<T,Allocator>& x, const_iterator i);
void splice(const_iterator position, list<T,Allocator>&& x, const_iterator i);
```

**Effects:** Inserts an element pointed to by \( i \) from list \( x \) before \( \text{position} \) and removes the element from \( x \). The result is unchanged if \( \text{position} \neq i \) or \( \text{position} \neq ++i \). Pointers and references to \(*i\) continue to refer to this same element but as a member of \(*\text{this}\). Iterators to \(*i\) (including \( i \) itself) continue to refer to the same element, but now behave as iterators into \(*\text{this}\), not into \( x \).

**Throws:** Nothing

**Requires:** \( i \) is a valid dereferenceable iterator of \( x \).

**Complexity:** Constant time.

---

\(^{270}\) As specified in 20.2.5, the requirements in this Clause apply only to lists whose allocators compare equal.
void splice(const_iterator position, list<T,Allocator>& x, const_iterator first, const_iterator last);
void splice(const_iterator position, list<T,Allocator>&& x, const_iterator first, const_iterator last);

Effects: Inserts elements in the range \([\text{first}, \text{last})\) before \text{position} and removes the elements from \text{x}.

Requires: \([\text{first}, \text{last})\) is a valid range in \text{x}. The result is undefined if \text{position} is an iterator in the range \([\text{first}, \text{last})\). Pointers and references to the moved elements of \text{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 \text{x}.

Throws: Nothing

Complexity: Constant time if \&\text{x} == this; otherwise, linear time.

void remove(const T& value);

Effects: Erases all the elements in the list referred by a list iterator \text{i} for which the following conditions hold: \(*\text{i} == \text{value}, \text{pred}(\text{i}) \neq \text{false}\).

Throws: Nothing unless an exception is thrown by \(*\text{i} == \text{value}\) or \text{pred}(\text{i}) \neq \text{false}\).

Remarks: Stable.

Complexity: Exactly \text{size()} applications of the corresponding predicate.

void unique();

template <class BinaryPredicate> void unique(BinaryPredicate binary_pred);

Effects: Eliminates all but the first element from every consecutive group of equal elements referred to by the iterator \text{i} in the range \([\text{first} + 1, \text{last})\) for which \(*\text{i} == *(\text{i} - 1)\) (for the version of \text{unique} with no arguments) or \text{pred}(\text{i}, *(\text{i} - 1)) (for the version of \text{unique} with a predicate argument) holds.

Throws: Nothing unless an exception in thrown by \(*\text{i} == *(\text{i} - 1)\) or \text{pred}(\text{i}, *(\text{i} - 1))

Complexity: If the range \([\text{first}, \text{last})\) is not empty, exactly \((\text{last} - \text{first}) - 1\) applications of the corresponding predicate, otherwise no applications of the predicate.

void merge(list<T,Allocator>& x);
void merge(list<T,Allocator>&& x);
template <class Compare> void merge(list<T,Allocator>& x, Compare comp);
template <class Compare> void merge(list<T,Allocator>&& x, Compare comp);

Requires: \text{comp} shall define a strict weak ordering \((25.4)\), and both the list and the argument list shall be sorted according to this ordering.

Effects: If \&(\text{x} == this) does nothing; otherwise, merges the two sorted ranges \([\text{begin()}, \text{end()}\) and \([\text{x.begin()}, \text{x.end()}\)\). 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 \text{i}, in the range other than the first, the condition \text{comp}(\text{i}, *(\text{i} - 1)) will be false.

Remarks: Stable. If \&(\text{x} != this) the range \([\text{x.begin()}, \text{x.end()}\) is empty after the merge.

Complexity: At most \text{size()} + \text{x.size()} - 1 applications of \text{comp} if \&(\text{x} != this); otherwise, no applications of \text{comp} are performed. If an exception is thrown other than by a comparison there are no effects.
void reverse();
26    
27    **Effects:** Reverses the order of the elements in the list.
28
29    **Throws:** Nothing.
30    
31    **Complexity:** Linear time.
32
33    void sort();
34    template <class Compare> void sort(Compare comp);
35
36    **Requires:** operator< (for the first version) or comp (for the second version) shall define a strict weak ordering (25.4).
37
38    **Effects:** Sorts the list according to the operator< or a Compare function object.
39    
40    **Remarks:** Stable.
41    
42    **Complexity:** Approximately \(N \log(N)\) comparisons, where \(N == \text{size()}\).
43
23.3.4.5 list specialized algorithms  
[link:list.special]
44    
45    template <class T, class Allocator>
46    void swap(list<T,Allocator>& x, list<T,Allocator>& y);
47
48    **Effects:**
49    
50    x.swap(y);
51
23.3.5 Container adaptors  
[link:container.adaptors]
52
53    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.
54
55    For container adaptors, no swap function throws an exception unless that exception is thrown by the swap of the adaptor’s Container or Compare object (if any).
56
23.3.5.1 Class template queue  
[link:queue]
57
58    Any sequence container supporting operations front(), back(), push_back() and pop_front() can be used to instantiate queue. In particular, list (23.3.4) and deque (23.3.2) can be used.
59
23.3.5.1.1 queue definition  
[link:queue.defn]
60    
61    namespace std {
62        
63        template <class T, class Container = deque<T> >
64        class queue {
65            
66            **public:**
67                
68                typedef typename Container::value_type value_type;
69                typedef typename Container::reference reference;
70                typedef typename Container::const_reference const_reference;
71                typedef typename Container::size_type size_type;
72                
73                **protected:**
74                    Container c;
75
76    § 23.3.5.1.1 761
public:
    explicit queue(const Container&);  
    explicit queue(Container&& = Container());
    queue(queue&& q);
    template <class Alloc> explicit queue(const Alloc&);
    template <class Alloc> queue(const Container&, const Alloc&);
    template <class Alloc> queue(Container&&, const Alloc&);
    template <class Alloc> queue(const queue&, const Alloc&);
    template <class Alloc> queue(queue&&, const Alloc&);
    queue& operator=(queue&& q);

    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    reference front() { return c.front(); }
    const_reference front() const { return c.front(); }
    reference back() { return c.back(); }
    const_reference back() const { return c.back(); }
    void push(const value_type& x) { c.push_back(x); }
    void push(value_type&& x) { c.push_back(std::move(x)); }
    template <class... Args> void emplace(Args&&... args)
        { c.emplace_back(std::forward<Args>(args)...); }
    void pop() { c.pop_front(); }
    void swap(queue& q) { c.swap(q.c); }
};

template <class T, class Container>
bool operator==(const queue<T, Container>& x, const queue<T, Container>& y);

§ 23.3.5.1.2 queue constructors

explicit queue(const Container& cont);

Effects: Initializes c with cont.

explicit queue(Container&& cont = Container());

Effects: Initializes c with std::move(cont).
queue(queue&& q);
3   \textit{Effects}: Initializes \( c \) with \texttt{std::move(q.c)}.

queue& operator=(queue&& q);
4   \textit{Effects}: Assigns \texttt{std::move(q.c)} to \( c \).
5   \textit{Returns}: *this.

23.3.5.1.3 queue constructors with allocators \([\text{queue.cons.alloc}]\)
1   If \texttt{uses_allocator<container\_type, Alloc>::value} is \texttt{false} the constructors in this subclause shall not participate in overload resolution.

template <class Alloc>
   explicit queue(const Alloc& a);
2   \textit{Effects}: Initializes \( c \) with \( a \).

template <class Alloc>
   queue(const container\_type& cont, const Alloc& a);
3   \textit{Effects}: Initializes \( c \) with \( \text{cont} \) as the first argument and \( a \) as the second argument.

template <class Alloc>
   queue(container\_type&& cont, const Alloc& a);
4   \textit{Effects}: Initializes \( c \) with \texttt{std::move(cont)} as the first argument and \( a \) as the second argument.

template <class Alloc>
   queue(const queue& q, const Alloc& a);
5   \textit{Effects}: Initializes \( c \) with \( q.c \) as the first argument and \( a \) as the second argument.

template <class Alloc>
   queue(queue&& q, const Alloc& a);
6   \textit{Effects}: Initializes \( c \) with \texttt{std::move(q.c)} as the first argument and \( a \) as the second argument.

23.3.5.1.4 queue operators \([\text{queue.ops}]\)

template <class T, class Container>
   bool operator==(const queue\_T, Container\_& x,
                  const queue\_T, Container\_& y);
1   \textit{Returns}: \( x.c == y.c \).

template <class T, class Container>
   bool operator!=(const queue\_T, Container\_& x,
                  const queue\_T, Container\_& y);
2   \textit{Returns}: \( x.c != y.c \).

template <class T, class Container>
   bool operator< (const queue\_T, Container\_& x,
                  const queue\_T, Container\_& y);
3   \textit{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.

23.3.5.1.5 queue specialized algorithms [queue.special]

template <class T, class Container>
void swap(queue<T, Container>& x, queue<T, Container>& y);
1
   Effects: x.swap(y).

23.3.5.2 Class template priority_queue [priority.queue]

Any sequence container with random access iterator and supporting operations front(), push_back() and pop_back() can be used to instantiate priority_queue. In particular, vector (23.3.6) and deque (23.3.2) 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 (25.4).

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&&);
template <class Alloc> explicit priority_queue(const Alloc&);

template <class Alloc> priority_queue(const Compare&,
    const Container&, const Alloc&);

template <class Alloc> priority_queue(const Compare&,
    Container&&, const Alloc&);

template <class Alloc> priority_queue(const priority_queue&,
    const Alloc&);

priority_queue& operator=(priority_queue&&);
5 Effects: Initializes c with std::move(q.c) and initializes comp with std::move(q.comp).

priority_queue& operator=(priority_queue&& q);
6 Effects: Assigns std::move(q.c) to c and assigns std::move(q.comp) to comp.
7 Returns: *this.

23.3.5.2.2 priority_queue constructors with allocators [priqueue.cons.alloc]

1 If uses_allocator<container_type, Alloc>::value is false the constructors in this subclause shall not participate in overload resolution.

template <class Alloc>
  explicit priority_queue(const Alloc& a);
2 Effects: Initializes c with a and value-initializes comp.

template <class Alloc>
  priority_queue(const Compare& compare, const Alloc& a);
3 Effects: Initializes c with a and initializes comp with compare.

template <class Alloc>
  priority_queue(const Compare& compare, const Container& cont, const Alloc& a);
4 Effects: Initializes c with cont as the first argument and a as the second argument, and initializes comp with compare.

template <class Alloc>
  priority_queue(const Compare& compare, Container&& cont, const Alloc& a);
5 Effects: Initializes c with std::move(cont) as the first argument and a as the second argument, and initializes comp with compare.

template <class Alloc>
  priority_queue(const priority_queue& q, const Alloc& a);
6 Effects: Initializes c with q.c as the first argument and a as the second argument, and initializes comp with q.comp.

template <class Alloc>
  priority_queue(priority_queue&& q, const Alloc& a);
7 Effects: Initializes c with std::move(q.c) as the first argument and a as the second argument, and initializes comp with std::move(q.comp).

23.3.5.2.3 priority_queue members [priqueue.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);
2 Effects:

§ 23.3.5.2.3
c.push_back(std::move(x));
push_heap(c.begin(), c.end(), comp);

template <class... Args> void emplace(Args&&... args)
3   Effects:
c.emplace_back(std::forward<Args>(args)...);
push_heap(c.begin(), c.end(), comp);

void pop();
4   Effects:
pop_heap(c.begin(), c.end(), comp);
c.pop_back();

23.3.5.2.4 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);
1   Effects: x.swap(y).

23.3.5.3 Class template stack
[ stack ]

1 Any sequence container supporting operations back(), push_back() and pop_back() can be used to instantiate stack. In particular, vector (23.3.6), list (23.3.4) and deque (23.3.2) can be used.

23.3.5.3.1 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;
   protected:
      Container c;

   public:
      explicit stack(const Container&);
      explicit stack(Container&& = Container());
      stack(stack&& s);
      template <class Alloc> explicit stack(const Alloc&);
      template <class Alloc> stack(const Container&, const Alloc&);
      template <class Alloc> stack(Container&, const Alloc&);
      template <class Alloc> stack(const stack&, const Alloc&);
      template <class Alloc> stack(stack&&, const Alloc&);
      stack& operator=(stack&& s);
      
      bool   empty() const  { return c.empty(); }  
      size_type size() const { return c.size(); }  
   }
reference top() { return c.back(); }
const_reference top() const { return c.back(); }
void push(const value_type& x) { c.push_back(x); }
void push(value_type&& x) { c.push_back(std::move(x)); }
template <class... Args> void emplace(Args&&... args)
    { c.emplace_back(std::forward<Args>(args)...); }
void pop() { c.pop_back(); }
void swap(stack& s) { 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>
    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, class Alloc>
    struct uses_allocator<stack<T, Container>, Alloc>
        : uses_allocator<Container, Alloc>::type { };

23.3.5.3.2 stack constructors

    stack(stack&& s);

    Effects: Initializes c with std::move(s.c).

    stack& operator=(stack&& s);

1    Effects: Assigns std::move(s.c) to c.

2    Returns: *this.

23.3.5.3.3 stack constructors with allocators

1 If uses_allocator<container_type, Alloc>::value is false the constructors in this subclause shall not participate in overload resolution.

    template <class Alloc>
    explicit stack(const Alloc& a);

2    Effects: Initializes c with a.

    template <class Alloc>
    stack(const container_type& cont, const Alloc& a);

3    Effects: Initializes c with cont as the first argument and a as the second argument.
template <class Alloc>
stack(container_type&& cont, const Alloc& a);

Effects: Initializes \( c \) with std::move\( (\text{cont}) \) as the first argument and \( a \) as the second argument.

template <class Alloc>
stack(const stack& s, const Alloc& a);

Effects: Initializes \( c \) with \( s.c \) as the first argument and \( a \) as the second argument.

template <class Alloc>
stack(stack&& s, const Alloc& a);

Effects: Initializes \( c \) with std::move\( (s.c) \) as the first argument and \( a \) as the second argument.

### 23.3.5.3.4 stack operators

[stack.ops]

template <class T, class Container>
bool operator==(const stack<T, Container>& x,
                const stack<T, Container>& y);

Returns: \( x.c == y.c \).

template <class T, class Container>
bool operator!=(const stack<T, Container>& x,
                const stack<T, Container>& y);

Returns: \( x.c != y.c \).

template <class T, class Container>
bool operator<(const stack<T, Container>& x,
               const stack<T, Container>& y);

Returns: \( x.c < y.c \).

template <class T, class Container>
bool operator<=(const stack<T, Container>& x,
                const stack<T, Container>& y);

Returns: \( x.c <= y.c \).

template <class T, class Container>
bool operator>(const stack<T, Container>& x,
               const stack<T, Container>& y);

Returns: \( x.c > y.c \).

template <class T, class Container>
bool operator>=(const stack<T, Container>& x,
                const stack<T, Container>& y);

Returns: \( x.c >= y.c \).

### 23.3.5.3.5 stack specialized algorithms

[stack.special]

template <class T, class Container>
void swap(stack<T, Container>& x, stack<T, Container>& y);

Effects: \( x \).swap\( (y) \).

§ 23.3.5.3.5 769
23.3.6 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 <= n < v.size()`.

A `vector` satisfies all of the requirements of a container and of a reversible container (given in two tables in 23.2), of a sequence container, including most of the optional sequence container requirements (23.2.3), and of an allocator-aware container (Table 96). 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 value_type& reference;
      typedef const value_type& const_reference;
      typedef implementation-defined iterator; // See 23.2
      typedef implementation-defined const_iterator; // See 23.2
      typedef implementation-defined size_type; // See 23.2
      typedef implementation-defined difference_type; // See 23.2
      typedef T value_type;
      typedef Allocator allocator_type;
      typedef typename allocator_traits<Allocator>::pointer pointer;
      typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
      typedef std::reverse_iterator<iterator> reverse_iterator;
      typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

      // 23.3.6.1 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&, const Allocator&);
      vector(vector&, const Allocator&);
      vector(initializer_list<T>, const Allocator& = Allocator());
      ~vector();
      vector<T,Allocator>& operator=(const vector<T,Allocator>& x);
      vector<T,Allocator>& operator=(vector<T,Allocator>&& x);
      vector& operator=(initializer_list<T>);
      template <class InputIterator>
      void assign(InputIterator first, InputIterator last);
      void assign(size_type n, const T& u);
      void assign(initializer_list<T>);
      allocator_type get_allocator() const;

      // 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;

// 23.3.6.2 capacity:
size_type size() const;
size_type max_size() const;
void resize(size_type sz);
void resize(size_type sz, const T& c);
size_type capacity() const;
bool empty() const;
void reserve(size_type n);
void shrink_to_fit();

// element access:
reference operator[](size_type n);
const_reference operator[](size_type n) const;
const_reference at(size_type n) const;
reference front();
const_reference front() const;
reference back();
const_reference back() const;

// 23.3.6.3 data access
T* data();
const T* data() const;

// 23.3.6.4 modifiers:
template <class... Args> void emplace_back(Args&&... args);
void push_back(const T& x);
void push_back(T&& x);
void pop_back();
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
iterator insert(const_iterator position, InputIterator first, InputIterator last);
iterator insert(const_iterator position, initializer_list<T> il);
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void swap(vector<T,Allocator>&);
```cpp
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>
bool operator!=(const vector<T,Allocator>& x, const vector<T,Allocator>& y);
template <class T, class Allocator>
bool operator>(const vector<T,Allocator>& x, const vector<T,Allocator>& y);
template <class T, class Allocator>
bool operator>=(const vector<T,Allocator>& x, const vector<T,Allocator>& y);
template <class T, class Allocator>
bool operator<=(const vector<T,Allocator>& x, const vector<T,Allocator>& y);

// specialized algorithms:
template <class T, class Allocator>
void swap(vector<T,Allocator>& x, vector<T,Allocator>& y);

23.3.6.1 vector constructors, copy, and assignment

explicit vector(const Allocator& = Allocator());

Effects: Constructs an empty vector, using the specified allocator.
Complexity: Constant.

explicit vector(size_type n);

Effects: Constructs a vector with n default constructed elements.
Requires: T shall be DefaultConstructible.
Complexity: Linear in n.

vector(size_type n, const T& value,
    const Allocator& = Allocator());

Effects: Constructs a vector with n copies of value, using the specified allocator.
Requires: T shall be CopyConstructible.
Complexity: Linear in n.

template <class InputIterator>
vector(InputIterator first, InputIterator last,
    const Allocator& = Allocator());

Effects: Constructs a vector equal to the range [first,last), using the specified allocator.
Complexity: Makes only N calls to the copy constructor of T (where N is the distance between first
and last) and no reallocations if iterators first and last are of forward, bidirectional, or random access
categories. It makes order N calls to the copy constructor of T and order log(N) reallocations if they
are just input iterators.

template <class InputIterator>
void assign(InputIterator first, InputIterator last);

§ 23.3.6.1 772
```
Effects:
  erase(begin(), end());
  insert(begin(), first, last);

void assign(size_type n, const T& t);

Effects:
  erase(begin(), end());
  insert(begin(), n, t);

23.3.6.2 vector capacity

size_type capacity() const;

Returns: The total number of elements that the vector can hold without requiring reallocation.

void reserve(size_type n);

Effects: A directive that informs a vector of a planned change in size, so that it can manage the storage allocation accordingly. After reserve(), capacity() is greater or equal to the argument of reserve if reallocation happens; and equal to the previous value of capacity() otherwise. Reallocation happens at this point if and only if the current capacity is less than the argument of reserve(). If an exception is thrown other than by the move constructor of a non-CopyConstructible type, there are no effects.

Complexity: It does not change the size of the sequence and takes at most linear time in the size of the sequence.

Throws: length_error if n > max_size().

Remarks: Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. 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().

void shrink_to_fit();

Remarks: shrink_to_fit is a non-binding request to reduce capacity() to size(). [Note: The request is non-binding to allow latitude for implementation-specific optimizations. — end note]

void swap(vector<T,Allocator>& x);

Effects: Exchanges the contents and capacity() of *this with that of x.

Complexity: Constant time.

void resize(size_type sz);

Effects: If sz < size(), equivalent to erase(begin() + sz, end()); If size() < sz, appends sz - size() default constructed elements to the sequence.

Requires: T shall be CopyConstructible.

void resize(size_type sz, const T& c);

Effects:

271) reserve() uses Allocator::allocate() which may throw an appropriate exception.
if (sz > size())
    insert(end(), sz-size(), c);
else if (sz < size())
    erase(begin()+sz, end());
else
    // do nothing

Requirements: If an exception is thrown other than by the move constructor of a non-CopyConstructible T there are no effects.

23.3.6.3 vector data

T* data();
const T* 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.

23.3.6.4 vector modifiers

iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
    iterator insert(const_iterator position, InputIterator first, InputIterator last);
iterator insert(const_iterator position, initializer_list<T>);
template <class... Args> void emplace_back(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
void push_back(const T& x);
void push_back(T&& x);

Remarks: Causes reallocation if the new size is greater than the old capacity. If no reallocation happens, all the iterators and references before the insertion point remain valid. If an exception is thrown other than by the copy constructor, move constructor, assignment operator, or move assignment operator of T or by any InputIterator operation there are no effects. If an exception is thrown by the move constructor of a non-CopyConstructible T, the effects are unspecified.

Complexity: The complexity is linear in the number of elements inserted plus the distance to the end of the vector.

iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);

Effects: Invalidates iterators and references at or after the point of the erase.

Complexity: The destructor of T is called the number of times equal to the number of the elements erased, but the move assignment operator of T is called the number of times equal to the number of elements in the vector after the erased elements.

Throws: Nothing unless an exception is thrown by the copy constructor, move constructor, assignment operator, or move assignment operator of T.
23.3.6.5 vector specialized algorithms

template <class T, class Allocator>
void swap(vector<T,Allocator>& x, vector<T,Allocator>& y);

Effects:
  x.swap(y);

23.3.7 Class vector<bool>

To optimize space allocation, a specialization of vector for bool elements is provided:

namespace std {
  template <class Allocator> class vector<bool, Allocator> {
    public:
      // types:
      typedef bool const_reference;
      typedef implementation-defined iterator;  // See 23.2
      typedef implementation-defined const_iterator; // See 23.2
      typedef implementation-defined size_type; // See 23.2
      typedef implementation-defined difference_type; // See 23.2
      typedef bool value_type;
      typedef Allocator allocator_type;
      typedef implementation-defined pointer;
      typedef implementation-defined const_pointer;
      typedef std::reverse_iterator<iterator> reverse_iterator;
      typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

      // bit reference:
      class reference {
        friend class vector;
        reference();
        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(initializer_list<bool>, const Allocator& = Allocator());
      ~vector();
      vector<bool,Allocator>& operator=(const vector<bool,Allocator>& x);
  };

§ 23.3.7 775
vector<bool, Allocator>& operator=(vector<bool, Allocator>&& x);
vector operator=(initializer_list<bool>);

template <class InputIterator>
  void assign(InputIterator first, InputIterator last);
void assign(size_type n, const bool& t);
void assign(initializer_list<bool>);
allocator_type get_allocator() const;

// 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);
void shrink_to_fit();

// element access:
reference operator[](size_type n);
const_reference operator[](size_type n) const;
const_reference at(size_type n) const;
reference at(size_type n);
reference front();
const_reference front() const;
reference back();
const_reference back() const;

// modifiers:
void push_back(const bool& x);
void pop_back();
iterator insert(const_iterator position, const bool& x);
iterator insert(const_iterator position, size_type n, const bool& x);
template <class InputIterator>
  iterator insert(const_iterator position,
                  InputIterator first, InputIterator last);
iterator insert(const_iterator position, initializer_list<bool> il);

iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void swap(vector<bool, Allocator>&);
static void swap(reference x, reference y);
void flip();  // flips all bits
void clear();
};

Unless described below, all operations have the same requirements and semantics as the primary vector template, except that operations dealing with the bool value type map to bit values in the container storage and allocator_traits::construct (20.9.4.2) is not used to construct these values.

There is no requirement that the data be stored as a contiguous allocation of bool values. A space-optimized representation of bits is recommended instead.

reference is a class that simulates the behavior of references of a single bit in vector<bool>. The conversion operator returns true when the bit is set, and false otherwise. The assignment operator sets the bit when the argument is (convertible to) true and clears it otherwise. flip reverses the state of the bit.

Effects: Replaces each element in the container with its complement.

static void swap(reference x, reference y);

Effects: exchanges the contents of x and y as if by

bool b = x;
x = y;
y = b;

template <class Allocator> struct hash<vector<bool, Allocator> >;

Requires: the template specialization shall meet the requirements of class template hash (20.8.15).

23.4 Associative containers

Headers <map> and <set>:

Header <map> synopsis

namespace std {
  #include <initializer_list>

  template <class Key, class T, class Compare = less<Key>,
  class Allocator = allocator<pair<const Key, T> > >
  class map;
  template <class Key, class T, class Compare, class Allocator>
  bool operator==(const map<Key,T,Compare,Allocator>& x,
  const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
  bool operator<(const map<Key,T,Compare,Allocator>& x,
  const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
  bool operator!=(const map<Key,T,Compare,Allocator>& x,
  const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
  bool operator>(const map<Key,T,Compare,Allocator>& x,
  const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
  bool operator>=(const map<Key,T,Compare,Allocator>& x,
  const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
  bool operator<(const map<Key,T,Compare,Allocator>& x,
  const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
  bool operator>=(const map<Key,T,Compare,Allocator>& x,
  const map<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator>=(const map<Key,T,Compare,Allocator>& x,
             const map<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator<=(const map<Key,T,Compare,Allocator>& x,
             const map<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  void swap(map<Key,T,Compare,Allocator>& x,
            map<Key,T,Compare,Allocator>& y);

template <class Key, class T, class Compare = less<Key>,
          class Allocator = allocator<pair<const Key, T> > >
  class multimap;
template <class Key, class T, class Compare, class Allocator>
  bool operator>=(const multimap<Key,T,Compare,Allocator>& x,
                   const multimap<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator< (const multimap<Key,T,Compare,Allocator>& x,
                   const multimap<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator!=(const multimap<Key,T,Compare,Allocator>& x,
                   const multimap<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator> (const multimap<Key,T,Compare,Allocator>& x,
                   const multimap<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator<= (const multimap<Key,T,Compare,Allocator>& x,
                  const multimap<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  void swap(multimap<Key,T,Compare,Allocator>& x,
            multimap<Key,T,Compare,Allocator>& y);

Header <set> synopsis

namespace std {
  #include <initializer_list>

  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>
    void swap(set<Key,Compare,Allocator>& x,
              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 Compare = less<Key>,
           class Allocator = allocator<Key> >
    class multiset;

template <class Key, class Compare, class Allocator>
  bool operator>=(const multiset<Key,Compare,Allocator>& x,
                  const multiset<Key,Compare,Allocator>& y);

template <class Key, class Compare, class Allocator>
  bool operator<=(const multiset<Key,Compare,Allocator>& x,
                  const multiset<Key,Compare,Allocator>& y);

template <class Key, class Compare, class Allocator>
  bool operator!=(const multiset<Key,Compare,Allocator>& x,
                  const multiset<Key,Compare,Allocator>& y);

template <class Key, class Compare, class Allocator>
  bool operator< (const multiset<Key,Compare,Allocator>& x,
                 const multiset<Key,Compare,Allocator>& y);

template <class Key, class Compare, class Allocator>
  bool operator> (const multiset<Key,Compare,Allocator>& x,
                  const multiset<Key,Compare,Allocator>& y);

template <class Key, class Compare, class Allocator>
  void swap(multiset<Key,Compare,Allocator>& x,
            multiset<Key,Compare,Allocator>& y);

23.4.1 Class template map

A map is an associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of values of another type T based on the keys. The map class supports bidirectional iterators.

A map satisfies all of the requirements of a container, of a reversible container (23.2), of an associative container (23.2.4), and of an allocator-aware container (Table 96). A map also provides most operations described in (23.2.4) for unique keys. This means that a map supports the a_unique operations in (23.2.4) but not the a_eq operations. For a map<Key,T> the key_type is Key and the value_type is pair<const Key, T>. Descriptions are provided here only for operations on map that are not described in one of those tables or for operations where there is additional semantic information.

namespace std {
    template <class Key, class T, class Compare = less<Key>,
              class Allocator = allocator<pair<const Key, T> > >
    class map {
        public:
            // types:
typedef Key key_type;
typedef T mapped_type;
typedef pair<const Key, T> value_type;
typedef Compare key_compare;
typedef Allocator allocator_type;
typedef value_type& reference;
typedef const value_type& const_reference;
typedef implementation-defined iterator; // See 23.2
typedef implementation-defined const_iterator; // See 23.2
typedef implementation-defined size_type; // See 23.2
typedef implementation-defined difference_type; // See 23.2
typedef typename allocator_traits<Allocator>::pointer pointer;
typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

class value_compare
  : 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);
  }
};

// 23.4.1.1 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(const Allocator&);
map(const map& const Allocator&);
map(map&, const Allocator&);
map(initializer_list<value_type> ,
     const Compare& = Compare(),
     const Allocator& = Allocator());
~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);
map& operator=(initializer_list<value_type>);
allocator_type get_allocator() const;

// iterators:
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// capacity:
bool empty() const;
size_type size() const;
size_type max_size() const;

// 23.4.1.2 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_hint(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);
template <class P>
    iterator insert(const_iterator position, P&&);
template <class InputIterator>
    void insert(InputIterator first, InputIterator last);
    void insert(initializer_list<value_type>);
iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);
void swap(map<Key,T,Compare,Allocator>&);
void clear();

// observers:
key_compare key_comp() const;
value_compare value_comp() const;

// 23.4.1.4 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);

template <class Key, class T, class Compare, class Allocator>
bool operator<(const map<Key, T, Compare, Allocator>& x,
    const map<Key, T, Compare, Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator!=(const map<Key, T, Compare, Allocator>& x,
    const map<Key, T, Compare, Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator>=(const map<Key, T, Compare, Allocator>& x,
    const map<Key, T, Compare, Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator<=(const map<Key, T, Compare, Allocator>& x,
    const map<Key, T, Compare, Allocator>& y);

// specialized algorithms:

template <class Key, class T, class Compare, class Allocator>
void swap(map<Key, T, Compare, Allocator>& x,
    map<Key, T, Compare, Allocator>& y);

23.4.1.1 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).

   Complexity: Linear in N if the range [first, last) is already sorted using comp and otherwise N log N,
   where N is last - first.

23.4.1.2 map element access

T& operator[](const key_type& x);
Effects: If there is no key equivalent to x in the map, inserts value_type(x, T()) into the map.

Requires: key_type shall be CopyConstructible and mapped_type shall be DefaultConstructible.

Returns: A reference to the mapped_type corresponding to x in *this.

Complexity: logarithmic.

T& operator[](key_type&& x);

Effects: If there is no key equivalent to x in the map, inserts value_type(std::move(x), T()) into the map.

Requires: mapped_type shall be DefaultConstructible.

Returns: A reference to the mapped_type corresponding to x in *this.

Complexity: logarithmic.

T& at(const key_type& x);
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 out_of_range if no such element is present.

Complexity: logarithmic.

23.4.1.3 map modifiers

template <class P> pair<iterator, bool> insert(P&& x);
template <class P> pair<iterator, bool> 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 mapped_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<const key_type, mapped_type>. Otherwise CopyConstructible is required for both key_type and mapped_type.

23.4.1.4 map operations

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>

§ 23.4.1.4

783
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 (23.2.4).

### 23.4.1.5 map specialized algorithms

```
template <class Key, class T, class Compare, class Allocator>
void swap(map<Key,T,Compare,Allocator>& x,
          map<Key,T,Compare,Allocator>& y);
```

Effects:

```
x.swap(y);
```

### 23.4.2 Class template multimap

1 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.

2 A multimap satisfies all of the requirements of a container and of a reversible container (23.2), of an associative container (23.2.4), and of an allocator-aware container (Table 96). A multimap also provides most operations described in (23.2.4) for equal keys. This means that a multimap supports the a_eq operations in (23.2.4) 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 value_type& reference;
typedef const value_type& const_reference;
typedef implementation-defined iterator; // See 23.2
typedef implementation-defined const_iterator; // See 23.2
typedef implementation-defined size_type; // See 23.2
typedef implementation-defined difference_type; // See 23.2
typedef typename allocator_traits<Allocator>::pointer pointer;
typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    class value_compare
        : public binary_function<value_type,value_type,bool> {
            friend class multimap;
            protected:
```

§ 23.4.2
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(Key,T,Compare,Allocator>&& x);
multimap(const Allocator&);
multimap(const multimap&, const Allocator&);
multimap(multimap&, const Allocator&);
multimap(initializer_list<value_type>,
         const Compare& = Compare(),
         const Allocator& = Allocator());
~multimap();
multimap<Key,T,Compare,Allocator>&
  operator=(const multimap<Key,T,Compare,Allocator>& x);
multimap<Key,T,Compare,Allocator>&
  operator=(multimap<Key,T,Compare,Allocator>&& x);
multimap& operator=(initializer_list<value_type>);
allocate_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_hint(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);
void insert(initializer_list<value_type>);

iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);
void swap(multimap<Key,T,Compare,Allocator>&);
void clear();

// observers:
key_compare key_comp() const;
value_compare value_comp() const;

// map operations:
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
size_type count(const key_type& x) const;

iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;

pair<iterator,iterator>
equal_range(const key_type& x);
pair<const_iterator,const_iterator>
equal_range(const key_type& x) const;

};

template <class Key, class T, class Compare, class Allocator>
bool operator==(const multimap<Key,T,Compare,Allocator>& x,
                const multimap<Key,T,Compare,Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator< (const multimap<Key,T,Compare,Allocator>& x,
                const multimap<Key,T,Compare,Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator!=(const multimap<Key,T,Compare,Allocator>& x,
                const multimap<Key,T,Compare,Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator> (const multimap<Key,T,Compare,Allocator>& x,
                const multimap<Key,T,Compare,Allocator>& y);

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:

§ 23.4.2 786
void swap(multimap<Key,T,Compare,Allocator>& x,
    multimap<Key,T,Compare,Allocator>& y);
}

23.4.2.1 multimap constructors

explicit multimap(const Compare& comp = Compare(),
    const Allocator& = Allocator());

Effects: Constructs an empty multimap using the specified comparison object and allocator.

Complexity: Constant.

template <class InputIterator>
multimap(InputIterator first, InputIterator last,
    const Compare& comp = Compare(),
    const Allocator& = Allocator());

Requires: If the iterator’s dereference operator returns an lvalue or a const rvalue pair<key_type, mapped_type>, then both key_type and mapped_type shall be CopyConstructible.

Effects: Constructs an empty multimap using the specified comparison object and allocator, and inserts elements from the range [first, last).

Complexity: Linear in N if the range [first, last) is already sorted using comp and otherwise N log N, where N is last - first.

23.4.2.2 multimap modifiers

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 mapped_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<const key_type, mapped_type>. Otherwise CopyConstructible is required for both key_type and mapped_type.

23.4.2.3 multimap operations

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;

§ 23.4.2.3
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` (23.2.4).

### 23.4.2.4 `multimap` specialized algorithms

```
template <class Key, class T, class Compare, class Allocator>
void swap(multimap<Key,T,Compare,Allocator>& x,  
multimap<Key,T,Compare,Allocator>& y);
```

**Effects:**

```
x.swap(y);
```

### 23.4.3 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, of a reversible container (23.2), of an associative container (23.2.4), and of an allocator-aware container (Table 96). A `set` also provides most operations described in (23.2.4) for unique keys. This means that a `set` supports the `a_unique` operations in (23.2.4) but not the `a_equal` operations. For a `set<Key>` both the `key_type` and `value_type` are `Key`. Descriptions are provided here only for operations on `set` that are not described in one of these tables and for operations where there is additional semantic information.

```cpp
namespace std {
    template <class Key, class Compare = less<Key>,  
              class Allocator = allocator<Key> >
    class set {
    public:
        // types:
        typedef Key key_type;
        typedef Key value_type;
        typedef Compare key_compare;
        typedef Compare value_compare;
        typedef Allocator allocator_type;

        typedef value_type& reference;
        typedef const value_type& const_reference;
        typedef implementation-defined iterator;  // See 23.2
        typedef implementation-defined const_iterator;  // See 23.2
        typedef implementation-defined size_type;  // See 23.2
        typedef implementation-defined difference_type;  // See 23.2

        typedef typename allocator_traits<Allocator>::pointer pointer;
        typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
        typedef std::reverse_iterator<iterator> reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

        // 23.4.3.1 construct/copy/destroy:
        explicit set(const Compare& comp = Compare(),  
                     const Allocator& = Allocator());

        template <class InputIterator>
        set(InputIterator first, InputIterator last,  
             const Compare& comp = Compare(), const Allocator& = Allocator());

        set(const set<Key,Compare,Allocator>& x);

```
set(set<Key,Compare,Allocator>&& x);
set(const Allocator&);
set(const set&, const Allocator&);
set(set&&, const Allocator&);
set(initializer_list<value_type>,
    const Compare& = Compare(),
    const Allocator& = Allocator());
~set();
set<Key,Compare,Allocator>& operator=(
    const set<Key,Compare,Allocator>& x);
set<Key,Compare,Allocator>& operator=(set<Key,Compare,Allocator>&& x);
set& operator=(initializer_list<value_type>);
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_hint(const_iterator position, Args&&... args);
pair<iterator, bool> insert(const value_type& x);
pair<iterator, bool> insert(value_type&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template <class InputIterator>
    void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);
iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);
void swap(set<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;

// specialized algorithms:
template <class Key, class Compare, class Allocator>
void swap(set<Key,Compare,Allocator>& x,
          set<Key,Compare,Allocator>& y);

23.4.3.1 set constructors, copy, and assignment

explicit set(const Compare& comp = Compare(),
             const Allocator& = Allocator());

Effects: Constructs an empty set using the specified comparison objects and allocator.

Complexity: Constant.

template <class InputIterator>
set(InputIterator first, InputIterator last,
    const Compare& comp = Compare(), const Allocator& = Allocator());
Effects: Constructs an empty set using the specified comparison object and allocator, and inserts elements from the range \([\text{first, last})\).

Requires: If the iterator’s dereference operator returns an lvalue or a non-const rvalue, then Key shall be CopyConstructible.

Complexity: Linear in \(N\) if the range \([\text{first, last})\) is already sorted using \(\text{comp}\) and otherwise \(N \log N\), where \(N\) is \(\text{last} - \text{first}\).

23.4.3.2 set specialized algorithms

\[
\text{template <class Key, class Compare, class Allocator>}
\text{ void swap(set<
\text{Key,Compare,Allocator}>& x,}
\text{ set<
\text{Key,Compare,Allocator}>& y);}
\]

Effects:
\(x\) swap \((y)\);

23.4.4 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.

A multiset satisfies all of the requirements of a container, of a reversible container (23.2), of an associative container (23.2.4), and of an allocator-aware container (Table 96). multiset also provides most operations described in (23.2.4) for duplicate keys. This means that a multiset supports the \(\text{a_eq}\) operations in (23.2.4) but not the \(\text{a_uniq}\) operations. For a \(\text{multiset<Key>}\) both the \text{key_type} and \text{value_type} are Key. Descriptions are provided here only for operations on multiset that are not described in one of these tables and for operations where there is additional semantic information.

namespace std {
\text{template <class Key, class Compare = less<Key>,}
\text{class Allocator = allocator<Key> >}
\text{class multiset {}
\text{public:}
\text{// types:}
\text{typedef Key key_type;}
\text{typedef Key value_type;}
\text{typedef Compare key_compare;}
\text{typedef Compare value_compare;}
\text{typedef Allocator allocator_type;}
\text{typedef value_type& reference;}
\text{typedef const value_type& const_reference;}
\text{typedef \text{implementation-defined} iterator; // See 23.2}
\text{typedef \text{implementation-defined} const_iterator; // See 23.2}
\text{typedef \text{implementation-defined} size_type; // See 23.2}
\text{typedef \text{implementation-defined} difference_type; // See 23.2}
\text{typedef typename allocator_traits<
\text{Allocator}>::pointer pointer;}
\text{typedef typename allocator_traits<
\text{Allocator}>::const_pointer const_pointer;}
\text{typedef std::reverse_iterator<
\text{iterator}> reverse_iterator;}
\text{typedef std::reverse_iterator<
\text{const_iterator}> const_reverse_iterator;}

\text{// construct/copy/destroy:}
\}\n\}
explicit multiset(const Compare& comp = Compare(),
               const Allocator& = Allocator());

template <class InputIterator>
multiset(InputIterator first, InputIterator last,
         const Compare& comp = Compare(),
         const Allocator& = Allocator());
multiset(const multiset<Key,Compare,Allocator>&& x);
multiset(multiset<Key,Compare,Allocator>&& x);
multiset(const Allocator&);
multiset(const multiset&, const Allocator&);
multiset(multiset&&, const Allocator&);
multiset(initializer_list<value_type>,
         const Compare& = Compare(),
         const Allocator& = Allocator());
~multiset();
multiset<Key,Compare,Allocator>&
   operator=(const multiset<Key,Compare,Allocator>&& x);
multiset<Key,Compare,Allocator>&
   operator=(multiset<Key,Compare,Allocator>&& x);
multiset<Key,Compare,Allocator>&
   operator=(initializer_list<value_type>);
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_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& x);
iterator insert(value_type&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template <class InputIterator>
void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);

iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);
void swap(multiset<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>
void swap(multiset<Key,Compare,Allocator>& x,
          multiset<Key,Compare,Allocator>& y);

// specialized algorithms:
void swap(multiset<Key,Compare,Allocator>& x,
          multiset<Key,Compare,Allocator>& y);

23.4.4.1 multiset constructors

explicit multiset(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>
multiset(InputIterator first, last,
const Compare& comp = Compare(), const Allocator& = Allocator());
```

3. **Requires:** If the iterator’s dereference operator returns an lvalue or a const rvalue, then Key shall be CopyConstructible.

4. **Effects:** Constructs an empty `multiset` using the specified comparison object and allocator, and inserts elements from the range `[first, last)`.

5. **Complexity:** Linear in \( N \) if the range `[first, last)` is already sorted using `comp` and otherwise \( N \log N \), where \( N \) is `last - first`.

### 23.4.4.2 multiset specialized algorithms

```cpp
template <class Key, class Compare, class Allocator>
void swap(multiset<Key,Compare,Allocator>& x,
multiset<Key,Compare,Allocator>& y);
```

### 23.5 Unordered associative containers

Headers `<unordered_map>` and `<unordered_set>`:

**Header `<unordered_map>` synopsis**

```cpp
namespace std {
    #include <initializer_list>

    // 23.5.1, 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;

    // 23.5.2, 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);
```
void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x, unordered_multimap<Key, T, Hash, Pred, Alloc>& y);

template <class Key, class T, class Hash, class Pred, class Alloc>
bool operator==(const unordered_map<Key, T, Hash, Pred, Alloc>& a, const unordered_map<Key, T, Hash, Pred, Alloc>& b);

template <class Key, class T, class Hash, class Pred, class Alloc>
bool operator!=(const unordered_map<Key, T, Hash, Pred, Alloc>& a, const unordered_map<Key, T, Hash, Pred, Alloc>& b);

template <class Key, class T, class Hash, class Pred, class Alloc>
bool operator==(const unordered_multimap<Key, T, Hash, Pred, Alloc>& a, const unordered_multimap<Key, T, Hash, Pred, Alloc>& b);

template <class Key, class T, class Hash, class Pred, class Alloc>
bool operator!=(const unordered_multimap<Key, T, Hash, Pred, Alloc>& a, const unordered_multimap<Key, T, Hash, Pred, Alloc>& b);

} // namespace std

Header <unordered_set> synopsis

namespace std {
  #include <initializer_list>

  // 23.5.3, class template unordered_set:
  template <class Key,
    class Hash = hash<Key>,
    class Pred = std::equal_to<Key>,
    class Alloc = std::allocator<Key> >
  class unordered_set;

  // 23.5.4, class template unordered_multiset:
  template <class Key,
    class Hash = hash<Key>,
    class Pred = std::equal_to<Key>,
    class Alloc = std::allocator<Key> >
  class unordered_multiset;

  template <class Key, class Hash, class Pred, class Alloc>
  void swap(unordered_set<Key, Hash, Pred, Alloc>& x, unordered_set<Key, Hash, Pred, Alloc>& y);

  template <class Key, class Hash, class Pred, class Alloc>
  void swap(unordered_multiset<Key, Hash, Pred, Alloc>& x, unordered_multiset<Key, Hash, Pred, Alloc>& y);

  template <class Key, class Hash, class Pred, class Alloc>
  bool operator==(const unordered_set<Key, Hash, Pred, Alloc>& a, const unordered_set<Key, Hash, Pred, Alloc>& b);

  template <class Key, class Hash, class Pred, class Alloc>
  bool operator!=(const unordered_set<Key, Hash, Pred, Alloc>& a, const unordered_set<Key, Hash, Pred, Alloc>& b);

  template <class Key, class Hash, class Pred, class Alloc>
  bool operator==(const unordered_multiset<Key, Hash, Pred, Alloc>& a, const unordered_multiset<Key, Hash, Pred, Alloc>& b);

  template <class Key, class Hash, class Pred, class Alloc>
  bool operator!=(const unordered_multiset<Key, Hash, Pred, Alloc>& a, const unordered_multiset<Key, Hash, Pred, Alloc>& b);

  } // namespace std

§ 23.5

795
23.5.1 Class template unordered_map

1 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.

2 An unordered_map satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 96). 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>.

3 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 = see below,
                           const hasher& hf = hasher(),
                           const key_equal& eql = key_equal(),
                           const allocator_type& a = allocator_type());
    template <class InputIterator>
    unordered_map(InputIterator f, InputIterator l,
                   size_type n = see below,
                   const hasher& hf = hasher(),
                   const key_equal& eql = key_equal(),
                   const allocator_type& a = allocator_type());

§ 23.5.1
unordered_map(const unordered_map&);
unordered_map(unordered_map&&);
unordered_map(const Allocator&);
unordered_map(const unordered_map&, const Allocator&);
unordered_map(unordered_map&, const Allocator&);
unordered_map(initializer_list<value_type>,
    size_type = see below,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
~unordered_map();
unordered_map& operator=(const unordered_map&);
unordered_map& operator=(unordered_map&&);
unordered_map& operator=(initializer_list<value_type>);
allocation_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_hint(const_iterator position, Args&&... args);
pair<iterator, bool> insert(const value_type& obj);
template <class P> iterator insert(const_iterator hint, P&& 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);
mapped_type& operator[](key_type&& k);
mapped_type& at(const key_type& k);
const mapped_type& at(const key_type& k) const;

// bucket interface
size_type bucket_count() const;
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_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;

// hash policy
float load_factor() const;
float max_load_factor() const;
void max_load_factor(float z);
void rehash(size_type n);
void reserve(size_type n);
};

template <class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
          unordered_map<Key, T, Hash, Pred, Alloc>& y);

23.5.1.1 unordered_map constructors

explicit unordered_map(size_type n = see below,
                        const hasher& hf = hasher(),
                        const key_equal& eql = key_equal(),
                        const allocator_type& a = allocator_type());

Effects: Constructs an empty unordered_map using the specified hash function, key equality function,
and allocator, and using at least n buckets. If n is not provided, the number of buckets is impledefault
number of buckets in unordered_map. max_load_factor() returns 1.0.

Complexity: Constant.

template <class InputIterator>
unordered_map(InputIterator f, InputIterator l,
               size_type n = see below,
               const hasher& hf = hasher(),
               const key_equal& eql = key_equal(),
               const allocator_type& a = allocator_type());

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 impledefault
number of buckets in unordered_map. Then inserts elements from the range [f, l). max_load_factor() returns 1.0.
Complexity: Average case linear, worst case quadratic.

23.5.1.2 unordered_map element access

 mapped_type& operator[](const key_type& k);
 mapped_type& operator[](key_type&& k);

 Requires: mapped_type shall be DefaultConstructible. For the first operator, key_type shall be CopyConstructible. For the second operator, key_type shall be MoveConstructible.

 Effects: If the unordered_map does not already contain an element whose key is equivalent to k, the first operator inserts the value value_type(k, mapped_type()) and the second operator inserts the value value_type(std::move(k), mapped_type()).

 Returns: A reference to x.second, where x is the (unique) element whose key is equivalent to k.

 Complexity: Average case $O(1)$, worst case $O(\text{size})$.

 mapped_type& at(const key_type& k);
 const mapped_type& at(const key_type& k) const;

 Returns: A reference to x.second, where x is the (unique) element whose key is equivalent to k.

 Throws: An exception object of type out_of_range if no such element is present.

23.5.1.3 unordered_map modifiers

 template <class P>
 pair<iterator, bool> insert(P& obj);

 Requires: value_type is constructible from std::forward<P>(obj).

 Effects: Inserts obj converted to value_type if and only if there is no element in the container with key equivalent to the key of value_type(obj).

 Returns: The bool component of the returned pair object indicates whether the insertion took place and the iterator component points to the element with key equivalent to the key of value_type(obj).

 Complexity: Average case $O(1)$, worst case $O(\text{size})$.

 Remarks: This signature shall not participate in overload resolution unless P is implicitly convertible to value_type.

 template <class P>
 iterator insert(const_iterator hint, P& obj);

 Requires: value_type is constructible from std::forward<P>(obj).

 Effects: Inserts obj converted to value_type if and only if there is no element in the container with key equivalent to the key of value_type(obj). The iterator hint is a hint pointing to where the search should start.

 Returns: An iterator that points to the element with key equivalent to the key of value_type(obj).

 Complexity: Average case $O(1)$, worst case $O(\text{size})$.

 Remarks: This signature shall not participate in overload resolution unless P is implicitly convertible to value_type.
23.5.1.4 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)`.

23.5.2 Class template unordered_multimap

1. 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.

2. An unordered_multimap satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 96). It provides the operations described in the preceding requirements table for equivalent keys; that is, an unordered_multimap supports the a_eq operations in that table, not the a_uniq operations. For an unordered_multimap<Key, T> the key type is Key, the mapped type is T, and the value type is std::pair<const Key, T>.

3. This section only describes operations on unordered_multimap that are not described in one of the requirement tables, or for which there is additional semantic information.

```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 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_multimap(size_type n = seebelow, 
                                     const hasher& hf = hasher(),
                                     const key_equal& eql = key_equal(),
                                     const allocator_type& a = allocator_type());
```
template <class InputIterator>
unordered_multimap(InputIterator f, InputIterator l,
    size_type n = see below,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
unordered_multimap(const unordered_multimap&);
unordered_multimap(unordered_multimap&&);
unordered_multimap(const Allocator&);
unordered_multimap(const unordered_multimap&,
    const Allocator&);
unordered_multimap(unordered_multimap&&,
    const Allocator&);
unordered_multimap(initializer_list<value_type>,
    size_type = see below,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
unordered_multimap();
unordered_multimap& operator=(const unordered_multimap&);
unordered_multimap& operator=(unordered_multimap&&);
unordered_multimap& operator=(initializer_list<value_type>);
~unordered_multimap();
unordered_multimap& operator=(const unordered_multimap&);
unordered_multimap& operator=(unordered_multimap&&);
unordered_multimap& operator=(initializer_list<value_type>);
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_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& obj);
iterator insert(const_iterator hint, const value_type& obj);
iterator insert(initializer_list<value_type>);
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;
23.5.2.1 unordered_multimap constructors  

**explicit unordered_multimap(size_type n = see below,**
const hasher& hf = hasher(),
const key_equal& eql = key_equal(),
const allocator_type& a = allocator_type());

**Effects:** Constructs an empty unordered_multimap using the specified hash function, key equality function, and allocator, and using at least \( n \) buckets. If \( n \) is not provided, the number of buckets is implementation-defined number of buckets in unordered_multimap. max_load_factor() returns 1.0.

**Complexity:** Constant.

**template <class InputIterator>**

unordered_multimap(InputIterator f, InputIterator l,
size_type n = see below,
const hasher& hf = hasher(),
const key_equal& eql = key_equal(),
const allocator_type& a = allocator_type());

**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
impldefdefault number of buckets in unordered_multimap. Then inserts elements from the range \([f, l)\). max_load_factor() returns 1.0.

**Complexity:** Average case linear, worst case quadratic.

### 23.5.2.2 unordered_multimap modifiers

_template <class P>
iterator insert(P&& obj);

1. **Requires:** value_type is constructible from std::forward<P>(obj).
2. **Effects:** Inserts obj converted to value_type.
3. **Returns:** An iterator that points to the element with key equivalent to the key of value_type(obj).
4. **Complexity:** Average case \(O(1)\), worst case \(O(size())\).
5. **Remarks:** This signature shall not participate in overload resolution unless P is implicitly convertible to value_type.

_template <class P>
iterator insert(const_iterator hint, P&& obj);

6. **Requires:** value_type is constructible from std::forward<P>(obj).
7. **Effects:** Inserts obj converted to value_type. The iterator hint is a hint pointing to where the search should start.
8. **Returns:** An iterator that points to the element with key equivalent to the key of value_type(obj).
9. **Complexity:** Average case \(O(1)\), worst case \(O(size())\).
10. **Remarks:** This signature shall not participate in overload resolution unless P is implicitly convertible to value_type.

### 23.5.2.3 unordered_multimap swap

_template <class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x, unordered_multimap<Key, T, Hash, Pred, Alloc>& y);

1. **Effects:** x.swap(y).

### 23.5.3 Class template unordered_set

_an unordered_set_ is an unordered associative container that supports unique keys (an unordered_set contains at most one of each key value) and in which the elements’ keys are the elements themselves.

2. An unordered_set satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 96). It provides the operations described in the preceding requirements table for unique keys; that is, an unordered_set supports the _a uniq_ operations in that table, not the _a eq_ operations. For an unordered_set<Key> the key type and the value type are both Key. The iterator and const_iterator types are both const iterator types. It is unspecified whether they are the same type.

3. 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 Key,
             class Hash = hash<Key>,
             class Pred = std::equal_to<Key>,
             class Alloc = std::allocator<Key> >
    class unordered_set
    {
    public:
        // types
        typedef Key key_type;
        typedef Key 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 = see below,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
        unordered_set(const unordered_set&);
        unordered_set(unordered_set&&);
        unordered_set(const Allocator&);
        unordered_set(const unordered_set&, const Allocator&);
        unordered_set(unordered_set&&, const Allocator&);
        unordered_set(initializer_list<value_type>,
                       size_type = see below,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
        ~unordered_set();
        unordered_set& operator=(const unordered_set&);
        unordered_set& operator=(unordered_set&&);
        unordered_set& operator=(initializer_list<value_type>);
        allocator_type get_allocator() const;

        // size and capacity
        bool empty() const;
    
    // 23.5.3 804
    
    // section 23.5.3
    
"}

§ 23.5.3 804
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_hint(const_iterator position, Args&&... args);
pair<iterator, bool> insert(const value_type& obj);
pair<iterator, bool> insert(value_type&& obj);
iterator insert(const_iterator hint, const value_type& obj);
iterator insert(const_iterator hint, value_type& obj);
template <class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);

iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear();

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;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend(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);
void reserve(size_type n);
);

template <class Key, class Hash, class Pred, class Alloc>
void swap(unordered_set<Key, Hash, Pred, Alloc>& x,
          unordered_set<Key, Hash, Pred, Alloc>& y);
}

23.5.3.1 unordered_set constructors [unord.set.cnstr]

explicit unordered_set(size_type n = see below,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
1
Effects: Constructs an empty unordered_set using the specified hash function, key equality function,
and allocator, and using at least n buckets. If n is not provided, the number of buckets is impldef
default number of buckets in unordered_set. max_load_factor() returns 1.0.
2
Complexity: Constant.

template <class InputIterator>
unordered_set(InputIterator f, InputIterator l,
    size_type n = see below,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
3
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 impldef
default number of buckets in unordered_set. Then inserts elements from the range [f, l). max_load_-factor() returns 1.0.
4
Complexity: Average case linear, worst case quadratic.

23.5.3.2 unordered_set swap [unord.set.swap]

template <class Key, class Hash, class Pred, class Alloc>
void swap(unordered_set<Key, Hash, Pred, Alloc>& x,
          unordered_set<Key, Hash, Pred, Alloc>& y);
1
Effects: x.swap(y).

23.5.4 Class template unordered_multiset [unord.multiset]

1 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.

2 An unordered_multiset satisfies all of the requirements of a container, of an unordered associative con-
tainer, and of an allocator-aware container (Table 96). It provides the operations described in the preceding
requirements table for equivalent keys; that is, an unordered_multiset supports the a_eq operations in
that table, not the a_uniq operations. For an unordered_multiset<Key> the key type and the value type
are both Key. The iterator and const_iterator types are both const iterator types. It is unspecified whether they are the same type.

3 This section only describes operations on unordered_multiset that are not described in one of the requirement tables, or for which there is additional semantic information.

namespace std {
    template <class Key, class Hash = hash<Key>, class Pred = std::equal_to<Key>, class Alloc = std::allocator<Key> >
    class unordered_multiset
    {
        public:
            // types
            typedef Key key_type;
            typedef Key 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 = see below,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
            unordered_multiset(const unordered_multiset&);
            unordered_multiset(unordered_multiset&&);
            unordered_multiset(const Allocator&);
            unordered_multiset(const unordered_multiset&, const Allocator&);
            unordered_multiset(unordered_multiset&&, const Allocator&);
            unordered_multiset(initializer_list<value_type>,
                size_type = see below,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
            unordered_multiset();
            unordered_multiset& operator=(const unordered_multiset&);
            unordered_multiset& operator=(unordered_multiset&&);

§ 23.5.4
unordered_multiset& operator=(initializer_list<value_type>);  
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_hint(const_iterator position, Args&&... args);  
iterator insert(const value_type& obj);  
iterator insert(value_type&& obj);  
iterator insert(const_iterator hint, const value_type& obj);  
iterator insert(const_iterator hint, value_type&& obj);  
template <class InputIterator> void insert(InputIterator first, InputIterator last);  
void insert(initializer_list<value_type>);  

iterator erase(const_iterator position);  
size_type erase(const key_type& k);  
iterator erase(const_iterator first, const_iterator last);  
void clear();  

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_local_iterator begin(size_type n) const;  
local_iterator end(size_type n);  
const_local_iterator end(size_type n) const;  
const_local_iterator cbegin(size_type n) const;  
const_local_iterator cend(size_type n) const;  

§ 23.5.4
// hash policy
float load_factor() const;
float max_load_factor() const;
void max_load_factor(float z);
void rehash(size_type n);
void reserve(size_type n);
};

template <class Key, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Key, Hash, Pred, Alloc>& x,
          unordered_multiset<Key, Hash, Pred, Alloc>& y);
}

23.5.4.1 unordered_multiset constructors

explicit unordered_multiset(size_type n = see below,
const hasher& hf = hasher(),
const key_equal& eql = key_equal(),
const allocator_type& a = allocator_type());

Effects: Constructs an empty unordered_multiset using the specified hash function, key equality function, and allocator, and using at least n buckets. If n is not provided, the number of buckets is impledeefault number of buckets in unordered_multiset. max_load_factor() returns 1.0.

Complexity: Constant.

template <class InputIterator>
unordered_multiset(InputIterator f, InputIterator l,
size_type n = see below,
const hasher& hf = hasher(),
const key_equal& eql = key_equal(),
const allocator_type& a = allocator_type());

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 impledeefault number of buckets in unordered_multiset. Then inserts elements from the range [f, l). max_load_factor() returns 1.0.

Complexity: Average case linear, worst case quadratic.

23.5.4.2 unordered_multiset swap

template <class Key, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Key, Hash, Pred, Alloc>& x,
          unordered_multiset<Key, Hash, Pred, Alloc>& y);

Effects: x.swap(y);
24 Iterators library

24.1 General

This Clause describes components that C++ programs may use to perform iterations over containers (Clause 23), streams (27.7), and stream buffers (27.6).

The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 101.

Table 101 — Iterators library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.2</td>
<td></td>
</tr>
<tr>
<td>24.4</td>
<td>&lt;iterator&gt;</td>
</tr>
<tr>
<td>24.5</td>
<td></td>
</tr>
<tr>
<td>24.6</td>
<td></td>
</tr>
</tbody>
</table>

24.2 Iterator requirements

24.2.1 In general

Iterators are a generalization of pointers that allow a C++ program to work with different data structures (containers) in a uniform manner. To be able to construct template algorithms that work correctly and efficiently on different types of data structures, the library formalizes not just the interfaces but also the semantics and complexity assumptions of iterators. 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 102.

Table 102 — Relations among iterator categories

<table>
<thead>
<tr>
<th>Random Access → Bidirectional → Forward → Input → Output</th>
</tr>
</thead>
</table>

Forward iterators satisfy all the requirements of input iterators and can be used whenever an input iterator is specified; Bidirectional iterators also satisfy all the requirements of forward iterators and can be used whenever a forward iterator is specified; Random access iterators also satisfy all the requirements of bidirectional iterators and can be used whenever a bidirectional iterator is specified.
Iterators that further satisfy the requirements of output iterators are called \textit{mutable iterators}. Nonmutable iterators are referred to as \textit{constant iterators}.

Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding sequence. These values are called \textit{past-the-end} values. Values of an iterator \texttt{i} for which the expression \texttt{*i} is defined are called \textit{dereferenceable}. The library never assumes that past-the-end values are dereferenceable. Iterators can also have singular values that are not associated with any sequence. \textit{Example: After the declaration of an uninitialized pointer \texttt{x} (as with \texttt{int* x;}), \texttt{x} must always be assumed to have a singular value of a pointer. \textit{— end example}]} Results of most expressions are undefined for singular values; the only exceptions are destroying an iterator that holds a singular value, the assignment of a non-singular value to an iterator that holds a singular value, and, for iterators that satisfy the \texttt{DefaultConstructible} requirements, using a value-initialized iterator as the source of a copy or move operation. \textit{[Note: This guarantee is not offered for default initialization, although the distinction only matters for types with trivial default constructors such as pointers or aggregates holding pointers. \textit{— end note}] In these cases the singular value is overwritten the same way as any other value. Dereferenceable values are always non-singular.

An iterator \texttt{j} is called \textit{reachable} from an iterator \texttt{i} if and only if there is a finite sequence of applications of the expression \texttt{++i} that makes \texttt{i} == \texttt{j}. If \texttt{j} is reachable from \texttt{i}, they refer to elements of the same sequence.

Most of the library’s algorithmic templates that operate on data structures have interfaces that use ranges. A \textit{range} is a pair of iterators that designate the beginning and end of the computation. A range \texttt{[i,i)} is an empty range; in general, a range \texttt{[i,j)} refers to the elements in the data structure starting with the element pointed to by \texttt{i} and up to but not including the element pointed to by \texttt{j}. Range \texttt{[i,j)} is valid if and only if \texttt{j} is reachable from \texttt{i}. The result of the application of functions in the library to invalid ranges is undefined.

All the categories of iterators require only those functions that are realizable for a given category in constant time (amortized). Therefore, requirement tables for the iterators do not have a complexity column.

Destruction of an iterator may invalidate pointers and references previously obtained from that iterator.

An \textit{invalid} iterator is an iterator that may be singular.\textsuperscript{272}

In the following sections, \texttt{a} and \texttt{b} denote values of type \texttt{X} or \texttt{const X}, \texttt{difference
type} and \texttt{reference} refer to the types \texttt{iterator_traits<X>::difference
type} and \texttt{iterator_traits<X>::reference}, respectively, \texttt{n} denotes a value of \texttt{difference
type}, \texttt{u}, \texttt{tmp}, and \texttt{m} denote identifiers, \texttt{r} denotes a value of \texttt{X&}, \texttt{t} denotes a value of value type \texttt{T}, \texttt{o} denotes a value of some type that is writable to the output iterator. \textit{[Note: For an iterator type \texttt{X} there must be an instantiation of \texttt{iterator_traits}<\texttt{X}> (24.4.1). \textit{— end note}]}

\textbf{24.2.2 \textit{Iterator}}

The \textit{Iterator} requirements form the basis of the iterator concept taxonomy; every iterator satisfies the \textit{Iterator} requirements. This set of requirements specifies operations for dereferencing and incrementing an iterator. Most algorithms will require additional operations to read (24.2.3) or write (24.2.4) values, or to provide a richer set of iterator movements (24.2.5, 24.2.6, 24.2.7)).

A type \texttt{X} satisfies the \textit{Iterator} requirements if:

\begin{itemize}
  \item \texttt{X} satisfies the \texttt{CopyConstructible}, \texttt{CopyAssignable}, and \texttt{Destructible} requirements (20.2.1) and \texttt{lvalues} of type \texttt{X} are swappable (20.2.2), and
  \item the expressions in Table 103 are valid and have the indicated semantics.
\end{itemize}

\textsuperscript{272} This definition applies to pointers, since pointers are iterators. The effect of dereferencing an iterator that has been invalidated is undefined.

\textsuperscript{811}
Table 103 — Iterator requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>*r</td>
<td>reference</td>
<td></td>
<td>pre: r is dereferenceable.</td>
</tr>
<tr>
<td>++r</td>
<td>X&amp;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

24.2.3 Input iterators

1 A class or a built-in type X satisfies the requirements of an input iterator for the value type T if X satisfies the Iterator (24.2.2) and EqualityComparable (Table 31) requirements and the expressions in Table 104 are valid and have the indicated semantics.

2 In Table 104, the term the domain of == is used in the ordinary mathematical sense to denote the set of values over which == is (required to be) defined. This set can change over time. Each algorithm places additional requirements on the domain of == for the iterator values it uses. These requirements can be inferred from the uses that algorithm makes of == and !=. [Example: the call find(a,b,x) is defined only if the value of a has the property p defined as follows: b has property p and a value i has property p if (*i==x) or if (*i!=x and ++i has property p). — end example]

Table 104 — Input iterator requirements (in addition to Iterator)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>a != b</td>
<td>contextually convertible to bool</td>
<td>!(a == b)</td>
<td>pre: (a, b) is in the domain of ==.</td>
</tr>
<tr>
<td>*a</td>
<td>convertible to T</td>
<td></td>
<td>pre: a is dereferenceable. The expression (void)*a, *a is equivalent to *a. If a == b and (a,b) is in the domain of == then *a is equivalent to *b.</td>
</tr>
<tr>
<td>a-&gt;m</td>
<td></td>
<td>(*a).m</td>
<td>pre: a is dereferenceable.</td>
</tr>
<tr>
<td>++r</td>
<td>X&amp;</td>
<td></td>
<td>pre: r is dereferenceable, post: r is dereferenceable or r is past-the-end. post: any copies of the previous value of r are no longer required either to be dereferenceable or to be in the domain of ==.</td>
</tr>
<tr>
<td>(void)r++</td>
<td></td>
<td></td>
<td>equivalent to (void)++r</td>
</tr>
<tr>
<td>*r++</td>
<td>convertible to T</td>
<td>{ T tmp = *r;    ++r;   return tmp; }</td>
<td></td>
</tr>
</tbody>
</table>

3 [Note: For input iterators, a == b does not imply ++a == ++b. (Equality does not guarantee the substi-
tution property or referential transparency.) Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be single pass algorithms. Value type T is not required to be a CopyAssignable type (Table 37). These algorithms can be used with istreams as the source of the input data through the istream_iterator class template. —end note]

### 24.2.4 Output iterators

1 A class or a built-in type X satisfies the requirements of an output iterator if X if X satisfies the Iterator requirements (24.2.2) and the expressions in Table 105 are valid and have the indicated semantics.

#### Table 105 — Output iterator requirements (in addition to Iterator)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>*r = o</td>
<td>result is not used</td>
<td></td>
<td>Remark: After this operation r is not required to be dereferenceable. post: r is incrementable.</td>
</tr>
<tr>
<td>++rX&amp; &amp;r == &amp;++r,</td>
<td>X&amp;</td>
<td></td>
<td>Remark: After this operation r is not required to be dereferenceable. post: r is incrementable.</td>
</tr>
<tr>
<td>r++convertible to</td>
<td>convertible to</td>
<td></td>
<td>Remark: After this operation r is not required to be dereferenceable. post: r is incrementable.</td>
</tr>
<tr>
<td></td>
<td>const X&amp;</td>
<td>{ X tmp = r; ++r; return tmp; }</td>
<td></td>
</tr>
<tr>
<td>*r++ = o</td>
<td>result is not used</td>
<td></td>
<td>Remark: After this operation r is not required to be dereferenceable. post: r is incrementable.</td>
</tr>
</tbody>
</table>

2 [Note: The only valid use of an operator* is on the left side of the assignment statement. Assignment through the same value of the iterator happens only once. Algorithms on output iterators should never attempt to pass through the same iterator twice. They should be single pass algorithms. Equality and inequality might not be defined. Algorithms that take output iterators can be used with ostreams as the destination for placing data through the ostream_iterator class as well as with insert iterators and insert pointers. —end note]

### 24.2.5 Forward iterators

1 A class or a built-in type X satisfies the requirements of a forward iterator if

- X satisfies the requirements of an input iterator (24.2.3),
- X satisfies the DefaultConstructible requirements (20.2.1),
- if X is a mutable iterator, reference is a reference to T; if X is a const iterator, reference is a reference to const T,
- the expressions in Table 106 are valid and have the indicated semantics, and
- objects of type X offer the multi-pass guarantee, described below.
The domain of \(==\) for forward iterators is that of iterators over the same underlying sequence.

Two dereferenceable iterators \(a\) and \(b\) of type \(X\) offer the **multi-pass guarantee** if:

1. \(a == b\) implies \(++a == ++b\) and
2. \(X\) is a pointer type or the expression \((\text{void})++X(a)\), \(*a\) is equivalent to the expression \(*a\).

[Note: The requirement that \(a == b\) implies \(++a == ++b\) (which is not true for input and output iterators) and the removal of the restrictions on the number of the assignments through a mutable iterator (which applies to output iterators) allows the use of multi-pass one-directional algorithms with forward iterators. — end note]

Table 106 — Forward iterator requirements (in addition to input iterator)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r++)</td>
<td>convertible to (X&amp;)</td>
<td>[ X \text{ tmp = } r; ++r; return \text{ tmp; } ]</td>
<td>pre-/post-condition</td>
</tr>
<tr>
<td>(*r++)</td>
<td>reference</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 bound to the same object.

### 24.2.6 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 107.

Table 107 — Bidirectional iterator requirements (in addition to forward iterator)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>(--r)</td>
<td>(X&amp;)</td>
<td>pre: there exists (s) such that (r == ++s). post: (r) is dereferenceable. (--(++r) == r). (--r == --s) implies (r == s). &amp;r == &amp;--r.</td>
<td></td>
</tr>
<tr>
<td>(--r--)</td>
<td>convertible to (X&amp;)</td>
<td>[ X \text{ tmp = } r; --r; return \text{ tmp; } ]</td>
<td></td>
</tr>
<tr>
<td>(*r--)</td>
<td>reference</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
24.2.7 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 108.

Table 108 — Random access iterator requirements (in addition to bidirectional iterator)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r += n</code></td>
<td><code>X</code></td>
<td><code>{ difference_type m = n; if (m &gt;= 0) while (m--) ++r; else while (m++) --r; return r; }</code></td>
<td><code>a + n == n + a</code></td>
</tr>
<tr>
<td><code>a + n</code></td>
<td><code>X</code></td>
<td><code>{ X tmp = a; return tmp += n; }</code></td>
<td><code>a + n == n + a</code></td>
</tr>
<tr>
<td><code>r -= n</code></td>
<td><code>X</code></td>
<td><code>return r += -n;</code></td>
<td><code>a - n</code></td>
</tr>
<tr>
<td><code>a - n</code></td>
<td><code>X</code></td>
<td><code>{ X tmp = a; return tmp -= n; }</code></td>
<td><code>b - a</code></td>
</tr>
<tr>
<td><code>b - a</code></td>
<td><code>difference_type</code></td>
<td><code>return n</code></td>
<td><code>b == a + (b - a)</code></td>
</tr>
<tr>
<td><code>a[n]</code></td>
<td>convertible to reference</td>
<td><code>*(a + n)</code></td>
<td><code>a &lt; b</code></td>
</tr>
<tr>
<td><code>a &lt; b</code></td>
<td>contextually convertible to bool</td>
<td><code>b - a &gt; 0</code></td>
<td><code>&lt;</code> is a total ordering relation</td>
</tr>
<tr>
<td><code>a &gt; b</code></td>
<td>contextually convertible to bool</td>
<td><code>b &lt; a</code></td>
<td><code>&gt;</code> is a total ordering relation opposite to <code>&lt;</code></td>
</tr>
<tr>
<td><code>a &gt;= b</code></td>
<td>contextually convertible to bool</td>
<td><code>!(a &lt; b)</code></td>
<td><code>a &lt;= b</code></td>
</tr>
<tr>
<td><code>a &lt;= b</code></td>
<td>contextually convertible to bool</td>
<td><code>!(a &gt; b)</code></td>
<td></td>
</tr>
</tbody>
</table>

24.3 Header `<iterator>` synopsis

```cpp
namespace std {
   // 24.4, primitives:
   template<class Iterator> struct iterator_traits;
   template<class T> struct iterator_traits<T*>;
```

§ 24.3
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 { };  

// 24.4.4, iterator operations:

template <class InputIterator, class Distance>
void advance(InputIterator& i, Distance n);

template <class InputIterator>
  typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);

template <class ForwardIterator>
  ForwardIterator next(ForwardIterator x,
        typename std::iterator_traits<ForwardIterator>::difference_type n = 1);

template <class BidirectionalIterator>
  BidirectionalIterator prev(BidirectionalIterator x,
        typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);

// 24.5, predefined iterators:

template <class Iterator> class reverse_iterator;

template <class Iterator1, class Iterator2>
bool operator<( 
           const reverse_iterator<Iterator1>& x,
           const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
bool operator<=(
           const reverse_iterator<Iterator1>& x,
           const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
bool operator>=(
           const reverse_iterator<Iterator1>& x,
           const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
bool operator> ( 
           const reverse_iterator<Iterator1>& x,
           const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
bool operator<=(
           const reverse_iterator<Iterator1>& x,
           const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
bool operator<=(
           const reverse_iterator<Iterator1>& x,
           const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
  auto operator-( 
           const reverse_iterator<Iterator1>& x,
           const reverse_iterator<Iterator2>& y) ->decltype(y.base() - x.base());

template <class Iterator>
reverse_iterator<Iterator>
    operator+(  
        typename reverse_iterator<Iterator>::difference_type n,  
        const reverse_iterator<Iterator>& x);

template <class Container> class back_insert_iterator;  
    template <class Container>  
        back_insert_iterator<Container> back_inserter(Container& x);

template <class Container> class front_insert_iterator;  
    template <class Container>  
        front_insert_iterator<Container> front_inserter(Container& x);

template <class Container> class insert_iterator;  
    template <class Container>  
        insert_iterator<Container> inserter(Container& x, typename Container::iterator i);

template <class Iterator> class move_iterator;  
    template <class Iterator1, class Iterator2>  
        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>  
        auto operator-(  
            const move_iterator<Iterator1>& x,  
            const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base());
    template <class Iterator>  
        move_iterator<Iterator> operator+(  
            typename move_iterator<Iterator>::difference_type n,  
            move_iterator<Iterator>& x);
    template <class Iterator>  
        move_iterator<Iterator> make_move_iterator(const Iterator& i);

    // 24.6, stream iterators:
    template <class T, class charT = char, class traits = char_traits<charT>,  
        class Distance = ptdiff_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>
```cpp
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;

// 24.6.5, range access:
template <class C> auto begin(C& c) -> decltype(c.begin());
template <class C> auto begin(const C& c) -> decltype(c.begin());
template <class C> auto end(C& c) -> decltype(c.end());
template <class C> auto end(const C& c) -> decltype(c.end());
template <class T, size_t N> T* begin(T (&array)[N]);
template <class T, size_t N> T* end(T (&array)[N]);
}
```

24.4 Iterator primitives

To simplify the task of defining iterators, the library provides several classes and functions:

24.4.1 Iterator traits

To implement algorithms only in terms of iterators, it is often necessary to determine the value and difference types that correspond to a particular iterator type. Accordingly, it is required that if Iterator is the type of an iterator, the types

```cpp
    iterator_traits<Iterator>::difference_type
    iterator_traits<Iterator>::value_type
    iterator_traits<Iterator>::iterator_category
```

be defined as the iterator’s difference type, value type and iterator category, respectively. In addition, the types

```cpp
    iterator_traits<Iterator>::reference
    iterator_traits<Iterator>::pointer
```

shall be defined as the iterator’s reference and pointer types, that is, for an iterator object a, the same type as the type of *a and a->, respectively. In the case of an output iterator, the types

```cpp
    iterator_traits<Iterator>::difference_type
    iterator_traits<Iterator>::value_type
    iterator_traits<Iterator>::reference
    iterator_traits<Iterator>::pointer
```

may be defined as void.
The template `iterator_traits<Iterator>` is defined as

```cpp
namespace std {
    template<class Iterator> struct iterator_traits {
        typedef typename Iterator::difference_type difference_type;
        typedef typename Iterator::value_type value_type;
        typedef typename Iterator::pointer pointer;
        typedef typename Iterator::reference reference;
        typedef typename Iterator::iterator_category iterator_category;
    };
}
```

It is specialized for pointers as

```cpp
namespace std {
    template<class T> struct iterator_traits<T*> {
        typedef ptrdiff_t difference_type;
        typedef T value_type;
        typedef T* pointer;
        typedef T& reference;
        typedef random_access_iterator_tag iterator_category;
    };
}
```

and for pointers to const as

```cpp
namespace std {
    template<class T> struct iterator_traits<const T*> {
        typedef ptrdiff_t difference_type;
        typedef T value_type;
        typedef const T* pointer;
        typedef const T& reference;
        typedef random_access_iterator_tag iterator_category;
    };
}
```

Note: If there is an additional pointer type `__far` such that the difference of two `__far` is of type `long`, an implementation may define

```cpp
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;
};
```

— end note

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) {
```
The `iterator_traits` template may be used as a base class to ease the definition of required types for new iterators.

```cpp
typename iterator_traits<BidirectionalIterator>::value_type
    tmp = *first;
    *first++ = *--last;
    *last = tmp;
    n -= 2;
}
}

— end example]

### 24.4.2 Basic iterator

The `iterator` template may be used as a base class to ease the definition of required types for new iterators.

```cpp
namespace std {
    template<class Category, class T, class Distance = ptrdiff_t,  
             class Pointer = T*, class Reference = T&>
    struct iterator {
        typedef T value_type;
        typedef Distance difference_type;
        typedef Pointer pointer;
        typedef Reference reference;
        typedef Category iterator_category;
    }
}
```

### 24.4.3 Standard iterator tags

It is often desirable for a function template specialization to find out what is the most specific category of its iterator argument, so that the function can select the most efficient algorithm at compile time. To facilitate this, the library introduces category tag classes which are used as compile time tags for algorithm selection. They are: `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag` and `random_access_iterator_tag`. For every iterator of type `Iterator`, `iterator_traits<Iterator>::iterator_category` shall be defined to be the most specific category tag that describes the iterator’s behavior.

```cpp
namespace std {
    struct input_iterator_tag { };
    struct output_iterator_tag { };
    struct forward_iterator_tag: public input_iterator_tag { };
    struct bidirectional_iterator_tag: public forward_iterator_tag { };
    struct random_access_iterator_tag: public bidirectional_iterator_tag { };
}
```

[Example: For a program-defined iterator `BinaryTreeIterator`, it could be included into the bidirectional iterator category by specializing the `iterator_traits` template:

```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*,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]

### 24.4.4 Iterator operations

[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);
```cpp```

2 Requires: `n` shall be negative only for bidirectional and random access iterators.

```cpp```
typeof iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```cpp```

4 Effects: If `InputIterator` meets the requirements of random access iterator, returns `(last - first)`; otherwise, returns the number of increments needed to get from `first` to `last`.
Requires: If InputIterator meets the requirements of random access iterator, last shall be reachable from first or first shall be reachable from last; otherwise, last shall be reachable from first.

```
template <class ForwardIterator>
ForwardIterator next(ForwardIterator x,
    typename std::iterator_traits<ForwardIterator>::difference_type n = 1);
```

Effects: Equivalent to advance(x, n); return x;

```
template <class BidirectionalIterator>
BidirectionalIterator prev(BidirectionalIterator x,
    typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);
```

Effects: Equivalent to advance(x, -n); return x;

### 24.5 Iterator adaptors

#### 24.5.1 Reverse iterators

Class template `reverse_iterator` is an iterator adaptor that iterates from the end of the sequence defined by its underlying iterator to the beginning of that sequence. The fundamental relation between a reverse iterator and its corresponding iterator `i` is established by the identity: `&*(reverse_iterator(i)) == &*(i - 1)`.

#### 24.5.1.1 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> {
        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);
```

§ 24.5.1.1
reverse_iterator operator- (difference_type n) const;
reverse_iterator& operator-=(difference_type n);
unspecified operator[] (difference_type n) const;
protected:
  Iterator current;
private:
  Iterator deref_tmp;  // exposition only
};

template <class Iterator1, class Iterator2>
bool operator==(const reverse_iterator<Iterator1>& x,
               const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
bool operator<(const reverse_iterator<Iterator1>& x,
               const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
bool operator!=(const reverse_iterator<Iterator1>& x,
               const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
bool operator>(const reverse_iterator<Iterator1>& x,
               const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
bool operator>= (const reverse_iterator<Iterator1>& x,
                const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
bool operator<=(const reverse_iterator<Iterator1>& x,
                const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
auto operator-(const reverse_iterator<Iterator1>& x,
               const reverse_iterator<Iterator2>& y) -> decltype(y.current - x.current);

template <class Iterator>
reverse_iterator<Iterator> operator+(typename reverse_iterator<Iterator>::difference_type n,
                      const reverse_iterator<Iterator>& x);

24.5.1.2 reverse_iterator requirements

1 The template parameter Iterator shall meet all the requirements of a Bidirectional Iterator (24.2.6).

2 Additionally, Iterator shall meet the requirements of a Random Access Iterator (24.2.7) if any of the members operator+ (24.5.1.3.8), operator- (24.5.1.3.10), operator+= (24.5.1.3.9), operator-= (24.5.1.3.11), operator [] (24.5.1.3.12), or the global operators operator< (24.5.1.3.14), operator> (24.5.1.3.16), operator <= (24.5.1.3.17), operator >= (24.5.1.3.18), operator< (24.5.1.3.19) or operator+ (24.5.1.3.20). is referenced in a way that
requires instantiation (14.7.1).

24.5.1.3 reverse_iterator operations

24.5.1.3.1 reverse_iterator constructor

reverse_iterator();
1    Effects: Value initializes current. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type Iterator.

explicit reverse_iterator(Iterator x);
2    Effects: Initializes current with x.

template <class U> reverse_iterator(const reverse_iterator<U> &u);
3    Effects: Initializes current with u.current.

24.5.1.3.2 reverse_iterator::operator=

template <class U>
reverse_iterator&
operator=(const reverse_iterator<U>& u);
1    Effects: Assigns u.base() to current.
2    Returns: *this.

24.5.1.3.3 Conversion

Iterator base() const;    // explicit
1    Returns: current.

24.5.1.3.4 operator*

reference operator*() const;
1    Effects:
        deref_tmp = current;
        --deref_tmp;
        return *deref_tmp;
2    [ Note: This operation must use an auxiliary member variable rather than a temporary variable to avoid returning a reference that persists beyond the lifetime of its associated iterator. (See 24.2.) — end note ]

24.5.1.3.5 operator->

pointer operator->() const;
1    Returns: &(operator*()).
24.5.1.3.6 operator++

reverse_iterator& operator++();
1   Effects:  --current;
2   Returns: *this.

reverse_iterator operator++(int);
3   Effects:
       reverse_iterator tmp = *this;
       --current;
       return tmp;

24.5.1.3.7 operator--

reverse_iterator& operator--();
1   Effects:  ++current
2   Returns: *this.

reverse_iterator operator--(int);
3   Effects:
       reverse_iterator tmp = *this;
       ++current;
       return tmp;

24.5.1.3.8 operator+

reverse_iterator
operator+(typename reverse_iterator<Iterator>::difference_type n) const;
1   Returns: reverse_iterator(current-n).

24.5.1.3.9 operator+=

reverse_iterator&
operator+=(typename reverse_iterator<Iterator>::difference_type n);
1   Effects:  current -= n;
2   Returns: *this.

24.5.1.3.10 operator-

reverse_iterator
operator-(typename reverse_iterator<Iterator>::difference_type n) const;
1   Returns: reverse_iterator(current+n).
24.5.1.3.11 operator-=`

reverse_iterator&
operator-=(typename reverse_iterator<Iterator>::difference_type n);

1   **Effects:** current += n;
2   **Returns:** *this.

24.5.1.3.12 operator[]

unspecified operator[](typename reverse_iterator<Iterator>::difference_type n) const;

1   **Returns:** current[-n-1].

24.5.1.3.13 operator==

template <class Iterator1, class Iterator2>
bool operator==(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

1   **Returns:** x.current == y.current.

24.5.1.3.14 operator<

template <class Iterator1, class Iterator2>
bool operator<(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

1   **Returns:** x.current > y.current.

24.5.1.3.15 operator!=

template <class Iterator1, class Iterator2>
bool operator!=(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

1   **Returns:** x.current != y.current.

24.5.1.3.16 operator>

template <class Iterator1, class Iterator2>
bool operator>(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

1   **Returns:** x.current < y.current.
24.5.1.3.17  operator>=

template <class Iterator1, class Iterator2>
bool operator>=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);

Returns: x.current <= y.current.

24.5.1.3.18  operator<=

template <class Iterator1, class Iterator2>
bool operator<=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);

Returns: x.current >= y.current.

24.5.1.3.19  operator-

template <class Iterator1, class Iterator2>
auto operator-(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y) -> decltype(y.current - x.current);

Returns: y.current - x.current.

24.5.1.3.20  operator+

template <class Iterator>
reverse_iterator<Iterator> operator+(  
    typename reverse_iterator<Iterator>::difference_type n,  
    const reverse_iterator<Iterator>& x);

Returns: reverse_iterator<Iterator> (x.current - n).

24.5.2  Insert iterators

To make it possible to deal with insertion in the same way as writing into an array, a special kind of iterator adaptors, called insert iterators, are provided in the library. With regular iterator classes,

while (first != last) *result++ = *first++;

causes a range [first, last) to be copied into a range starting with result. The same code with result being an insert iterator will insert corresponding elements into the container. This device allows all of the copying algorithms in the library to work in the insert mode instead of the regular overwrite mode.

An insert iterator is constructed from a container and possibly one of its iterators pointing to where insertion takes place if it is neither at the beginning nor at the end of the container. Insert iterators satisfy the requirements of output iterators. operator* returns the insert iterator itself. The assignment operator=(const T& x) is defined on insert iterators to allow writing into them, it inserts x right before where the insert iterator is pointing. In other words, an insert iterator is like a cursor pointing into the container where the insertion takes place. back_insert_iterator inserts elements at the end of a container, front_insert_iterator inserts elements at the beginning of a container, and insert_iterator inserts elements where
the iterator points to in a container. back_inserter, front_inserter, and inserter are three functions making the insert iterators out of a container.

24.5.2.1 Class template back_insert_iterator

```cpp
namespace std {
    template <class Container>
    class back_insert_iterator :
        public iterator<output_iterator_tag,void,void,void,void> {
    protected:
        Container* container;

    public:
        typedef Container container_type;
        explicit back_insert_iterator(Container& x);
        back_insert_iterator<Container>&
            operator=(typename Container::const_reference value);
        back_insert_iterator<Container>&
            operator=(typename Container::value_type&& value);
        back_insert_iterator<Container>& operator*();
        back_insert_iterator<Container>& operator++();
        back_insert_iterator<Container> operator++(int);
    }

    template <class Container>
    back_insert_iterator<Container> back_inserter(Container& x);
}
Returns: *this.

24.5.2.2.4 back_insert_iterator::operator++

back_insert_iterator<Container>& operator++();
back_insert_iterator<Container> operator++(int);

Returns: *this.

24.5.2.2.5 back_inserter

template <class Container>
back_insert_iterator<Container> back_inserter(Container& x);

Returns: back_insert_iterator<Container>(x).

24.5.2.3 Class template front_insert_iterator

namespace std {
  template <class Container>
  class front_insert_iterator :
    public iterator<output_iterator_tag,void,void,void,void> {
  protected:
    Container* container;

  public:
    typedef Container container_type;
    explicit front_insert_iterator(Container& x);
    front_insert_iterator<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);
}

24.5.2.4 front_insert_iterator operations

24.5.2.4.1 front_insert_iterator constructor

explicit front_insert_iterator(Container& x);

Effects: Initializes container with &x.

24.5.2.4.2 front_insert_iterator::operator=

front_insert_iterator<Container>&
  operator=(typename Container::const_reference value);
1. \textit{Effects:} \texttt{container->push\_front(value)};

2. \textit{Returns:} \texttt{*this}.

\begin{verbatim}
front_insert_iterator<Container>&
operator=(typename Container::value_type&& value);
\end{verbatim}

3. \textit{Effects:} \texttt{container->push\_front(std::move(value))};

4. \textit{Returns:} \texttt{*this}.

\begin{verbatim}
24.5.2.4.3 front_insert_iterator::operator*
\end{verbatim}

\begin{verbatim}
front_insert_iterator<Container>& operator*();
\end{verbatim}

1. \textit{Returns:} \texttt{*this}.

\begin{verbatim}
24.5.2.4.4 front_insert_iterator::operator++
\end{verbatim}

\begin{verbatim}
front_insert_iterator<Container>& operator++();
front_insert_iterator<Container>& operator++(int);
\end{verbatim}

1. \textit{Returns:} \texttt{*this}.

\begin{verbatim}
24.5.2.4.5 front_inserter
\end{verbatim}

\begin{verbatim}
template <class Container>
    front_insert_iterator<Container> front_inserter(Container& x);
\end{verbatim}

1. \textit{Returns:} \texttt{front_insert_iterator<Container>(x)}.

\begin{verbatim}
24.5.2.5 Class template insert_iterator
\end{verbatim}

\begin{verbatim}
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>
        insert_iterator<Container> inserter(Container& x, typename Container::iterator i);

\section*{§ 24.5.2.5}
\end{verbatim}

830
24.5.2.6 insert_iterator operations [insert.iter.ops]

24.5.2.6.1 insert_iterator constructor [insert.iter.cons]

    insert_iterator(Container& x, typename Container::iterator i);

1       Effects: Initializes container with &x and iter with i.

24.5.2.6.2 insert_iterator::operator=[insert.iter.op=]

    insert_iterator<Container>&
    operator=(typename Container::const_reference value);

1       Effects:
       iter = container->insert(iter, value);
       ++iter;

2       Returns: *this.

    insert_iterator<Container>&
    operator=(typename Container::value_type&& value);

3       Effects:
       iter = container->insert(iter, std::move(value));
       ++iter;

4       Returns: *this.

24.5.2.6.3 insert_iterator::operator* [insert.iter.op*]

    insert_iterator<Container>& operator*();

1       Returns: *this.

24.5.2.6.4 insert_iterator::operator++ [insert.iter.op++]

    insert_iterator<Container>& operator++();
    insert_iterator<Container>& operator++(int);

1       Returns: *this.

24.5.2.6.5 inserter [inserter]

    template <class Container>
    insert_iterator<Container> inserter(Container& x, typename Container::iterator i);

1       Returns: insert_iterator<Container>(x, i).
24.5.3 Move iterators

Class template `move_iterator` is an iterator adaptor with the same behavior as the underlying iterator except that its 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
class std::move_iterator
```

24.5.3.1 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 Iterator pointer;
            typedef typename iterator_traits<Iterator>::value_type value_type;
            typedef typename iterator_traits<Iterator>::iterator_category iterator_category;
            typedef value_type&& reference;

            move_iterator();
            explicit move_iterator(Iterator i);
            template <class U> move_iterator(const move_iterator<U>& u);
            template <class U> move_iterator& operator=(const move_iterator<U>& u);

            iterator_type base() const;
            reference operator*() const;
            pointer operator->() const;

            move_iterator& operator++();
            move_iterator operator++(int);
            move_iterator& operator--();
            move_iterator operator--(int);

            move_iterator operator+(difference_type n) const;
            move_iterator& operator+=(difference_type n);
            move_iterator operator-(difference_type n) const;
            move_iterator& operator-=(difference_type n);

            unspecified operator[](difference_type n) const;

        private:
            Iterator current;  // exposition only
        }
    }
```

§ 24.5.3.1
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>
    auto operator-(
        const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y) ->
        decltype(x.base() - y.base());

    template <class Iterator>
    move_iterator<Iterator> operator+;

    template <class Iterator>
    move_iterator<Iterator> make_move_iterator(const Iterator& i);

24.5.3.2 move_iterator requirements

1 The template parameter Iterator shall meet the requirements for an Input Iterator (24.2.3). Additionally, if any of the bidirectional or random access traversal functions are instantiated, the template parameter shall meet the requirements for a Bidirectional Iterator (24.2.6) or a Random Access Iterator (24.2.7), respectively.

24.5.3.3 move_iterator operations

24.5.3.3.1 move_iterator constructors

move_iterator();

1 Effects: Constructs a move_iterator, value initializing current. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type Iterator.

explicit move_iterator(Iterator i);

2 Effects: Constructs a move_iterator, initializing current with i.

template <class U> move_iterator(const move_iterator<U>& u);

3 Effects: Constructs a move_iterator, initializing current with u.base().

4 Requires: U shall be convertible to Iterator.

24.5.3.3.2 move_iterator::operator=

§ 24.5.3.3.2
template <class U> move_iterator& operator=(const move_iterator<U>& u);

Effects: Assigns u.base() to current.

Requires: U shall be convertible to Iterator.

24.5.3.3.3 move_iterator conversion

Iterator base() const;

Returns: current.

24.5.3.3.4 move_iterator::operator*

reference operator*() const;

Returns: std::move(*current).

24.5.3.3.5 move_iterator::operator->

pointer operator->() const;

Returns: current.

24.5.3.3.6 move_iterator::operator++

move_iterator& operator++();

Effects: ++current.

Returns: *this.

move_iterator operator++(int);

Effects:
    move_iterator tmp = *this;
    ++current;
    return tmp;

24.5.3.3.7 move_iterator::operator--

move_iterator& operator--();

Effects: --current.

Returns: *this.

move_iterator operator--(int);

Effects:
    move_iterator tmp = *this;
    --current;
    return tmp;
24.5.3.3.8  move_iterator::operator+

move_iterator operator+(difference_type n) const;
1    Returns: move_iterator(current + n).

24.5.3.3.9  move_iterator::operator+=

move_iterator& operator+=(difference_type n);
1    Effects: current += n.
2    Returns: *this.

24.5.3.3.10 move_iterator::operator-

move_iterator operator-(difference_type n) const;
1    Returns: move_iterator(current - n).

24.5.3.3.11 move_iterator::operator==

move_iterator& operator==(difference_type n);
1    Effects: current -= n.
2    Returns: *this.

24.5.3.3.12 move_iterator::operator[]

unspecifed operator[](difference_type n) const;
1    Returns: std::move(current[n]).

24.5.3.3.13 move_iterator comparisons

template <class Iterator1, class Iterator2>
bool operator==(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
1    Returns: x.base() == y.base().

template <class Iterator1, class Iterator2>
bool operator!=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
2    Returns: !(x == y).

template <class Iterator1, class Iterator2>
bool operator<(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
3    Returns: x.base() < y.base().

template <class Iterator1, class Iterator2>
bool operator<=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
4    Returns: !(y < x).
template <class Iterator1, class Iterator2>
bool operator>(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

Returns: y < x.

template <class Iterator1, class Iterator2>
bool operator>=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

Returns: !(x < y).

## 24.5.3.3.14 move_iterator non-member functions

### template <class Iterator1, class Iterator2>

```cpp
auto operator-(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base());
```

Returns: x.base() - y.base().

### template <class Iterator1, class Iterator2>

```cpp
move_iterator<Iterator1> operator+(typename move_iterator<Iterator1>::difference_type n, const move_iterator<Iterator1>& x);
```

Returns: x + n.

### template <class Iterator1, class Iterator2>

```cpp
move_iterator<Iterator1> make_move_iterator(const Iterator1& i);
```

Returns: move_iterator<Iterator1>(i).

## 24.6 Stream iterators

To make it possible for algorithmic templates to work directly with input/output streams, appropriate iterator-like class templates are provided.

[Example:

```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]

### 24.6.1 Class template istream_iterator

The class template `istream_iterator` is an input iterator (24.2.3) that reads (using `operator>>`) successive elements from the input stream for which it was constructed. After it is constructed, and every time `++` is used, the iterator reads and stores a value of `T`. If the iterator fails to read and store a value of `T` (`fail()` on the stream returns `true`), the iterator becomes equal to the end-of-stream iterator value. The constructor with no arguments `istream_iterator()` always constructs an end-of-stream input iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end-of-stream iterator is not defined. For any other iterator value a `const T&` is returned. The result of `operator->` on an end-of-stream iterator is not defined. For any other iterator value a `const T*` is returned. The behavior of a program that applies `operator++()` to an end-of-stream iterator is undefined. It is impossible to store things into istream iterators.
Two end-of-stream iterators are always equal. An end-of-stream iterator is not equal to a non-end-of-stream iterator. Two non-end-of-stream iterators are equal when they are constructed from the same stream.

```cpp
namespace std {
    template <class T, class charT = char, class traits = char_traits<charT>,
              class Distance = ptrdiff_t>
    class istream_iterator:
    public iterator<input_iterator_tag, T, Distance, const T*, const T&> {
    public:
        typedef charT char_type;
        typedef traits traits_type;
        typedef basic_istream<charT,traits> istream_type;
        see below istream_iterator();
        istream_iterator(istream_type& s);
        istream_iterator(const istream_iterator& x) = default;
    ~istream_iterator() = default;

        const T& operator*() const;
        const T* operator->() const;
        istream_iterator<T,charT,traits,Distance>& operator++();
        istream_iterator<T,charT,traits,Distance> operator++(int);
    private:
        basic_istream<charT,traits>* in_stream; // exposition only
        T value; // exposition only
    }

    template <class T, class charT, class traits, class Distance>
    bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
                    const istream_iterator<T,charT,traits,Distance>& y);
    template <class T, class charT, class traits, class Distance>
    bool operator!=(const istream_iterator<T,charT,traits,Distance>& x,
                    const istream_iterator<T,charT,traits,Distance>& y);
};
```

### 24.6.1.1 istream_iterator constructors and destructor

**24.6.1.1** istream_iterator constructors and destructor

[istream.iterator.cons]

`see below istream_iterator();`

1  **Effects:** Constructs the end-of-stream iterator. If `T` is a literal type, then this constructor shall be a `constexpr` constructor.

2  **Postcondition:** `in_stream == 0`.

`istream_iterator(istream_type& s);`

3  **Effects:** Initializes `in_stream` with `&s`. `value` may be initialized during construction or the first time it is referenced.

4  **Postcondition:** `in_stream == &s`.

`istream_iterator(const istream_iterator& x) = default;`

5  **Effects:** Constructs a copy of `x`. If `T` is a literal type, then this constructor shall be a trivial copy constructor.

6  **Postcondition:** `in_stream == x.in_stream`.

`~istream_iterator() = default;`
Effects: The iterator is destroyed. If \( T \) is a literal type, then this destructor shall be a trivial destructor.

24.6.1.2 \texttt{istream\_iterator} operations

\begin{verbatim}
const T& operator*() const;
Returns: value.
const T* operator->() const;
Returns: &operator*( ).
istream\_iterator<T,charT,traits,Distance>& operator++();
Requires: \texttt{in\_stream} \(!=!\ 0\).
Effects: \texttt{*in\_stream} >> value.
Returns: \texttt{*this}.
istream\_iterator<T,charT,traits,Distance>& operator++(int);
Requires: \texttt{in\_stream} \(!=!\ 0\).
Effects:
\begin{verbatim}
    istream\_iterator<T,charT,traits,Distance> tmp = \texttt{*this};
    \texttt{*in\_stream} >> value;
    return (tmp);
\end{verbatim}
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: \texttt{x.in\_stream} == \texttt{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)
\end{verbatim}

24.6.2 Class template \texttt{ostream\_iterator}

\texttt{ostream\_iterator} writes (using \texttt{operator<<}) successive elements onto the output stream from which it was constructed. If it was constructed with \texttt{charT*} as a constructor argument, this string, called a \textit{delimiter string}, is written to the stream after every \( T \) is written. It is not possible to get a value out of the output iterator. Its only use is as an output iterator in situations like
\begin{verbatim}
while (first != last)
    \*result++ = \*first++;
\end{verbatim}

\texttt{ostream\_iterator} is defined as:
\begin{verbatim}
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:
\end{verbatim}

§ 24.6.2
typedef char char_type;
typedef traits traits_type;
typedef basic_ostream<charT,traits> ostream_type;
ostream_iterator(ostream_type& s);
ostream_iterator(ostream_type& s, const charT* delimiter);
ostream_iterator(const ostream_iterator<T,charT,traits>& x);
~ostream_iterator();
ostream_iterator<T,charT,traits>& operator=(const T& value);
ostream_iterator<T,charT,traits>& operator*();
ostream_iterator<T,charT,traits>& operator++();
ostream_iterator<T,charT,traits>& operator++(int);

private:
  basic_ostream<charT,traits>* out_stream; // exposition only
  const charT* delim; // exposition only
};

24.6.2.1 ostream_iterator constructors and destructor

ostream_iterator(ostream_type& s); 1
Effects: Initializes out_stream with &s and delim with null.
ostream_iterator(ostream_type& s, const charT* delimiter); 2
Effects: Initializes out_stream with &s and delim with delimiter.
ostream_iterator(const ostream_iterator& x); 3
Effects: Constructs a copy of x.
~ostream_iterator(); 4
Effects: The iterator is destroyed.

24.6.2.2 ostream_iterator operations

ostream_iterator& operator=(const T& value); 1
Effects:
  *out_stream << value;
  if(delim != 0)
    *out_stream << delim;
  return (*this);
ostream_iterator& operator*(); 2
Returns: *this.
ostream_iterator& operator++();
ostream_iterator& operator++(int); 3
Returns: *this.
24.6.3 Class template istreambuf_iterator

The class template `istreambuf_iterator` defines an input iterator (24.2.3) that reads successive characters from the streambuf for which it was constructed. `operator*` provides access to the current input character, if any. [Note: `operator->` may return a proxy. — end note] Each time `operator++` is evaluated, the iterator advances to the next input character. If the end of stream is reached (streambuf_type::sgetc() returns `traits::eof()`), the iterator becomes equal to the end-of-stream iterator value. The default constructor `istreambuf_iterator()` and the constructor `istreambuf_iterator(0)` both construct an end-of-stream iterator object suitable for use as an end-of-range. All specializations of `istreambuf_iterator` shall have a trivial copy constructor, a constexpr default constructor, and a trivial destructor.

The result of `operator*()` on an end-of-stream iterator is undefined. For any other iterator value a `char_`-type value is returned. It is impossible to assign a character via an input iterator.

```cpp
namespace std {
    template<class charT, class traits = char_traits<charT> >
    class istreambuf_iterator
    : public iterator<input_iterator_tag, charT, typename traits::off_type, unspecified, charT> {
        public:
            typedef charT char_type;
            typedef traits traits_type;
            typedef typename traits::int_type int_type;
            typedef basic_streambuf<charT,traits> streambuf_type;
            typedef basic_istream<charT,traits> istream_type;

            class proxy;       // exposition only

            constexpr istreambuf_iterator() throw();
            istreambuf_iterator(const istreambuf_iterator&) throw() = default;
            ~istreambuf_iterator() throw() = default;
            istreambuf_iterator(istream_type& s) throw();
            istreambuf_iterator(istream_type* s) throw();
            istreambuf_iterator(const proxy& p) throw();
            charT operator*() const;
            pointer operator->() const;
            istreambuf_iterator<charT,traits>& operator++();
            proxy operator++(int);
            bool equal(const istreambuf_iterator& b) const;

        private:
            streambuf_type* sbuf_;       // exposition only
        };

    template <class charT, class traits>
    bool operator==(const istreambuf_iterator<charT,traits>& a,
                const istreambuf_iterator<charT,traits>& b);

    template <class charT, class traits>
    bool operator!=(const istreambuf_iterator<charT,traits>& a,
                const istreambuf_iterator<charT,traits>& b);

    }

24.6.3.1 Class template istreambuf_iterator::proxy

Namespace std {
    template <class charT, class traits = char_traits<charT> >
    class istreambuf_iterator<charT, traits>::proxy {

§ 24.6.3.1 840
```
charT keep_;
basic_streambuf<charT,traits>* sbuf_;
proxy(charT c,
    basic_streambuf<charT,traits>* sbuf)
    : keep_(c), sbuf_(sbuf) { }
public:
    charT operator*() { return keep_; }
};
```

1. Class `istreambuf_iterator<charT,traits>::proxy` is for exposition only. An implementation is permitted to provide equivalent functionality without providing a class with this name. Class `istreambuf_iterator<charT,traits>::proxy` provides a temporary placeholder as the return value of the post-increment operator (`operator++`). It keeps the character pointed to by the previous value of the iterator for some possible future access to get the character.

24.6.3.2 `istreambuf_iterator` constructors

```cpp
constexpr istreambuf_iterator() throw();
```

1. **Effects:** Constructs the end-of-stream iterator.

```cpp
istreambuf_iterator(basic_istream<charT,traits>& s) throw();
istreambuf_iterator(basic_streambuf<charT,traits>* s) throw();
```

2. **Effects:** Constructs an `istreambuf_iterator<>` that uses the `basic_streambuf<>` object *(s.rdbuf()), or *s, respectively. Constructs an end-of-stream iterator if s.rdbuf() is null.

```cpp
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.

24.6.3.3 `istreambuf_iterator`: operator*

```cpp
charT operator*() const
```

1. **Returns:** The character obtained via the streambuf member `sbuf_->sgetc()`.

24.6.3.4 `istreambuf_iterator`: operator++

```cpp
istreambuf_iterator<charT,traits>&
    istreambuf_iterator<charT,traits>::operator++();
```

1. **Effects:** `sbuf_->sbumpc()`.

2. **Returns:** *this.

```cpp
proxy istreambuf_iterator<charT,traits>::operator++(int);
```

3. **Returns:** `proxy(sbuf_->sbumpc(), sbuf_)`.

24.6.3.5 `istreambuf_iterator`: equal

```cpp
bool equal(const istreambuf_iterator<charT,traits>& b) const;
```

§ 24.6.3.5
24.6.3.6 \texttt{operator==} \hfill \texttt{[istreambuf.iterator::op==]}

\begin{verbatim}
template <class charT, class traits>
bool operator===(const istreambuf_iterator<charT,traits>& a,
                 const istreambuf_iterator<charT,traits>& b);
\end{verbatim}

\textit{Returns:} \texttt{a.equal(b)}.

24.6.3.7 \texttt{operator!=} \hfill \texttt{[istreambuf.iterator::op!=]}

\begin{verbatim}
template <class charT, class traits>
bool operator!=(const istreambuf_iterator<charT,traits>& a,
                 const istreambuf_iterator<charT,traits>& b);
\end{verbatim}

\textit{Returns:} \texttt{!a.equal(b)}.

24.6.4 Class template \texttt{ostreambuf\_iterator} \hfill \texttt{[ostreambuf.iterator]}

\begin{verbatim}
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
            
        }
}
\end{verbatim}


The class template \texttt{ostreambuf\_iterator} writes successive \textit{characters} onto the output stream from which it was constructed. It is not possible to get a character value out of the output iterator.

24.6.4.1 \texttt{ostreambuf\_iterator} constructors \hfill \texttt{[ostreambuf.iter.cons]}

\begin{verbatim}
ostreambuf_iterator(ostream_type& s) throw();
\end{verbatim}

\textit{Requires:} \texttt{s.rdbuf()} shall not null pointer.
Effects: :sbuf_(s.rdbuf()) {}.

ostreambuf_iterator(streambuf_type* s) throw();

Requires: s shall not be a null pointer.

Effects: : sbuf_(s) {}.

24.6.4.2 ostreambuf_iterator operations

ostreambuf_iterator<charT,traits>&
operator=(charT c);

Effects: If failed() yields false, calls sbuf_\rightarrow sputc(c); otherwise has no effect.

Returns: *this.

ostreambuf_iterator<longcharT,traits>>& operator*();

Returns: *this.

ostreambuf_iterator<charT,traits>& operator++();
ostreambuf_iterator<charT,traits>& operator++(int);

Returns: *this.

bool failed() const throw();

Returns: true if in any prior use of member operator=, the call to sbuf_\rightarrow sputc() returned traits::eof(); or false otherwise.

24.6.5 range access

In addition to being available via inclusion of the <iterator> header, the function templates in 24.6.5 are available when any of the following headers are included: <array>, <deque>, <forward_list>, <list>, <map>, <regex>, <set>, <string>, <unordered_map>, <unordered_set>, and <vector>.

template <class C> auto begin(C& c) -> decltype(c.begin());
template <class C> auto begin(const C& c) -> decltype(c.begin());

Returns: c.begin().

template <class C> auto end(C& c) -> decltype(c.end());
template <class C> auto end(const C& c) -> decltype(c.end());

Returns: c.end().

template <class T, size_t N> T* begin(T (&array)[N]);

Returns: array.

template <class T, size_t N> T* end(T (&array)[N]);

Returns: array + N.
25 Algorithms library

25.1 General

This Clause describes components that C++ programs may use to perform algorithmic operations on containers (Clause 23) and other sequences.

The following subclauses describe components for non-modifying sequence operation, modifying sequence operations, sorting and related operations, and algorithms from the ISO C library, as summarized in Table 109.

Table 109 — Algorithms library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.2 Non-modifying sequence operations</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>25.3 Mutating sequence operations</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>25.4 Sorting and related operations</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>25.5 C library algorithms</td>
<td>&lt;cstdlib&gt;</td>
</tr>
</tbody>
</table>

Header <algorithm> synopsis

```cpp
namespace std {

  #include <initializer_list>

  // 25.2, non-modifying sequence operations:
  template <class InputIterator, class Predicate>
  bool all_of(InputIterator first, InputIterator last, Predicate pred);
  template <class InputIterator, class Predicate>
  bool any_of(InputIterator first, InputIterator last, Predicate pred);
  template <class InputIterator, class Predicate>
  bool none_of(InputIterator first, InputIterator last, Predicate pred);

  template <class InputIterator, class Function>
  Function for_each(InputIterator first, InputIterator last, Function f);
  template <class InputIterator, class T>
  InputIterator find(InputIterator first, InputIterator last, const T& value);
  template <class InputIterator, class Predicate>
  InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);
  template <class InputIterator, class Predicate>
  InputIterator find_if_not(InputIterator first, InputIterator last, Predicate pred);
  template <class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
  template <class ForwardIterator1, class ForwardIterator2,
            class BinaryPredicate>
  ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);

§ 25.1
844
```
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);

typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& value);

typename iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, 
   Predicate pred);

pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1, 
   InputIterator2 first2);

pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1, 
   InputIterator2 first2, BinaryPredicate pred);

bool equal(InputIterator1 first1, InputIterator1 last1, 
   InputIterator2 first2);

bool equal(InputIterator1 first1, InputIterator1 last1, 
   InputIterator2 first2, BinaryPredicate pred);

bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1, 
   ForwardIterator2 first2);

bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1, 
   ForwardIterator2 first2, BinaryPredicate pred);
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
ForwardIterator1 search(
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2,
    BinaryPredicate pred);

template<class ForwardIterator, class Size, class T>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                        Size count, const T& value);

template<class ForwardIterator, class Size, class T, class BinaryPredicate>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                        Size count, const T& value,
                        BinaryPredicate pred);

// 25.3, modifying sequence operations:
// 25.3.1, copy:
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result);

template<class InputIterator, class Size, class OutputIterator>
OutputIterator copy_n(InputIterator first, Size n, OutputIterator result);

template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator copy_if(InputIterator first, InputIterator last, OutputIterator result, Predicate pred);

template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(   
    BidirectionalIterator1 first, BidirectionalIterator1 last, 
    BidirectionalIterator2 result);

// 25.3.2, move:
template<class InputIterator, class OutputIterator>
OutputIterator move(InputIterator first, InputIterator last, OutputIterator result);

template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 move_backward(   
    BidirectionalIterator1 first, BidirectionalIterator1 last, 
    BidirectionalIterator2 result);

// 25.3.3, swap:
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2);

template<class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);

template<class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last, OutputIterator result, UnaryOperation op);
template<class InputIterator1, class InputIterator2, class OutputIterator,
    class BinaryOperation>
    OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, OutputIterator result,
        BinaryOperation binary_op);

template<class ForwardIterator, class T>
    void replace(ForwardIterator first, ForwardIterator last,
        const T& old_value, const T& new_value);

template<class ForwardIterator, class Predicate, class T>
    void replace_if(ForwardIterator first, ForwardIterator last,
        Predicate pred, const T& new_value);

template<class InputIterator, class OutputIterator, class T>
    OutputIterator replace_copy(InputIterator first, InputIterator last,
        OutputIterator result,
        const T& old_value, const T& new_value);

template<class InputIterator, class OutputIterator, class Predicate, class T>
    OutputIterator replace_copy_if(InputIterator first, InputIterator last,
        OutputIterator result,
        Predicate pred, const T& new_value);

template<class ForwardIterator, class T>
    void fill(ForwardIterator first, ForwardIterator last, const T& value);

template<class OutputIterator, class Size, class T>
    OutputIterator fill_n(OutputIterator first, Size n, const T& value);

template<class ForwardIterator, class Generator>
    void generate(ForwardIterator first, ForwardIterator last,
        Generator gen);

template<class OutputIterator, class Size, class Generator>
    OutputIterator generate_n(OutputIterator first, Size n, Generator gen);

template<class ForwardIterator, class T>
    ForwardIterator remove(ForwardIterator first, ForwardIterator last,
        const T& value);

template<class ForwardIterator, class Predicate>
    ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
        Predicate pred);

template<class InputIterator, class OutputIterator, class T>
    OutputIterator remove_copy(InputIterator first, InputIterator last,
        OutputIterator result, const T& value);

template<class InputIterator, class OutputIterator, class Predicate>
    OutputIterator remove_copy_if(InputIterator first, InputIterator last,
        OutputIterator result, Predicate pred);

template<class ForwardIterator>
    ForwardIterator unique(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
    ForwardIterator unique(ForwardIterator first, ForwardIterator last,
        BinaryPredicate pred);

template<class InputIterator, class OutputIterator>
    OutputIterator unique_copy(InputIterator first, InputIterator last,
        OutputIterator result);

template<class InputIterator, class OutputIterator, class BinaryPredicate>
    OutputIterator unique_copy(InputIterator first, InputIterator last,
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);

template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
BidirectionalIterator last,
OutputIterator result);

// 25.4, sorting and related operations:
// 25.4.1, sorting:

// 25.3.13, partitions:

// 25.3.13, partitions:
void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

template<class RandomAccessIterator>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last, Compare comp);

template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(
    InputIterator first, InputIterator last, 
    RandomAccessIterator result_first, RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator, class Compare>
RandomAccessIterator partial_sort_copy(
    InputIterator first, InputIterator last, 
    RandomAccessIterator result_first, RandomAccessIterator result_last, 
    Compare comp);

template<class ForwardIterator>
bool is_sorted(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
bool is_sorted(ForwardIterator first, ForwardIterator last, Compare comp);

template<class ForwardIterator>
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last, Compare comp);

template<class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last, Compare comp);

// 25.4.3, binary search:

template<class ForwardIterator, class T>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);
template<class ForwardIterator, class T>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
const T& value, Compare comp);

template<class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last,
const T& value);

template<class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last,
const T& value, Compare comp);

template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
const T& value);

template<class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last,
const T& value, Compare comp);

// 25.4.4, merge:
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
OutputIterator result);

template<class InputIterator1, class InputIterator2, class OutputIterator,
class Compare>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
OutputIterator result, Compare comp);

template<class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first,
BidirectionalIterator middle,
BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator first,
BidirectionalIterator middle,
BidirectionalIterator last, Compare comp);

// 25.4.5, set operations:
template<class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool includes(
InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, Compare comp);

template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
OutputIterator result);
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);

// 25.4.6, heap operations:
template<class RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);

template<class RandomAccessIterator>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
template<class RandomAccessIterator>
    void make_heap(RandomAccessIterator first, RandomAccessIterator last);
    template<class RandomAccessIterator, class Compare>
    void make_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

template<class RandomAccessIterator>
    void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
    template<class RandomAccessIterator, class Compare>
    void sort_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

template<class RandomAccessIterator>
    bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
    template<class RandomAccessIterator, class Compare>
    bool is_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

// 25.4.7, 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, class Compare>
pair<ForwardIterator, ForwardIterator>
  minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);

template<class InputIterator1, class InputIterator2>
bool lexicographical_compare(
  InputIterator1 first1, InputIterator1 last1,
  InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool lexicographical_compare(
  InputIterator1 first1, InputIterator1 last1,
  InputIterator2 first2, InputIterator2 last2, Compare comp);

// 25.4.9, permutations:

template<class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first, BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool next_permutation(BidirectionalIterator first, BidirectionalIterator last, Compare comp);

template<class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last, Compare comp);

3 All of the algorithms are separated from the particular implementations of data structures and are parameterized by iterator types. Because of this, they can work with program-defined data structures, as long as these data structures have iterator types satisfying the assumptions on the algorithms.

4 For purposes of determining the existence of data races, algorithms shall not modify objects referenced through an iterator argument unless the specification requires such modification.

5 Throughout this Clause, the names of template parameters are used to express type requirements. If an algorithm’s template parameter is InputIterator, InputIterator1, or InputIterator2, the actual template argument shall satisfy the requirements of an input iterator (24.2.3). If an algorithm’s template parameter is OutputIterator, OutputIterator1, or OutputIterator2, the actual template argument shall satisfy the requirements of an output iterator (24.2.4). If an algorithm’s template parameter is ForwardIterator, ForwardIterator1, or ForwardIterator2, the actual template argument shall satisfy the requirements of a forward iterator (24.2.5). If an algorithm’s template parameter is BidirectionalIterator, BidirectionalIterator1, or BidirectionalIterator2, the actual template argument shall satisfy the requirements of a bidirectional iterator (24.2.6). If an algorithm’s template parameter is RandomAccessIterator, RandomAccessIterator1, RandomAccessIterator2,
or `RandomAccessIterator2`, the actual template argument shall satisfy the requirements of a random-access iterator (24.2.7).

6 If an algorithm’s **Effects** section says that a value pointed to by any iterator passed as an argument is modified, then that algorithm has an additional type requirement: The type of that argument shall satisfy the requirements of a mutable iterator (24.2). [Note: this requirement does not affect arguments that are declared as `OutputIterator`, `OutputIterator1`, or `OutputIterator2`, because output iterators must always be mutable. — end note]

7 Both in-place and copying versions are provided for certain algorithms. When such a version is provided for `algorithm` it is called `algorithm_copy`. Algorithms that take predicates end with the suffix `_if` (which follows the suffix `_copy`).

8 The `Predicate` parameter is used whenever an algorithm expects a function object (20.8) that, when applied to the result of dereferencing the corresponding iterator, returns a value testable as `true`. In other words, if an algorithm takes `Predicate pred` as its argument and `first` as its iterator argument, it should work correctly in the construct `pred(*first)` contextually converted to `bool` (4). The function object `pred` shall not apply any non-constant function through the dereferenced iterator.

9 The `BinaryPredicate` parameter is used whenever an algorithm expects a function object that when applied to the result of dereferencing two corresponding iterators or to dereferencing an iterator and type `T` when `T` is part of the signature returns a value testable as `true`. In other words, if an algorithm takes `BinaryPredicate binary_pred` as its argument and `first1` and `first2` as its iterator arguments, it should work correctly in the construct `binary_pred(*first1, *first2)` contextually converted to `bool` (4). `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 construct `binary_pred(*first1, value)` contextually converted to `bool` (4). `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>` (20.8.4), or some equivalent solution. — end note]

11 When the description of an algorithm gives an expression such as `*first == value` for a condition, the expression shall evaluate to either true or false in boolean contexts.

12 In the description of the algorithms operators `+` and `-` are used for some of the iterator categories for which they do not have to be defined. In these cases the semantics of `a+n` is the same as that of

```cpp
X tmp = a;
advance(tmp, n);
return tmp;
```

and that of `b-a` is the same as of

```cpp
return distance(a, b);
```

---

25.2 Non-modifying sequence operations [alg.nonmodifying]

25.2.1 All of [alg.all_of]

273) The decision whether to include a copying version was usually based on complexity considerations. When the cost of doing the operation dominates the cost of copy, the copying version is not included. For example, `sort_copy` is not included because the cost of sorting is much more significant, and users might as well do `copy` followed by `sort`. 

§ 25.2.1
template <class InputIterator, class Predicate>
bool all_of(InputIterator first, InputIterator last, Predicate pred);


Returns: true if \([\text{first}, \text{last})\) is empty or if \(\text{pred}(*i)\) is true for every iterator \(i\) in the range \([\text{first}, \text{last})\), and false otherwise.

Complexity: At most \(\text{last} - \text{first}\) applications of the predicate.

25.2.2 Any of

template <class InputIterator, class Predicate>
bool any_of(InputIterator first, InputIterator last, Predicate pred);


Returns: false if \([\text{first}, \text{last})\) is empty or if there is no iterator \(i\) in the range \([\text{first}, \text{last})\) such that \(\text{pred}(*i)\) is true, and true otherwise.

Complexity: At most \(\text{last} - \text{first}\) applications of the predicate.

25.2.3 None of

template <class InputIterator, class Predicate>
bool none_of(InputIterator first, InputIterator last, Predicate pred);


Returns: true if \([\text{first}, \text{last})\) is empty or if \(\text{pred}(*i)\) is false for every iterator \(i\) in the range \([\text{first}, \text{last})\), and false otherwise.

Complexity: At most \(\text{last} - \text{first}\) applications of the predicate.

25.2.4 For each

template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);

Requires: Function shall meet the requirements of MoveConstructible (Table 34). [Note: Function need not meet the requirements of CopyConstructible (Table 35). — end note]

Effects: Applies \(f\) to the result of dereferencing every iterator in the range \([\text{first}, \text{last})\), starting from \(\text{first}\) and proceeding to \(\text{last} - 1\). [Note: If the type of \(\text{first}\) satisfies the requirements of a mutable iterator, \(f\) may apply nonconstant functions through the dereferenced iterator. — end note]

Returns: std::move(f).

Complexity: Applies \(f\) exactly \(\text{last} - \text{first}\) times.

Remarks: If \(f\) returns a result, the result is ignored.

25.2.5 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);

template<class InputIterator, class Predicate>
InputIterator find_if_not(InputIterator first, InputIterator last, Predicate pred);

**Returns:** The first iterator \( i \) in the range \([\text{first}, \text{last})\) for which the following corresponding conditions hold: \( *i == \text{value}, \text{pred}(*i) \neq \text{false} \), \( \text{pred}(*i) == \text{false} \). Returns \text{last} if no such iterator is found.

**Complexity:** At most \( \text{last} - \text{first} \) applications of the corresponding predicate.

### 25.2.6 Find End

[alg.find.end]

template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1
find_end(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
ForwardIterator1
find_end(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);

**Effects:** Finds a subsequence of equal values in a sequence.

**Returns:** The last iterator \( i \) in the range \([\text{first1}, \text{last1} - (\text{last2} - \text{first2}))\) such that for any non-negative integer \( n < (\text{last2} - \text{first2}) \), the following corresponding conditions hold: \( *(i + n) == *(\text{first2} + n) \), \( \text{pred}(*(i + n), *(\text{first2} + n)) \neq \text{false} \). Returns \text{last1} if \([\text{first2}, \text{last2})\) is empty or if no such iterator is found.

**Complexity:** At most \( (\text{last2} - \text{first2}) \times (\text{last1} - \text{first1} - (\text{last2} - \text{first2}) + 1) \) applications of the corresponding predicate.

### 25.2.7 Find First

[alg.find.first.of]

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 \([\text{first1}, \text{last1})\) such that for some iterator \( j \) in the range \([\text{first2}, \text{last2})\) the following conditions hold: \( *i == *j \), \( \text{pred}(*i, *j) \neq \text{false} \). Returns \text{last1} if \([\text{first2}, \text{last2})\) is empty or if no such iterator is found.

**Complexity:** At most \( (\text{last1}-\text{first1}) \times (\text{last2}-\text{first2}) \) applications of the corresponding predicate.
25.2.8 Adjacent find

template<class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last,
                              BinaryPredicate pred);

1 Returns: The first iterator i such that both i and i + 1 are in the range [first, last) for which the following corresponding conditions hold: *i == *(i + 1), pred(*i, *(i + 1)) != false. Returns last if no such iterator is found.

2 Complexity: For a nonempty range, exactly min((i - first) + 1, (last - first) - 1) applications of the corresponding predicate, where i is adjacent_find’s return value.

25.2.9 Count

template<class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
   count(InputIterator first, InputIterator last, const T& value);

template<class InputIterator, class Predicate>
typename iterator_traits<InputIterator>::difference_type
   count_if(InputIterator first, InputIterator last, Predicate pred);

1 Effects: Returns the number of iterators i in the range [first, last) for which the following corresponding conditions hold: *i == value, pred(*i) != false.

2 Complexity: Exactly last - first applications of the corresponding predicate.

25.2.10 Mismatch

template<class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
         InputIterator2 first2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
         InputIterator2 first2, BinaryPredicate pred);

1 Returns: A pair of iterators i and j such that j == first2 + (i - first1) and i is the first iterator in the range [first1, last1) for which the following corresponding conditions hold:

   !(i == *(first2 + (i - first1)))
   pred(*i, *(first2 + (i - first1))) == false

   Returns the pair last1 and first2 + (last1 - first1) if such an iterator i is not found.

2 Complexity: At most last1 - first1 applications of the corresponding predicate.
25.2.11 Equal [alg.equal]

template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2);

template<class InputIterator1, class InputIterator2,
    class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, BinaryPredicate pred);

1  Returns: true if for every iterator i in the range [first1, last1) the following corresponding conditions hold: *i == *(first2 + (i - first1)), pred(*i, *(first2 + (i - first1))) != false. Otherwise, returns false.

2  Complexity: At most last1 - first1 applications of the corresponding predicate.

25.2.12 Is permutation [alg.is_permutation]

template<class ForwardIterator1, class ForwardIterator2>
bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2);

template<class ForwardIterator1, class ForwardIterator2,
    class BinaryPredicate>
bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, BinaryPredicate pred);

1  Returns: true if there exists a permutation of the elements in the range [first2, first2 + (last1 - first1)), beginning with ForwardIterator2 begin, such that equal(first1, last1, begin) returns true or equal(first1, last1, begin, pred) returns true; otherwise, returns false.

2  Complexity: Exactly distance(first1, last1) applications of the corresponding predicate if equal(first1, last1, first2) would return true or equal(first1, last1, first2, pred) would return true; otherwise, at worst $O(N^2)$, where $N$ has the value distance(first1, last1).

25.2.13 Search [alg.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);

1  Effects: Finds a subsequence of equal values in a sequence.

2  Returns: The first iterator i in the range [first1, last1 - (last2 - first2)) such that for any non-negative integer n less than last2 - first2 the following corresponding conditions hold: *(i + n) == *(first2 + n), pred(*(i + n), *(first2 + n)) != false. Returns first1 if [first2, last2) is empty or if no such iterator is found.
Complexity: At most \((last1 - first1) \times (last2 - first2)\) applications of the corresponding predicate.

```cpp
template<class ForwardIterator, class Size, class T>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last, Size count,
            const T& value);
```

```cpp
template<class ForwardIterator, class Size, class T,
        class BinaryPredicate>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last, Size count,
            const T& value, BinaryPredicate pred);
```

**Requires:** The type `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 \([first, last - count)\) such that for any non-negative integer \(n\) less than \(count\) the following corresponding conditions hold: \(*(i + n) == value, pred(* (i + n), value) != false\). Returns `last` if no such iterator is found.

**Complexity:** At most \(last - first\) applications of the corresponding predicate.

### 25.3 Mutating sequence operations

#### 25.3.1 Copy

```cpp
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
            OutputIterator result);
```

**Effects:** Copies elements in the range \([first, last)\) into the range \([result, result + (last - first))\) starting from `first` and proceeding to `last`. For each non-negative integer \(n < (last - first)\), performs \(*(result + n) = *(first + n)\).

**Returns:** \(result + (last - first)\).

**Requires:** `result` shall not be in the range \([first, last)\).

**Complexity:** Exactly \(last - first\) assignments.

```cpp
template<class InputIterator, class Size, class OutputIterator>
OutputIterator copy_n(InputIterator first, Size n,
            OutputIterator result);
```

**Effects:** For each non-negative integer \(i < n\), performs \(*(result + i) = *(first + i)\).

**Returns:** \(result + n\).

**Complexity:** Exactly \(n\) assignments.

```cpp
template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator copy_if(InputIterator first, InputIterator last,
            OutputIterator result, Predicate pred);
```

**Requires:** The ranges \([first, last)\) and \([result, result + (last - first))\) shall not overlap.

§ 25.3.1
Effects: Copies all of the elements referred to by the iterator \( i \) in the range \([\text{first}, \text{last})\) for which \( \text{pred}(i) \) is true.

Complexity: Exactly \( \text{last} - \text{first} \) applications of the corresponding predicate.

Remarks: Stable.

```cpp
template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2
copy_backward(BidirectionalIterator1 first,
BidirectionalIterator1 last,
BidirectionalIterator2 result);
```

Effects: Copies elements in the range \([\text{first}, \text{last})\) into the range \([\text{result} - (\text{last} - \text{first}), \text{result})\) starting from \( \text{last} - 1 \) and proceeding to \( \text{first} \). For each positive integer \( n \leq (\text{last} - \text{first}) \), performs \( *(\text{result} - n) = *(\text{last} - n) \).

Requires: \( \text{result} \) shall not be in the range \([\text{first}, \text{last})\).

Returns: \( \text{result} - (\text{last} - \text{first}) \).

Complexity: Exactly \( \text{last} - \text{first} \) assignments.

### 25.3.2 Move

```cpp
template<class InputIterator, class OutputIterator>
OutputIterator move(InputIterator first, InputIterator last,
OutputIterator result);
```

Effects: Moves elements in the range \([\text{first}, \text{last})\) into the range \([\text{result}, \text{result} + (\text{last} - \text{first}))\) starting from \( \text{first} \) and proceeding to \( \text{last} \). For each non-negative integer \( n < (\text{last} - \text{first}) \), performs \( *(\text{result} + n) = \text{std::move}(*(\text{first} + n)) \).

Returns: \( \text{result} + (\text{last} - \text{first}) \).

Requires: \( \text{result} \) shall not be in the range \([\text{first}, \text{last})\).

Complexity: Exactly \( \text{last} - \text{first} \) move assignments.

```cpp
template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2
move_backward(BidirectionalIterator1 first,
BidirectionalIterator1 last,
BidirectionalIterator2 result);
```

Effects: Moves elements in the range \([\text{first}, \text{last})\) into the range \([\text{result} - (\text{last} - \text{first}), \text{result})\) starting from \( \text{last} - 1 \) and proceeding to \( \text{first} \). For each positive integer \( n \leq (\text{last} - \text{first}) \), performs \( *(\text{result} - n) = \text{std::move}(*(\text{last} - n)) \).

Requires: \( \text{result} \) shall not be in the range \([\text{first}, \text{last})\).

Returns: \( \text{result} - (\text{last} - \text{first}) \).

Complexity: Exactly \( \text{last} - \text{first} \) assignments.

---

274) \text{copy\_backward} should be used instead of \text{copy} when \( \text{last} \) is in the range \([\text{result} - (\text{last} - \text{first}), \text{result})\).

275) \text{move\_backward} should be used instead of \text{move} when \( \text{last} \) is in the range \([\text{result} - (\text{last} - \text{first}), \text{result})\).
25.3.3 swap

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2
swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
            ForwardIterator2 first2);
```

**Effects:** For each non-negative integer \( n < (last1 - first1) \) performs: 
\[
\text{swap}(\star(first1 + n), \star(first2 + n)).
\]

**Requires:** The two ranges \([\text{first1},last1)\) and \([\text{first2},\text{first2} + (last1 - first1))\) shall not overlap. 
\(\star(first1 + n)\) shall be swappable with (20.2.2) \(\star(first2 + n)\).

**Returns:** \(\text{first2} + (last1 - first1)\).

**Complexity:** Exactly \(last1 - first1\) swaps.

```
template<class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

**Effects:** \(\text{swap}(\star a, \star b)\).

**Requires:** \(a\) and \(b\) shall be dereferenceable. \(\star a\) shall be swappable with (20.2.2) \(\star b\).

25.3.4 Transform

```
template<class InputIterator, class OutputIterator,
         class UnaryOperation>
OutputIterator
transform(InputIterator first, InputIterator last,
          OutputIterator result, UnaryOperation op);
```

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class BinaryOperation>
OutputIterator
transform(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, OutputIterator result,
          BinaryOperation binary_op);
```

**Effects:** Assigns through every iterator \(i\) in the range \([\text{result},\text{result} + (last1 - first1))\) a new corresponding value equal to 
\[
op(\star(first1 + (i - \text{result}))) \text{ or } \text{binary_op}(\star(first1 + (i - \text{result}))), \star(first2 + (i - \text{result}))).
\]

**Requires:** \(\text{op}\) and \(\text{binary_op}\) shall not invalidate iterators or subranges, or modify elements in the ranges \([\text{first1},\text{last1}]\), \([\text{first2},\text{first2} + (last1 - first1))\), and \([\text{result},\text{result} + (last1 - first1))\].

**Returns:** \(\text{result} + (last1 - first1)\).

**Complexity:** Exactly \(last1 - first1\) 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.

---

\(^{276}\) The use of fully closed ranges is intentional.
25.3.5 Replace

\[ \text{alg.replace} \]

```cpp
template<class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
             const T& old_value, const T& new_value);
```

```cpp
template<class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last,
                Predicate pred, const T& new_value);
```

1. **Requires:** The expression \( *\text{first} = \text{new_value} \) shall be valid.

2. **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: \( *i == \text{old_value}, \text{pred}(i) != \text{false} \).

3. **Complexity:** Exactly \( \text{last} - \text{first} \) applications of the corresponding predicate.

```cpp
template<class InputIterator, class OutputIterator, class T>
OutputIterator
replace_copy(InputIterator first, InputIterator last,
             OutputIterator result,
             const T& old_value, const T& new_value);
```

```cpp
template<class InputIterator, class OutputIterator, class Predicate, class T>
OutputIterator
replace_copy_if(InputIterator first, InputIterator last,
                OutputIterator result,
                Predicate pred, const T& new_value);
```

4. **Requires:** The results of the expressions \( \text{first} \) and \text{new_value} shall be writable to the \text{result} output iterator. The ranges \([\text{first}, \text{last})\) and \([\text{result}, \text{result} + (\text{last} - \text{first}))\) shall not overlap.

5. **Effects:** Assigns to every iterator \( i \) in the range \([\text{result}, \text{result} + (\text{last} - \text{first}))\) either \text{new_value} or \( *(\text{first} + (i - \text{result})) \) depending on whether the following corresponding conditions hold:

   \[
   *\text{(first} + (i - \text{result})) == \text{old_value} \\
   \text{pred}(\text{*}(\text{first} + (i - \text{result}))) != \text{false}
   \]

6. **Returns:** \text{result} + (\text{last} - \text{first}).

7. **Complexity:** Exactly \( \text{last} - \text{first} \) applications of the corresponding predicate.

25.3.6 Fill

\[ \text{alg.fill} \]

```cpp
template<class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value);
```

```cpp
template<class OutputIterator, class Size, class T>
OutputIterator
fill_n(OutputIterator first, Size n, const T& value);
```

1. **Requires:** The expression \text{value} shall be writable to the output iterator. The type \text{Size} shall be convertible to an integral type (4.7, 12.3).

2. **Effects:** The first algorithm assigns \text{value} through all the iterators in the range \([\text{first}, \text{last})\). The second algorithm assigns \text{value} through all the iterators in the range \([\text{first}, \text{first} + \text{n})\) if \text{n} is positive, otherwise it does nothing.
3  Returns: `fill_n` returns `first + n` for non-negative values of `n` and `first` for negative values.
4  Complexity: Exactly `last - first`, `n`, or 0 assignments, respectively.

25.3.7 Generate

```cpp
template<class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last,
              Generator gen);
```

```cpp
template<class OutputIterator, class Size, class Generator>
OutputIterator generate_n(OutputIterator first, Size n, Generator gen);
```

1  Effects: The first algorithm invokes the function object `gen` and assigns the return value of `gen` through all the iterators in the range `[first, last)`. The second algorithm invokes the function object `gen` and assigns the return value of `gen` through all the iterators in the range `[first, first + n)` if `n` is positive, otherwise it does nothing.
2  Requires: `gen` takes no arguments, `Size` shall be convertible to an integral type (4.7, 12.3).
3  Returns: `generate_n` returns `first + n` for non-negative values of `n` and `first` for negative values.
4  Complexity: Exactly `last - first`, `n`, or 0 invocations of `gen` and assignments, respectively.

25.3.8 Remove

```cpp
template<class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                       const T& value);
```

```cpp
template<class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                          Predicate pred);
```

1  Requires: The type of `*first` shall satisfy the MoveAssignable requirements (Table 36).
2  Effects: Eliminates all the elements referred to by iterator `i` in the range `[first, last)` for which the following corresponding conditions hold: `*i == value, pred(*i) != false`.
3  Returns: The end of the resulting range.
4  Remarks: Stable.
5  Complexity: Exactly `last - first` applications of the corresponding predicate.

```cpp
template<class InputIterator, class OutputIterator, class T>
OutputIterator
remove_copy(InputIterator first, InputIterator last,
            OutputIterator result, const T& value);
```

```cpp
template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator
remove_copy_if(InputIterator first, InputIterator last,
               OutputIterator result, Predicate pred);
```

6  Requires: The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap. The expression `*result = *first` shall be valid.

§ 25.3.8
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.

### 25.3.9 Unique

[alg.unique]

```cpp
template<class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last);
```

```cpp
template<class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                      BinaryPredicate pred);
```

Effects: For a nonempty range, eliminates all but the first element from every consecutive group of equivalent elements referred to by the iterator \( i \) in the range \([\text{first} + 1, \text{last})\) for which the following conditions hold: \(*\text{(i - 1)} == *i \) or \(\text{pred}(*\text{(i - 1)}, *i) != \text{false}\.

Requires: The comparison function shall be an equivalence relation. The type of \( \ast\text{first} \) shall satisfy the MoveAssignable requirements (Table 36).

Returns: The end of the resulting range.

Complexity: For nonempty ranges, exactly \( (\text{last} - \text{first}) - 1 \) applications of the corresponding predicate.

```cpp
template<class InputIterator, class OutputIterator>
OutputIterator
unique_copy(InputIterator first, InputIterator last,
            OutputIterator result);
```

```cpp
template<class InputIterator, class OutputIterator,
         class BinaryPredicate>
OutputIterator
unique_copy(InputIterator first, InputIterator last,
            OutputIterator result, BinaryPredicate pred);
```

Requires: The comparison function shall be an equivalence relation. The ranges \([\text{first}, \text{last})\) and \([\text{result}, \text{result} + (\text{last} - \text{first}))\) shall not overlap. The expression \(\ast\text{result} = \ast\text{first}\) shall be valid. If neither \text{InputIterator} nor \text{OutputIterator} meets the requirements of forward iterator then the value type of \text{InputIterator} shall be CopyConstructible (35) and CopyAssignable (table 37). 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: \(*i == \ast\text{(i - 1)} \) or \(\text{pred}(*i, \ast\text{(i - 1)}) != \text{false}\.

Returns: The end of the resulting range.

Complexity: For nonempty ranges, exactly \( \text{last} - \text{first} - 1 \) applications of the corresponding predicate.
25.3.10 Reverse

```
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

1. **Effects:** For each non-negative integer \( i \leq (last - first)/2 \), applies \texttt{iter_swap} to all pairs of iterators \( first + i \), \( (last - i) - 1 \).

2. **Requires:** \*\texttt{first} shall be swappable (20.2.2).

3. **Complexity:** Exactly \( (last - first)/2 \) swaps.

```
template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
BidirectionalIterator last, OutputIterator result);
```

4. **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: \( *(result + (last - first) - i) = *(first + i) \).

5. **Requires:** The ranges \([first, last)\) and \([result, result+(last-first))\) shall not overlap.

6. **Returns:** \( result + (last - first) \).

7. **Complexity:** Exactly \( (last - first) \) assignments.

25.3.11 Rotate

```
template<class ForwardIterator>
ForwardIterator rotate(ForwardIterator first, ForwardIterator middle,
ForwardIterator last);
```

1. **Effects:** For each non-negative integer \( i < (last - first) \), places the element from the position \( first + i \) into position \( first + (i + (last - middle)) \% (last - first) \).

2. **Returns:** \( first + (last - middle) \).

3. **Remarks:** This is a left rotate.

4. **Requires:** \([first, middle)\) and \([middle, last)\) shall be valid ranges. \texttt{ForwardIterator} shall satisfy the requirements of \texttt{ValueSwappable} (20.2.2). The type of \*\texttt{first} shall satisfy the requirements of \texttt{MoveConstructible} (Table 34) and the requirements of \texttt{MoveAssignable} (Table 36).

5. **Complexity:** At most \( last - first \) swaps.

```
template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
ForwardIterator last, OutputIterator result);
```

6. **Effects:** Copies the range \([first, last)\) to the range \([result, result + (last - first))\) such that for each non-negative integer \( i < (last - first) \) the following assignment takes place: \( *(result + i) = *(first + (i + (middle - first)) \% (last - first)) \).

7. **Returns:** \( result + (last - first) \).

8. **Requires:** The ranges \([first, last)\) and \([result, result + (last - first))\) shall not overlap.

9. **Complexity:** Exactly \( last - first \) assignments.
25.3.12 Random shuffle

```cpp
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 shuffle(RandomAccessIterator first,
             RandomAccessIterator last,
             UniformRandomNumberGenerator& g);
```

1 Effects: Permutes the elements in the range \([\text{first},\text{last})\) such that each possible permutation of those elements has equal probability of appearance.

2 Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (20.2.2). The random number generating function object rand shall have a return type that is convertible to iterator_traits(RandomAccessIterator)::difference_type, and the call rand(n) shall return a randomly chosen value in the interval \([0,n)\), for \(n > 0\) of type iterator_traits(RandomAccessIterator)::difference_type. The type UniformRandomNumberGenerator shall meet the requirements of a uniform random number generator (26.5.1.3) type whose return type is convertible to iterator_traits(RandomAccessIterator)::difference_type.

3 Complexity: Exactly \((\text{last} - \text{first}) - 1\) swaps.

4 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 rand function from the standard C library.

   - In the second form of the function, the function object rand shall serve as the implementation’s source of randomness.

   - In the third shuffle form of the function, the object g shall serve as the implementation’s source of randomness.

25.3.13 Partitions

```cpp
template <class InputIterator, class Predicate>
bool is_partitioned(InputIterator first, InputIterator last, Predicate pred);
```

1 Requires: InputIterator’s value type shall be convertible to Predicate’s argument type.

2 Returns: true if \([\text{first},\text{last})\) is empty or if \([\text{first},\text{last})\) is partitioned by \text{pred}, i.e. if all elements that satisfy \text{pred} appear before those that do not.

3 Complexity: Linear. At most \((\text{last} - \text{first})\) applications of \text{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: \(\text{BidirectionalIterator}\) shall satisfy the requirements of \(\text{ValueSwappable}\) (20.2.2). The type of \(*\text{first}\) shall satisfy the requirements of \(\text{MoveConstructible}\) (Table 34) and of \(\text{MoveAssignable}\) (Table 36).

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.

\[\text{template <class InputIterator, class OutputIterator1, class OutputIterator2, class Predicate>}
\text{pair<OutputIterator1, OutputIterator2>}\]
\(\text{partition_copy(InputIterator first, InputIterator last, OutputIterator1 out_true, OutputIterator2 out_false, Predicate pred);}\)

Requires: \(\text{InputIterator}\)'s value type shall be Assignable, and shall be writable to the \(\text{out_true}\) and \(\text{out_false}\) OutputIterators, and shall be convertible to \(\text{Predicate}\)'s argument type. The input range shall not overlap with either of the output ranges.

Effects: For each iterator \(i\) in \([\text{first}, \text{last})\), copies \(*i\) to the output range beginning with \(\text{out_true}\) if \(\text{pred}(\ast i)\) is \(\text{true}\), or to the output range beginning with \(\text{out_false}\) otherwise.

Returns: A pair \(p\) such that \(p.\text{first}\) is the end of the output range beginning at \(\text{out_true}\) and \(p.\text{second}\) is the end of the output range beginning at \(\text{out_false}\).

Complexity: Exactly \(\text{last} - \text{first}\) applications of \(\text{pred}\).

\[\text{template<class ForwardIterator, class Predicate>}
\text{ForwardIterator partition_point(ForwardIterator first, ForwardIterator last, Predicate pred);}\)

Requires: \(\text{ForwardIterator}\)'s value type shall be convertible to \(\text{Predicate}\)'s argument type. \([\text{first}, \text{last})\) shall be partitioned by \(\text{pred}\), i.e. all elements that satisfy \(\text{pred}\) shall appear before those that do not.
Returns: An iterator mid such that all_of(first, mid, pred) and none_of(mid, last, pred) are both true.

Complexity: $O(\log(last - first))$ applications of pred.

25.4 Sorting and related operations

All the operations in 25.4 have two versions: one that takes a function object of type Compare and one that uses an operator<.

Compare is a function object type (20.8). The return value of the function call operation applied to an object of type Compare, when contextually converted to bool (4), yields true if the first argument of the call is less than the second, and false otherwise. Compare comp is used throughout for algorithms assuming an ordering relation. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.

For all algorithms that take Compare, there is a version that uses operator< instead. That is, comp(*i, *j) != false defaults to *i < *j != false. For algorithms other than those described in 25.4.3 to work correctly, comp has to induce a strict weak ordering on the values.

The term strict refers to the requirement of an irreflexive relation (!comp(x, x) for all x), and the term weak to requirements that are not as strong as those for a total ordering, but stronger than those for a partial ordering. If we define equiv(a, b) as !comp(a, b) && !comp(b, a), then the requirements are that comp and equiv both be transitive relations:

- comp(a, b) && comp(b, c) implies comp(a, c)
- equiv(a, b) && equiv(b, c) implies equiv(a, c) [Note: Under these conditions, it can be shown that
  - equiv is an equivalence relation
  - comp induces a well-defined relation on the equivalence classes determined by equiv
  - The induced relation is a strict total ordering. — end note]

A sequence is sorted with respect to a comparator comp if for any iterator i pointing to the sequence and any non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, comp(*i + n), *i) == false.

A sequence [start,finish) is partitioned with respect to an expression f(e) if there exists an integer n such that for all 0 <= i < distance(start, finish), f(*(start + i)) is true if and only if i < n.

In the descriptions of the functions that deal with ordering relationships we frequently use a notion of equivalence to describe concepts such as stability. The equivalence to which we refer is not necessarily an operator==, but an equivalence relation induced by the strict weak ordering. That is, two elements a and b are considered equivalent if and only if !a < b) && !b < a).

25.4.1 Sorting

25.4.1.1 sort

```cpp
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

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: RandomAccessIterator shall satisfy the requirements of ValueSwappable (20.2.2). The type of \(*\text{first}\) shall satisfy the requirements of MoveConstructible (Table 34) and of MoveAssignable (Table 36).

Complexity: \(\Theta(N \log(N))\) (where \(N = \text{last} - \text{first}\)) comparisons.

25.4.1.2 stable_sort

```
template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Effects: Sorts the elements in the range \([\text{first}, \text{last})\).

Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (20.2.2). The type of \(*\text{first}\) shall satisfy the requirements of MoveConstructible (Table 34) and of MoveAssignable (Table 36).

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.

25.4.1.3 partial_sort

```
template<class RandomAccessIterator>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last, Compare comp);
```

Effects: Places the first \(\text{middle} - \text{first}\) sorted elements from the range \([\text{first}, \text{last})\) into the range \([\text{first}, \text{middle})\). The rest of the elements in the range \([\text{middle}, \text{last})\) are placed in an unspecified order.

Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (20.2.2). The type of \(*\text{first}\) shall satisfy the requirements of MoveConstructible (Table 34) and of MoveAssignable (Table 36).

Complexity: It takes approximately \((\text{last} - \text{first}) \times \log(\text{middle} - \text{first})\) comparisons.

25.4.1.4 partial_sort_copy

```
§ 25.4.1.4
```
template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
RandomAccessIterator result_first,
RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator,
class Compare>
RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
RandomAccessIterator result_first,
RandomAccessIterator result_last,
Compare comp);

1 Effects: Places the first min(last - first, result_last - result_first) sorted elements into the
range [result_first, result_first + min(last - first, result_last - result_first)).
2 Returns: The smaller of: result_last or result_first + (last - first).
3 Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (20.2.2). The type
of *result_first shall satisfy the requirements of MoveConstructible (Table 34) and of MoveAssignable
(Table 36).
4 Complexity: Approximately (last - first) * log(min(last - first, result_last - result_first)) comparisons.

25.4.1.5 is_sorted

template<class ForwardIterator>
bool is_sorted(ForwardIterator first, ForwardIterator last);
1 Returns: is_sorted_until(first, last) == last

template<class ForwardIterator, class Compare>
bool is_sorted(ForwardIterator first, ForwardIterator last,
Compare comp);
2 Returns: is_sorted_until(first, last, comp) == last

template<class ForwardIterator>
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last,
Compare comp);
3 Returns: If distance(first, last) < 2, returns last. Otherwise, returns the last iterator i in
[first,last] for which the range [first,i) is sorted.
4 Complexity: Linear.

25.4.2 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: RandomAccessIterator shall satisfy the requirements of ValueSwappable (20.2.2). The type 
of *first shall satisfy the requirements of MoveConstructible (Table 34) and of MoveAssignable 
(Table 36).

Complexity: Linear on average.

25.4.3 Binary search

All of the algorithms in this section are versions of binary search and assume that the sequence being 
searched is partitioned with respect to an expression formed by binding the search key to an argument of 
the implied or explicit comparison function. They work on non-random access iterators minimizing the 
number of comparisons, which will be logarithmic for all types of iterators. They are especially appropriate 
for random access iterators, because these algorithms do a logarithmic number of steps through the data 
structure. For non-random access iterators they execute a linear number of steps.

25.4.3.1 lower_bound

template<class ForwardIterator, class T> 
ForwardIterator 
    lower_bound(ForwardIterator first, ForwardIterator last, 
                const T& value);

template<class ForwardIterator, class T, class Compare> 
ForwardIterator 
    lower_bound(ForwardIterator first, ForwardIterator last, 
                const T& value, Compare comp);

Requires: The elements e of [first,last) 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 \(\log_2(last - first) + \Theta(1)\) comparisons.

25.4.3.2 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 \([\text{first}, \text{last})\) shall be partitioned with respect to the expression \((\text{value} < e) \text{ or } !\text{comp(value, e)}\).

Returns: The furthermost iterator i in the range \([\text{first}, \text{last}]\) such that for any iterator j in the range \([\text{first}, i)\) the following corresponding conditions hold: \((\text{value} < *j) \text{ or } \text{comp(value, *j)} == \text{false}\).

Complexity: At most \(\log_2(\text{last} - \text{first}) + O(1)\) comparisons.

25.4.3.3 equal_range

\[
\text{template<class ForwardIterator, class T>} \\
\quad \text{pair<ForwardIterator, ForwardIterator>} \\
\quad \quad \text{equal_range(ForwardIterator first,} \\
\quad \quad \quad \quad \quad \text{ForwardIterator last, const T& value);} \\
\text{template<class ForwardIterator, class T, class Compare>} \\
\quad \text{pair<ForwardIterator, ForwardIterator>} \\
\quad \quad \text{equal_range(ForwardIterator first,} \\
\quad \quad \quad \quad \quad \text{ForwardIterator last, const T& value,} \\
\quad \quad \quad \quad \quad \text{Compare comp);} \\
\]

Requires: The elements e of \([\text{first}, \text{last})\) shall be partitioned with respect to the expressions e < value and !\((\text{value} < e) \text{ or } \text{comp(e, value)} \text{ and } !\text{comp(value, e)}\). Also, for all elements e of \([\text{first}, \text{last}), e < \text{value}\) shall imply \((\text{value} < e) \text{ or } \text{comp(e, value)} \text{ shall imply } !\text{comp(value, e)}\).

Returns:

\[
\text{make_pair(lower_bound(first, last, value),} \\
\quad \quad \quad \quad \quad \text{upper_bound(first, last, value))} \\
\quad \text{or} \\
\quad \quad \quad \quad \quad \text{make_pair(lower_bound(first, last, value, comp),} \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{upper_bound(first, last, value, comp))} \\
\]

Complexity: At most \(2 * \log_2(\text{last} - \text{first}) + O(1)\) comparisons.

25.4.3.4 binary_search

\[
\text{template<class ForwardIterator, class T>} \\
\quad \text{bool binary_search(ForwardIterator first, ForwardIterator last,} \\
\quad \quad \quad \quad \quad \text{const T& value);} \\
\text{template<class ForwardIterator, class T, class Compare>} \\
\quad \text{bool binary_search(ForwardIterator first, ForwardIterator last,} \\
\quad \quad \quad \quad \quad \text{const T& value, Compare comp);} \\
\]

Requires: The elements e of \([\text{first}, \text{last})\) are partitioned with respect to the expressions e < value and \((\text{value} < e) \text{ or } \text{comp(e, value)} \text{ and } !\text{comp(value, e)}\). Also, for all elements e of \([\text{first}, \text{last}), e < \text{value}\) implies \((\text{value} < e) \text{ or } \text{comp(e, value)} \text{ implies } !\text{comp(value, e)}\).

Returns: true if there is an iterator i in the range \([\text{first}, \text{last})\) that satisfies the corresponding conditions: \(!(*i < \text{value}) && !\text{comp(value, *i)} \quad \text{or} \quad \text{comp(*i, value)} == \text{false} && \text{comp(value, *i)} == \text{false}\).
Complexity: At most \( \log_2(\text{last} - \text{first}) + O(1) \) comparisons.

### 25.4.4 Merge

```cpp
template<class InputIterator1, class InputIterator2,
        class OutputIterator>
OutputIterator
merge(InputIterator1 first1, InputIterator1 last1,
      InputIterator2 first2, InputIterator2 last2,
      OutputIterator result);
```

```cpp
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator
merge(InputIterator1 first1, InputIterator1 last1,
      InputIterator2 first2, InputIterator2 last2,
      OutputIterator result, Compare comp);
```

Effects: Copies all the elements of the two ranges \([\text{first1}, \text{last1})\) and \([\text{first2}, \text{last2})\) into the range \([\text{result}, \text{result}_\text{last})\), where \(\text{result}_\text{last} = \text{result} + (\text{last1} - \text{first1}) + (\text{last2} - \text{first2})\), such that the resulting range satisfies \(\text{is_sorted} (\text{result}, \text{result}_\text{last})\) or \(\text{is_sorted} (\text{result}, \text{result}_\text{last}, \text{comp})\), respectively.

Requires: The ranges \([\text{first1}, \text{last1})\) and \([\text{first2}, \text{last2})\) shall be sorted with respect to \(\text{operator<}\) or \(\text{comp}\). The resulting range shall not overlap with either of the original ranges.

Returns: \(\text{result} + (\text{last1} - \text{first1}) + (\text{last2} - \text{first2})\).

Complexity: At most \((\text{last1} - \text{first1}) + (\text{last2} - \text{first2}) - 1\) comparisons.

Remarks: Stable.

```cpp
template<class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first,
                   BidirectionalIterator middle,
                   BidirectionalIterator last);
```

```cpp
template<class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator first,
                   BidirectionalIterator middle,
                   BidirectionalIterator last, Compare comp);
```

Effects: Merges two sorted consecutive ranges \([\text{first}, \text{middle})\) and \([\text{middle}, \text{last})\), putting the result of the merge into the range \([\text{first}, \text{last})\). The resulting range will be in non-decreasing order; that is, for every iterator \(i\) in \([\text{first}, \text{last})\) other than \(\text{first}\), the condition \(*i < *(i - 1)\) or, respectively, \(\text{comp}(*i, *(i - 1))\) will be false.

Requires: The ranges \([\text{first}, \text{middle})\) and \([\text{middle}, \text{last})\) shall be sorted with respect to \(\text{operator<}\) or \(\text{comp}\). \(\text{BidirectionalIterator}\) shall satisfy the requirements of \(\text{ValueSwappable}\) (20.2.2). The type of \(*\text{first}\) shall satisfy the requirements of \(\text{MoveConstructible}\) (Table 34) and of \(\text{MoveAssignable}\) (Table 36).

Complexity: When enough additional memory is available, \((\text{last} - \text{first}) - 1\) comparisons. If no additional memory is available, an algorithm with complexity \(N \log(N)\) (where \(N\) is equal to \(\text{last} - \text{first}\)) may be used.

Remarks: Stable.
25.4.5 Set operations on sorted structures

This section defines all the basic set operations on sorted structures. They also work with multisets (23.4.4) 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.

25.4.5.1 includes

```template<class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2);
```

Returns: true if [first2, last2) is empty or if every element in the range [first2, last2) is contained in the range [first1, last1). Returns false otherwise.

Complexity: At most 2 * ((last1 - first1) + (last2 - first2)) - 1 comparisons.

25.4.5.2 set_union

```template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2,
                          OutputIterator result);
```

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 * ((last1 - first1) + (last2 - first2)) - 1 comparisons.

Remarks: If [first1, last1) contains m elements that are equivalent to each other and [first2, last2) contains n elements that are equivalent to them, then all m elements from the first range shall be copied to the output range, in order, and then \(\max(n - m, 0)\) elements from the second range shall be copied to the output range, in order.

25.4.5.3 set_intersection

§ 25.4.5.3
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);

Effects: Constructs a sorted intersection of the elements from the two ranges; that is, the set of elements
that are present in both of the ranges.

Requires: The resulting range shall not overlap with either of the original ranges.

Returns: The end of the constructed range.

Complexity: At most $2 \times ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1$ comparisons.

Remarks: If $[\text{first1},\text{last1})$ contains $m$ elements that are equivalent to each other and $[\text{first2}, \text{last2})$ contains $n$ elements that are equivalent to them, the first $\min(m,n)$ elements shall be copied
from the first range to the output range, in order.

25.4.5.4 set_difference

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);

Effects: Copies the elements of the range $[\text{first1},\text{last1})$ which are not present in the range $[\text{first2}, \text{last2})$ to the range beginning at result. The elements in the constructed range are sorted.

Requires: The resulting range shall not overlap with either of the original ranges.

Returns: The end of the constructed range.

Complexity: At most $2 \times ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1$ comparisons.

Remarks: If $[\text{first1},\text{last1})$ contains $m$ elements that are equivalent to each other and $[\text{first2}, \text{last2})$ contains $n$ elements that are equivalent to them, the last $\max(m-n,0)$ elements from $[\text{first1}, \text{last1})$ shall be copied to the output range.
25.4.5.5 set_symmetric_difference

```cpp
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                                       InputIterator2 first2, InputIterator2 last2,
                                       OutputIterator result);
```

```cpp
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
OutputIterator set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                                       InputIterator2 first2, InputIterator2 last2,
                                       OutputIterator result, Compare comp);
```

1. **Effects:** Copies the elements of the range \([first1, last1)\) 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.

2. **Requires:** The resulting range shall not overlap with either of the original ranges.

3. **Returns:** The end of the constructed range.

4. **Complexity:** At most \(2 \times ((last1 - first1) + (last2 - first2)) - 1\) comparisons.

5. **Remarks:** If \([first1, last1)\) contains \(m\) elements that are equivalent to each other and \([first2, last2)\) contains \(n\) elements that are equivalent to them, then \(|m - n|\) 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 \(m < n\).

25.4.6 Heap operations

1. 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.

2. These properties make heaps useful as priority queues.

3. `make_heap()` converts a range into a heap and `sort_heap()` turns a heap into a sorted sequence.

25.4.6.1 push_heap

```cpp
template<class RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

1. **Effects:** Places the value in the location `last - 1` into the resulting heap \([first, last)\).

2. **Requires:** The range \([first, last - 1)\) shall be a valid heap. The type of \(*first\) shall satisfy the MoveConstructible requirements (Table 34) and the the MoveAssignable requirements (Table 36).
25.4.6.2 pop_heap

```cpp
template<class RandomAccessIterator>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);
```

1 **Requires:** The range \([\textit{first}, \textit{last})\) shall be a valid non-empty heap. \texttt{RandomAccessIterator} shall satisfy the requirements of \texttt{ValueSwappable} (20.2.2). The type of \*\texttt{first} shall satisfy the requirements of \texttt{MoveConstructible} (Table 34) and of \texttt{MoveAssignable} (Table 36).

2 **Effects:** Swaps the value in the location \texttt{first} with the value in the location \(\texttt{last} - 1\) and makes \([\texttt{first}, \texttt{last} - 1)\) into a heap.

3 **Complexity:** At most \(2 \times \log(\texttt{last} - \texttt{first})\) comparisons.

25.4.6.3 make_heap

```cpp
template<class RandomAccessIterator>
void make_heap(RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
void make_heap(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);
```

1 **Effects:** Constructs a heap out of the range \([\texttt{first}, \texttt{last})\).

2 **Requires:** The type of \*\texttt{first} shall satisfy the \texttt{MoveConstructible} requirements (Table 34) and the \texttt{MoveAssignable} requirements (Table 36).

3 **Complexity:** At most \(3 \times (\texttt{last} - \texttt{first})\) comparisons.

25.4.6.4 sort_heap

```cpp
template<class RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);
```

1 **Effects:** Sorts elements in the heap \([\texttt{first}, \texttt{last})\).

2 **Requires:** The range \([\texttt{first}, \texttt{last})\) shall be a valid heap. \texttt{RandomAccessIterator} shall satisfy the requirements of \texttt{ValueSwappable} (20.2.2). The type of \*\texttt{first} shall satisfy the requirements of \texttt{MoveConstructible} (Table 34) and of \texttt{MoveAssignable} (Table 36).

3 **Complexity:** At most \(N \log(N)\) comparisons (where \(N == \texttt{last} - \texttt{first}\)).
25.4.6.5  is_heap

```cpp
template<class RandomAccessIterator>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
```

Returns: is_heap_until(first, last) == last

```cpp
template<class RandomAccessIterator, class Compare>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Returns: is_heap_until(first, last, comp) == last

```cpp
template<class RandomAccessIterator>
RandomAccessIterator is_heap_until(RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
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.

25.4.7  Minimum and maximum

```cpp
template<class T> const T& min(const T& a, const T& b);
```

```cpp
template<class T, class Compare>
const T& min(const T& a, const T& b, Compare comp);
```

Requires: Type \(T\) is LessThanComparable (32).

Returns: The smaller value.

Remarks: Returns the first argument when the arguments are equivalent.

```cpp
template<class T> const T& min(const T& a, const T& b, const T& c);
```

```cpp
template<class T, class... Args>
const T& min(const T& a, const T& b, Compare comp);
```

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.

```cpp
template<class T, class U, class... Args>
const T& min(const T& a, const U& b, Compare comp);
```

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);
```

```cpp
template<class T, class Compare>
```

§ 25.4.7
const T& max(const T& a, const T& b, Compare comp);

Requires: Type T is LessThanComparable (32).

Returns: The larger value.

Remarks: Returns the first argument when the arguments are equivalent.

template<class T> 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.

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.

template<class T> pair<const T&, const T&> minmax(const T& a, const T& b);
template<class T, class Compare>
pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);

Requires: Type T shall be LessThanComparable (32).

Returns: pair<const T&, const T&>(b, a) if b is smaller than a, and pair<const T&, const T&>(a, b) otherwise.

Remarks: Returns pair<const T&, const T&>(a, b) when the arguments are equivalent.

Complexity: Exactly one comparison.

template<class 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);

Requires: T is LessThanComparable, and all types forming Args... are the same as T.

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.

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.

Complexity: At most \((3/2)\) sizeof...(Args) applications of the corresponding predicate.

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:** \( \text{pair<const } T\&, \text{ const } T\&>(x, y) \) 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 \( \text{pair<const } T\&, \text{ const } T\&>(a, a) \) if sizeof...(Args) is 0.

30 **Complexity:** At most \( (3/2) \) sizeof...(Args) applications of the corresponding predicate.

31 template<class ForwardIterator>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last);

32 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: \((*j < *i) \) or \( \text{comp}(*j, *i) == \text{false} \). Returns last if \( first == last \).

34 **Complexity:** Exactly \( \text{max}((last - first) - 1, 0) \) applications of the corresponding comparisons.

35 template<class ForwardIterator>
pair<ForwardIterator, ForwardIterator>
minmax_element(ForwardIterator first, ForwardIterator last);

36 template<class ForwardIterator, class Compare>
pair<ForwardIterator, ForwardIterator>
minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);

37 **Returns:** \( \text{make_pair(first, first)} \) if \([first,last)\) is empty, otherwise \( \text{make_pair(m, M)} \), where \( m \) is the first iterator in \([first,last)\) such that no iterator in the range refers to a smaller element, and where \( M \) is the last iterator in \([first,last)\) such that no iterator in the range refers to a larger element.

38 **Complexity:** At most \( \text{max}([1/2(N - 1)], 0) \) applications of the corresponding predicate, where \( N \) is distance(first, last).

25.4.8 Lexicographical comparison

§ 25.4.8
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) and false otherwise.

Complexity: At most 2*min((last1 - first1), (last2 - first2)) applications of the correspond-
ing 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;

Remarks: An empty sequence is lexicographically less than any non-empty sequence, but not less than
any empty sequence.

25.4.9 Permutation generators

template<class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last, Compare comp);

Effects: Takes a sequence defined by the range [first,last) and transforms it into the next permu-
tation. The next permutation is found by assuming that the set of all permutations is lexicographically
sorted with respect to operator< or comp. If such a permutation exists, it returns true. Otherwise, it
transforms the sequence into the smallest permutation, that is, the ascendingly sorted one, and returns
false.

Requires: BidirectionalIterator shall satisfy the requirements of ValueSwappable (20.2.2).

Complexity: At most (last - first)/2 swaps.

template<class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first,
BidirectionalIterator last, Compare comp);

Effects: Takes a sequence defined by the range \([\text{first}, \text{last})\) 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 \text{operator<} or \text{comp}.

Returns: \text{true} if such a permutation exists. Otherwise, it transforms the sequence into the largest permutation, that is, the descendingly sorted one, and returns \text{false}.

Requires: BidirectionalIterator shall satisfy the requirements of ValueSwappable (20.2.2).

Complexity: At most \((\text{last} - \text{first})/2\) swaps.

25.5 C library algorithms

Table 110 describes some of the contents of the header \text{<cstdlib>}.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>\text{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 \text{<stdlib.h>} with the following exceptions:

The function signature:

\[
\text{bsearch(const void *, const void *, size_t, size_t, int (*)(const void *, const void *))};
\]

is replaced by the two declarations:

\[
\text{extern "C" void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));}
\]
\[
\text{extern "C++" void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));}
\]

both of which have the same behavior as the original declaration.

The function signature:

\[
\text{qsort(void *, size_t, size_t, int (*)(const void *, const void *))};
\]

is replaced by the two declarations:

\[
\text{extern "C" void qsort(void* base, size_t nmemb, size_t size, int (*compar)(const void*, const void*));}
\]
\[
\text{extern "C++" void qsort(void* base, size_t nmemb, size_t size, int (*compar)(const void*, const void*));}
\]

both of which have the same behavior as the original declaration. The behavior is undefined unless the objects in the array pointed to by \text{base} are of trivial type.

\[\text{Note: Because the function argument compar() may throw an exception, bsearch() and qsort() are allowed to propagate the exception (17.6.4.11). — end note}\]
See also: ISO C 7.10.5.
26 Numerics library

26.1 General

This Clause describes components that C++ programs may use to perform seminumerical operations.

The following subclauses describe components for complex number types, random number generation, numeric (n-at-a-time) arrays, generalized numeric algorithms, and facilities included from the ISO C library, as summarized in Table 111.

Table 111 — Numerics library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>26.2</td>
<td></td>
</tr>
<tr>
<td>26.4</td>
<td>&lt;complex&gt;</td>
</tr>
<tr>
<td>26.5</td>
<td>&lt;random&gt;</td>
</tr>
<tr>
<td>26.6</td>
<td>&lt;valarray&gt;</td>
</tr>
<tr>
<td>26.7</td>
<td>&lt;numeric&gt;</td>
</tr>
<tr>
<td>26.8</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>

26.2 Numeric type requirements

The complex and valarray components are parameterized by the type of information they contain and manipulate. A C++ program shall instantiate these components only with a type T that satisfies the following requirements:

— T is not an abstract class (it has no pure virtual member functions);
— T is not a reference type;
— T is not cv-qualified;
— If T is a class, it has a public default constructor;
— If T is a class, it has a public copy constructor with the signature T::T(const T&);
— If T is a class, it has a public destructor;
— If T is a class, it has a public assignment operator whose signature is either T& T::operator=(const T&) or T& T::operator=(T)
— 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.

277) In other words, value types. These include arithmetic types, pointers, the library class complex, and instantiations of valarray for value types.
Destruction of an object, followed by initialization of its raw storage using the copy constructor, is semantically equivalent to assignment to the original object.

[Note: This rule states that there shall not be any subtle differences in the semantics of initialization versus assignment. This gives an implementation considerable flexibility in how arrays are initialized.]

[Example: An implementation is allowed to initialize a valarray by allocating storage using the new operator (which implies a call to the default constructor for each element) and then assigning each element its value. Or the implementation can allocate raw storage and use the copy constructor to initialize each element. —end example]

If the distinction between initialization and assignment is important for a class, or if it fails to satisfy any of the other conditions listed above, the programmer should use vector (23.3.6) instead of valarray for that class; —end note]

— If T is a class, it does not overload unary operator&.

2 If any operation on T throws an exception the effects are undefined.

3 In addition, many member and related functions of valarray<T> can be successfully instantiated and will exhibit well-defined behavior if and only if T satisfies additional requirements specified for each such member or related function.

4 [Example: It is valid to instantiate valarray<complex>, but operator()> will not be successfully instantiated for valarray<complex> operands, since complex does not have any ordering operators. —end example]

26.3 The floating-point environment [cfenv]

26.3.1 Header <cfenv> synopsis [cfenv.syn]

namespace std {

    // types
    typedef object type fenv_t;
    typedef integer type fexcept_t;

    // functions
    int feclearexcept(int except);
    int fegetexceptflag(fexcept_t *pflag, int except);
    int feraiseexcept(int except);
    int fesetexceptflag(const fexcept_t *pflag, int except);
    int fetestexcept(int except);
    int fegetround(void);
    int fesetround(int mode);

    int fegetenv(fenv_t *penv);
    int feholdexcept(fenv_t *penv);
    int fesetenv(const fenv_t *penv);
    int feupdateenv(const fenv_t *penv);

}

1 The header also defines the macros:

    FE_ALL_EXCEPT
    FE_DIVBYZERO
    FE_INEXACT

§ 26.3.1 885
2 The header defines all functions, types, and macros the same as Clause 7.6 of the C standard.

3 The floating-point environment has thread storage duration (3.7.2). The initial state for a thread’s floating-point environment is the state of the floating-point environment of the thread that constructs the corresponding std::thread object (30.3.1) at the time it constructed the object. [Note: that is, the child thread gets the floating-point state of the parent thread at the time of the child’s creation. — end note]

4 A separate floating-point environment shall be maintained for each thread. Each function accesses the environment corresponding to its calling thread.

26.4 Complex numbers [complex.numbers]

1 The header <complex> defines a class template, and numerous functions for representing and manipulating complex numbers.

2 The effect of instantiating the template complex for any type other than float, double, or long double is unspecified. The specializations complex<float>, complex<double>, and complex<long double> are literal types (3.9).

3 If the result of a function is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

4 If z is an lvalue expression of type cv std::complex<T> then:
   — the expression reinterpret_cast<cv T(&)[2]>(z) shall be well-formed,
   — reinterpret_cast<cv T(&)[2]>(z)[0] shall designate the real part of z, and
   — reinterpret_cast<cv T(&)[2]>(z)[1] shall designate the imaginary part of z.

Moreover, if a is an expression of type cv std::complex<T>∗ and the expression a[i] is well-defined for an integer expression i, then:
   — reinterpret_cast<cv T*>(a)[2*i] shall designate the real part of a[i], and
   — reinterpret_cast<cv T*>(a)[2*i + 1] shall designate the imaginary part of a[i].

26.4.1 Header <complex> synopsis [complex.syn]

namespace std {
   template<class T> class complex;
   template<> class complex<float>;
   template<> class complex<double>;
   template<> class complex<long double>;

   // 26.4.6 operators:
   template<class T>
template<class T> complex<T> operator+(const complex<T>&, const complex<T>&);
template<class T> complex<T> operator+(const complex<T>&, const T&);
template<class T> complex<T> operator+(const T&, const complex<T>&);

template<class T> complex<T> operator-(
    const complex<T>&, const complex<T>&);
template<class T> complex<T> operator-(const complex<T>&, const T&);
template<class T> complex<T> operator-(const T&, const complex<T>&);

template<class T> complex<T> operator*(
    const complex<T>&, const complex<T>&);
template<class T> complex<T> operator*(const complex<T>&, const T&);
template<class T> complex<T> operator*(const T&, const complex<T>&);

template<class T> complex<T> operator/(
    const complex<T>&, 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>&);

// 26.4.7 values:
template<class T, class charT, class traits>
    basic_istream<charT, traits>&
        operator>>(basic_istream<charT, traits>&, complex<T>&);

template<class T, class charT, class traits>
    basic_ostream<charT, traits>&
        operator<< (basic_ostream<charT, traits>&, const complex<T>&);

// 26.4.8 transcendental:
template<class T> complex<T> acos(const complex<T>&);
template<class T> complex<T> asin(const complex<T>&);
template<class T> complex<T> atan(const complex<T>&);
template<class T> complex<T> acosh(const complex<T>&);
template<class T> complex<T> asinh(const complex<T>&);
template<class T> complex<T> atanh(const complex<T>&);

template<class T> complex<T> cos (const complex<T>&);
template<class T> complex<T> cosh (const complex<T>&);
template<class T> complex<T> exp  (const complex<T>&);
template<class T> complex<T> log  (const complex<T>&);
template<class T> complex<T> log10(const complex<T>&);

template<class T> complex<T> pow(const complex<T>&, const T&);
template<class T> complex<T> pow(const complex<T>&, const complex<T>&);
template<class T> complex<T> pow(const T&, const complex<T>&);

template<class T> complex<T> sin (const complex<T>&);
template<class T> complex<T> sinh (const complex<T>&);
template<class T> complex<T> sqrt (const complex<T>&);
template<class T> complex<T> tan  (const complex<T>&);
template<class T> complex<T> tanh (const complex<T>&);

26.4.2 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;
            void real(T);
            T imag() const;
            void imag(T);

            complex<T>& operator= (const T&);
            complex<T>& operator+=(const T&);
            complex<T>& operator-=(const T&);
            complex<T>& operator*=(const T&);
            complex<T>& operator/=(const T&);

            complex& operator=(const complex&);
            template<class X> complex<T>& operator= (const complex<X>&);
            template<class X> complex<T>& operator+=(const complex<X>&);
            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>&);
    }
}

§ 26.4.2
The class `complex` describes an object that can store the Cartesian components, `real()` and `imag()`, of a complex number.

### 26.4.3 complex specializations

```cpp
namespace std {
    template<> class complex<float> {
        public:
            typedef float value_type;

            constexpr complex(float re = 0.0f, float im = 0.0f);
            explicit constexpr complex(const complex<double>&);
            explicit constexpr complex(const complex<long double>&);

            constexpr float real();
            void real(float);
            constexpr float imag();
            void imag(float);

            complex<float>& operator= (float);
            complex<float>& operator+=(float);
            complex<float>& operator-=(float);
            complex<float>& operator*=(float);
            complex<float>& operator/=(float);

            complex<float>& operator=(const complex<float>&);
            template<class X> complex<float>& operator= (const complex<X>&);
            template<class X> complex<float>& operator+=(const complex<X>&);
            template<class X> complex<float>& operator-=(const complex<X>&);
            template<class X> complex<float>& operator*=(const complex<X>&);
            template<class X> complex<float>& operator/=(const complex<X>&);
    }

    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();
            void real(double);
            constexpr double imag();
            void imag(double);

            complex<double>& operator= (double);
            complex<double>& operator+=(double);
            complex<double>& operator-=(double);
            complex<double>& operator*=(double);
            complex<double>& operator/=(double);

            complex<double>& operator=(const complex<double>&);
            template<class X> complex<double>& operator= (const complex<X>&);
            template<class X> complex<double>& operator+=(const complex<X>&);
            template<class X> complex<double>& operator-=(const complex<X>&);
            template<class X> complex<double>& operator*=(const complex<X>&);
            template<class X> complex<double>& operator/=(const complex<X>&);
}
```
template<class X> complex<double>& operator*=(const complex<X>&);
template<class X> complex<double>& operator/=(const complex<X>&);

};

template<> class complex<long double> {
public:
typedef long double value_type;

conestructor complex(long double re = 0.0L, long double im = 0.0L);
conestructor complex(const complex<float>&);
conestructor complex(const complex<double>&);

conestructor long double real();
void real(long double);
conestructor long double imag();
void imag(long double);

complex<long double>& operator=(const complex<long double>&);
complex<long double>& operator=(long double);
complex<long double>& operator+=(long double);
complex<long double>& operator-=(long double);
complex<long double>& operator*=(long double);
complex<long double>& operator/=(long double);

template<class X> complex<long double>& operator= (const complex<X>&);
template<class X> complex<long double>& operator+=(const complex<X>&);
template<class X> complex<long double>& operator-=(const complex<X>&);
template<class X> complex<long double>& operator*=(const complex<X>&);
template<class X> complex<long double>& operator/=(const complex<X>&);
};

26.4.4 complex member functions

template<class T> complex(const T& re = T(), const T& im = T());

Effects: Constructs an object of class complex.

Postcondition: real() == re & im() == im.

T real() const;
Returns: the value of the real component.

void real(T val);
Effects: Assigns val to the real component.

T imag() const;
Returns: the value of the imaginary component.

void imag(T val);
Effects: Assigns val to the imaginary component.

26.4.5 complex member operators
complex<T>& operator+=(const T& rhs);

Effects: Adds the scalar value rhs to the real part of the complex value *this and stores the result in the real part of *this, leaving the imaginary part unchanged.

Returns: *this.

complex<T>& operator-=(const T& rhs);

Effects: Subtracts the scalar value rhs from the real part of the complex value *this and stores the result in the real part of *this, leaving the imaginary part unchanged.

Returns: *this.

complex<T>& operator*=(const T& rhs);

Effects: Multiplies the scalar value rhs by the complex value *this and stores the result in *this.

Returns: *this.

complex<T>& operator/=(const T& rhs);

Effects: Divides the scalar value rhs into the complex value *this and stores the result in *this.

Returns: *this.

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.

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.

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.

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.

26.4.6 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);
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: \( \text{complex}\langle T\rangle(\text{lhs}) \pm \text{rhs} \).

template<class T> complex<T> operator+(const complex<T>& lhs, const complex<T>& rhs);

Returns: \( \text{complex}\langle T\rangle(-\text{lhs}\text{.real()},-\text{lhs}\text{.imag()}) \).

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: \( \text{complex}\langle T\rangle(\text{lhs}) \pm \text{rhs} \).

template<class T> complex<T> operator*(const complex<T>& lhs, const complex<T>& rhs);

Returns: \( \text{complex}\langle T\rangle(\text{lhs}) \pm \text{rhs} \).

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: \( \text{complex}\langle T\rangle(\text{lhs}) \pm \text{rhs} \).

template<class T> complex<T> operator/(const complex<T>& lhs, const complex<T>& rhs);

Returns: \( \text{complex}\langle T\rangle(\text{lhs}) \pm \text{rhs} \).

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: \( \text{lhs}\text{.real()} = \text{rhs}\text{.real()} \&\& \text{lhs}\text{.imag()} = \text{rhs}\text{.imag()} \).

Remarks: The imaginary part is assumed to be \( \text{T}() \), or 0.0, for the \( \text{T} \) arguments.

template<class T> bool operator==(const complex<T>& lhs, const complex<T>& rhs);

Returns: \( \text{lhs}\text{.real()} = \text{rhs}\text{.real()} \&\& \text{lhs}\text{.imag()} = \text{rhs}\text{.imag()} \).

Remarks: \( \text{The imaginary part is assumed to be T() , or 0.0, for the T arguments.} \)

template<class T> bool operator!=(const complex<T>& lhs, const complex<T>& rhs);

Returns: \( \text{rhs}\text{.real()} \neq \text{lhs}\text{.real()} \) || \( \text{rhs}\text{.imag()} \neq \text{lhs}\text{.imag()} \).

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 \( (27.7.1.2) \).

Requires: The input values shall be convertible to \( T \).

If bad input is encountered, calls \( \text{is.setstate(ios_base::failbit)} \) (which may throw \( \text{ios::failure} \) \( (27.5.4.3) \).

Returns: \( \text{is} \).

Remarks: This extraction is performed as a series of simpler extractions. Therefore, the skipping of
whitespace is specified to be the same for each of the simpler extractions.

§ 26.4.6


```cpp

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();
}
```

**Note:** In a locale in which comma is used as a decimal point character, the use of comma as a field separator can be ambiguous. Inserting `std::ios_base::showpoint` into the output stream forces all outputs to show an explicit decimal point character; as a result, all inserted sequences of complex numbers can be extracted unambiguously.

### 26.4.7 complex value operations

```

template<class T> T real(const complex<T>& x);
```

**Returns:** \(x.\text{real}()\).

```

template<class T> T imag(const complex<T>& x);
```

**Returns:** \(x.\text{imag}()\).

```

template<class T> T abs(const complex<T>& x);
```

**Returns:** the magnitude of \(x\).

```

template<class T> T arg(const complex<T>& x);
```

**Returns:** the phase angle of \(x\), or \(\text{atan2}(\text{imag}(x), \text{real}(x))\).

```

template<class T> T norm(const complex<T>& x);
```

**Returns:** the squared magnitude of \(x\).

```

template<class T> complex<T> conj(const complex<T>& x);
```

**Returns:** the complex conjugate of \(x\).

```

template<class T> complex<T> proj(const complex<T>& x);
```

**Effects:** Behaves the same as the C function \(\text{cproj}\), defined in 7.3.9.4.

```

template<class T> complex<T> polar(const T& rho, const T& theta = 0);
```

**Returns:** the complex value corresponding to a complex number whose magnitude is \(\text{rho}\) and whose phase angle is \(\text{theta}\).
26.4.8 complex transcendentals

template<class T> complex<T> acos(const complex<T>& x);
  Effects: Behaves the same as C function cacos, defined in 7.3.5.1.

template<class T> complex<T> asin(const complex<T>& x);
  Effects: Behaves the same as C function casin, defined in 7.3.5.2.

template<class T> complex<T> atan(const complex<T>& x);
  Effects: Behaves the same as C function catan, defined in 7.3.5.3.

template<class T> complex<T> acosh(const complex<T>& x);
  Effects: Behaves the same as C function cacosh, defined in 7.3.6.1.

template<class T> complex<T> asinh(const complex<T>& x);
  Effects: Behaves the same as C function casinh, defined in 7.3.6.2.

template<class T> complex<T> atanh(const complex<T>& x);
  Effects: Behaves the same as C function catanh, defined in 7.3.6.3.

template<class T> complex<T> cos(const complex<T>& x);
  Returns: the complex cosine of x.

template<class T> complex<T> cosh(const complex<T>& x);
  Returns: the complex hyperbolic cosine of x.

template<class T> complex<T> exp(const complex<T>& x);
  Returns: the complex base e exponential of x.

template<class T> complex<T> log(const complex<T>& x);
  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, const complex<T>& y);

template<class T> complex<T> pow (const complex<T>& x, const T& y);
template<class T> complex<T> pow (const T& x, const complex<T>& y);
  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*log(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 \( x \).

```cpp
template<class T> complex<T> sinh (const complex<T>& x);
```

Returns: the complex hyperbolic sine of \( x \).

```cpp
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.

```cpp
template<class T> complex<T> tan (const complex<T>& x);
```

Returns: the complex tangent of \( x \).

```cpp
template<class T> complex<T> tanh (const complex<T>& x);
```

26.4.9 Additional Overloads

The following function templates shall have additional overloads:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arg</code></td>
<td></td>
</tr>
<tr>
<td><code>conj</code></td>
<td></td>
</tr>
<tr>
<td><code>imag</code></td>
<td></td>
</tr>
<tr>
<td><code>norm</code></td>
<td></td>
</tr>
<tr>
<td><code>proj</code></td>
<td></td>
</tr>
<tr>
<td><code>real</code></td>
<td></td>
</tr>
</tbody>
</table>

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>`.

All the specified overloads shall have a return type which is the nested `value_type` of the effectively cast arguments.

Function template `pow` shall have additional overloads sufficient to ensure, for a call with at least one argument of type `complex<T>`:

1. If either argument has type `complex<long double>` or type `long double`, then both arguments are effectively cast to `complex<long double>`.
2. Otherwise, if either argument has type `complex<double>`, `double`, or an integer type, then both arguments are effectively cast to `complex<double>`.
3. Otherwise, if either argument has type `complex<float>` or `float`, then both arguments are effectively cast to `complex<float>`.

26.4.10 Header `<ccomplex>`

The header behaves as if it simply includes the header `<complex>`.

26.5 Random number generation

This subclause defines a facility for generating (pseudo-)random numbers.
In addition to a few utilities, four categories of entities are described: uniform random number generators, random number engines, random number engine adaptors, and random number distributions. These categorizations are applicable to types that satisfy the corresponding requirements, to objects instantiated from such types, and to templates producing such types when instantiated. [Note: These entities are specified in such a way as to permit the binding of any uniform random number generator object e as the argument to any random number distribution object d, thus producing a zero-argument function object such as given by \texttt{bind(d,e)}. — end note]

Each of the entities specified via this subclause has an associated arithmetic type (3.9.1) identified as \texttt{result_type}. With \( T \) as the \texttt{result_type} thus associated with such an entity, that entity is characterized:

a) as \texttt{boolean} or equivalently as \texttt{boolean-valued}, if \( T \) is \texttt{bool};

b) otherwise as \texttt{integral} or equivalently as \texttt{integer-valued}, if \texttt{numeric_limits\:<T>::is_integer} is true;

c) otherwise as \texttt{floating} or equivalently as \texttt{real-valued}.

If integer-valued, an entity may optionally be further characterized as \texttt{signed} or \texttt{unsigned}, according to \texttt{numeric_limits\:<T>::is_signed}.

Unless otherwise specified, all descriptions of calculations in this subclause use mathematical real numbers.

Throughout this subclause, the operators \texttt{bitand}, \texttt{bitor}, and \texttt{xor} denote the respective conventional bitwise operations. Further:

a) the operator \texttt{rshift} denotes a bitwise right shift with zero-valued bits appearing in the high bits of the result, and

b) the operator \texttt{lshift\:w} denotes a bitwise left shift with zero-valued bits appearing in the low bits of the result, and whose result is always taken modulo \( 2^w \).

### 26.5.1 Requirements [rand.req]

#### 26.5.1.1 General requirements [rand.req.genl]

Throughout this subclause 26.5, the effect of instantiating a template:

a) that has a template type parameter named \texttt{Sseq} is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of seed sequence (26.5.1.2).

b) that has a template type parameter named \texttt{URNG} is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of uniform random number generator (26.5.1.3).

c) that has a template type parameter named \texttt{Engine} is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of random number engine (26.5.1.4).

d) that has a template type parameter named \texttt{RealType} is undefined unless the corresponding template argument is cv-unqualified and is one of \texttt{float}, \texttt{double}, or \texttt{long double}.

e) that has a template type parameter named \texttt{IntType} is undefined unless the corresponding template argument is cv-unqualified and is one of \texttt{short}, \texttt{int}, \texttt{long}, \texttt{long long}, \texttt{unsigned short}, \texttt{unsigned int}, \texttt{unsigned long}, or \texttt{unsigned long long}.

f) that has a template type parameter named \texttt{UIntType} is undefined unless the corresponding template argument is cv-unqualified and is one of \texttt{unsigned short}, \texttt{unsigned int}, \texttt{unsigned long}, or \texttt{unsigned long long}.

§ 26.5.1.1 896
Throughout this subclause 26.5, phrases of the form “x is an iterator of a specific kind” shall be interpreted as equivalent to the more formal requirement that “x is a value of a type satisfying the requirements of the specified iterator type.”

Throughout this subclause 26.5, any constructor that can be called with a single argument and that satisfies a requirement specified in this subclause shall be declared explicit.

26.5.1.2 Seed sequence requirements

A seed sequence is an object that consumes a sequence of integer-valued data and produces a requested number of unsigned integer values \(i\), \(0 \leq i < 2^{32}\), based on the consumed data. [Note: Such an object provides a mechanism to avoid replication of streams of random variates. This can be useful, for example, in applications requiring large numbers of random number engines. — end note]

A class \(S\) satisfies the requirements of a seed sequence if the expressions shown in Table 112 are valid and have the indicated semantics, and if \(S\) also satisfies all other requirements of this section 26.5.1.2. In that Table and throughout this section:

- a) \(T\) is the type named by \(S\)’s associated \result{type};
- b) \(q\) is a value of \(S\) and \(r\) is a possibly const value of \(S\);
- c) \(ib\) and \(ie\) are input iterators with an unsigned integer \value{type} of at least 32 bits;
- d) \(rb\) and \(re\) are mutable random access iterators with an unsigned integer \value{type} of at least 32 bits;
- e) \(ob\) is an output iterator; and
- f) \(il\) is a value of \initializer{list}\(\langle T\rangle\).

Table 112 — Seed sequence requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S::\result{type})</td>
<td>(T)</td>
<td>(T) is an unsigned integer type ((3.9.1)) of at least 32 bits.</td>
<td>compile-time</td>
</tr>
<tr>
<td>(S())</td>
<td>—</td>
<td>Creates a seed sequence with the same initial state as all other default-constructed seed sequences of type (S).</td>
<td>constant</td>
</tr>
<tr>
<td>(S(ib,ie))</td>
<td>—</td>
<td>Creates a seed sequence having internal state that depends on some or all of the bits of the supplied sequence ([ib,ie)).</td>
<td>(O(ie - ib))</td>
</tr>
<tr>
<td>(S(il))</td>
<td>—</td>
<td>Same as (S(il.begin(), il.end())).</td>
<td>same as (S(il.begin(), il.end()))</td>
</tr>
</tbody>
</table>

§ 26.5.1.2 897
Table 112 — Seed sequence requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>q.generate(rb, re)</td>
<td>void</td>
<td>Does nothing if rb == re. Otherwise, fills the supplied sequence [rb, re) with 32-bit quantities that depend on the sequence supplied to the constructor and possibly also depend on the history of generate’s previous invocations.</td>
<td>$O(re - rb)$</td>
</tr>
<tr>
<td>r.size()</td>
<td>size_t</td>
<td>The number of 32-bit units that would be copied by a call to r.param.</td>
<td>constant</td>
</tr>
<tr>
<td>r.param(ob)</td>
<td>void</td>
<td>Copies to the given destination a sequence of 32-bit units that can be provided to the constructor of a second object of type S, and that would reproduce in that second object a state indistinguishable from the state of the first object.</td>
<td>$O(r.size())$</td>
</tr>
</tbody>
</table>

26.5.1.3 Uniform random number generator requirements

1 A uniform random number generator $g$ of type $G$ is a function object returning unsigned integral values such that each value in the range of possible results has (ideally) equal probability of being returned. [Note: The degree to which $g$’s results approximate the ideal is often determined statistically. — end note]

2 A class $G$ satisfies the requirements of a uniform random number generator if the expressions shown in Table 113 are valid and have the indicated semantics, and if $G$ also satisfies all other requirements of this section 26.5.1.3. In that Table and throughout this section:

a) $T$ is the type named by $G$’s associated result_type, and

b) $g$ is a value of $G$.

Table 113 — Uniform random number generator requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>G::result_type</td>
<td>$T$</td>
<td>$T$ is an unsigned integer type (3.9.1).</td>
<td>compile-time</td>
</tr>
<tr>
<td>$g()$</td>
<td>$T$</td>
<td>Returns a value in the closed interval [$G:\min()$, $G:\max()$].</td>
<td>amortized constant</td>
</tr>
<tr>
<td>G::min()</td>
<td>$T$</td>
<td>Denotes the least value potentially returned by operator().</td>
<td>compile-time</td>
</tr>
<tr>
<td>G::max()</td>
<td>$T$</td>
<td>Denotes the greatest value potentially returned by operator().</td>
<td>compile-time</td>
</tr>
</tbody>
</table>

§ 26.5.1.3
The following relation shall hold: \( G::\text{min}() < G::\text{max}() \).

26.5.1.4 Random number engine requirements

A random number engine (commonly shortened to engine) \( e \) of type \( E \) is a uniform random number generator that additionally meets the requirements (e.g., for seeding and for input/output) specified in this section.

At any given time, \( e \) has a state \( e_i \) for some integer \( i \geq 0 \). Upon construction, \( e \) has an initial state \( e_0 \). An engine’s state may be established via a constructor, a seed function, assignment, or a suitable operator\( \gg \).

E’s specification shall define:

a) the size of \( E \)’s state in multiples of the size of result_type, given as an integral constant expression;

b) the transition algorithm \( TA \) by which \( e \)’s state \( e_i \) is advanced to its successor state \( e_{i+1} \); and

c) the generation algorithm \( GA \) by which an engine’s state is mapped to a value of type result_type.

A class \( E \) that satisfies the requirements of a uniform random number generator (26.5.1.3) also satisfies the requirements of a random number engine if the expressions shown in Table 114 are valid and have the indicated semantics, and if \( E \) also satisfies all other requirements of this section 26.5.1.4. In that Table and throughout this section:

a) \( T \) is the type named by \( E \)’s associated result_type;

b) \( e \) is a value of \( E \), \( v \) is an lvalue of \( E \), \( x \) and \( y \) are (possibly const) values of \( E \);

c) \( s \) is a value of \( T \);

d) \( q \) is an lvalue satisfying the requirements of a seed sequence (26.5.1.2);

e) \( z \) is a value of type 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 Clause 21 and Clause 27.

Table 114 — Random number engine requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E() )</td>
<td>—</td>
<td>Creates an engine with the same initial state as all other default-constructed engines of type ( E ).</td>
<td>( \mathcal{O}(\text{size of state}) )</td>
</tr>
<tr>
<td>( E(x) )</td>
<td>—</td>
<td>Creates an engine that compares equal to ( x ).</td>
<td>( \mathcal{O}(\text{size of state}) )</td>
</tr>
<tr>
<td>( E(s) )</td>
<td>—</td>
<td>Creates an engine with initial state determined by ( s ).</td>
<td>( \mathcal{O}(\text{size of state}) )</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>E(q)\textsuperscript{278}</td>
<td>—</td>
<td>Creates an engine with an initial state that depends on a sequence produced by one call to q.generate.</td>
<td>same as complexity of q.generate called on a sequence whose length is size of state</td>
</tr>
<tr>
<td>e.seed()</td>
<td>void</td>
<td>post: e == E().</td>
<td>same as E()</td>
</tr>
<tr>
<td>e.seed(s)</td>
<td>void</td>
<td>post: e == E(s).</td>
<td>same as E(s)</td>
</tr>
<tr>
<td>e.seed(q)</td>
<td>void</td>
<td>post: e == E(q).</td>
<td>same as E(q)</td>
</tr>
<tr>
<td>e()</td>
<td>T</td>
<td>Advances e’s state (e_i) to (e_{i+1} = TA(e_i)) and returns (GA(e_i)).</td>
<td>per Table 113</td>
</tr>
<tr>
<td>e.discard(z)\textsuperscript{279}</td>
<td>void</td>
<td>Advances e’s state (e_i) to (e_{i+z}) by any means equivalent to (z) consecutive calls e().</td>
<td>no worse than the complexity of (z) consecutive calls e()</td>
</tr>
<tr>
<td>x == y</td>
<td>bool</td>
<td>This operator is an equivalence relation. With (S_x) and (S_y) as the infinite sequences of values that would be generated by repeated future calls to x() and y(), respectively, returns \textit{true} if (S_x = S_y); else returns \textit{false}.</td>
<td>(O(\text{size of state}))</td>
</tr>
<tr>
<td>x != y</td>
<td>bool</td>
<td>!(x == y).</td>
<td>(O(\text{size of state}))</td>
</tr>
<tr>
<td>os &lt;&lt; x</td>
<td>reference to the type of os</td>
<td>With os.fmtflags set to ios_base::dec</td>
<td>ios_base::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>
</tr>
</tbody>
</table>

\textsuperscript{278} This constructor (as well as the subsequent corresponding seed() function) may be particularly useful to applications requiring a large number of independent random sequences.

\textsuperscript{279} This operation is common in user code, and can often be implemented in an engine-specific manner so as to provide significant performance improvements over an equivalent naive loop that makes \(z\) consecutive calls e().

§ 26.5.1.4

900
<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>is &gt;&gt; v</td>
<td>reference to the type of is</td>
<td>With <code>is.fmtflags</code> set to <code>ios_base::dec</code>, 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 <code>is.setstate(ios::failbit)</code> (which may throw <code>ios::failure [27.5.4.3]</code>). If a textual representation written via <code>os &lt;&lt; x</code> was subsequently read via <code>is &gt;&gt; v</code>, then <code>x == v</code> provided that there have been no intervening invocations of x or of v.</td>
<td>( \mathcal{O}(\text{size of state}) )</td>
</tr>
</tbody>
</table>

5. E shall meet the requirements of CopyConstructible (Table 35) and CopyAssignable (Table 37) types. These operations shall each be of complexity no worse than \( \mathcal{O}(\text{size of state}) \).

26.5.1.5 Random number engine adaptor requirements

1. A random number engine adaptor (commonly shortened to adaptor) \( A \) of type \( A \) is a random number engine that takes values produced by some other random number engine, and applies an algorithm to those values in order to deliver a sequence of values with different randomness properties. An engine \( B \) of type \( B \) adapted in this way is termed a base engine in this context.

2. The requirements of a random number engine type shall be interpreted as follows with respect to a random number engine adaptor type.

\[
A::A();
\]

3. **Effects**: The base engine is initialized as if by its default constructor.

\[
\text{bool operator==}(\text{const } A& \text{ a1, const } A& \text{ a2});
\]

4. **Returns**: `true` if \( a1 \)’s base engine is equal to \( a2 \)’s base engine. Otherwise returns `false`. 

§ 26.5.1.5
A::A(result_type s);

Effect: The base engine is initialized with s.

template<class Sseq> void A::A(Sseq& q);

Effect: The base engine is initialized with q.

void seed();

Effect: With b as the base engine, invokes b.seed().

void seed(result_type s);

Effect: With b as the base engine, invokes b.seed(s).

template<class Sseq> void seed(Sseq& q);

Effect: With b as the base engine, invokes b.seed(q).

A shall also satisfy the following additional requirements:

a) The complexity of each function shall not exceed the complexity of the corresponding function applied to the base engine.

b) The state of A shall include the state of its base engine. The size of A’s state shall be no less than the size of the base engine.

c) Copying A’s state (e.g., during copy construction or copy assignment) shall include copying the state of the base engine of A.

d) The textual representation of A shall include the textual representation of its base engine.

26.5.1.6 Random number distribution requirements

A random number distribution (commonly shortened to distribution) d of type D is a function object returning values that are distributed according to an associated mathematical probability density function \( p(z) \) or according to an associated discrete probability function \( P(z_i) \). A distribution’s specification identifies its associated probability function \( p(z) \) or \( P(z_i) \).

An associated probability function is typically expressed using certain externally-supplied quantities known as the parameters of the distribution. Such distribution parameters are identified in this context by writing, for example, \( p(z \mid a, b) \) or \( P(z_i \mid a, b) \), to name specific parameters, or by writing, for example, \( p(z \mid \{p\}) \) or \( P(z_i \mid \{p\}) \), to denote a distribution’s parameters \( p \) taken as a whole.

A class D satisfies the requirements of a random number distribution if the expressions shown in Table 115 are valid and have the indicated semantics, and if D and its associated types also satisfy all other requirements of this section 26.5.1.6. In that Table and throughout this section,

a) T is the type named by D’s associated result_type;

b) P is the type named by D’s associated param_type;

c) d is a value of D, and x and y are (possibly const) values of D.
d) $\text{glb}$ and $\text{lub}$ are values of $T$ respectively corresponding to the greatest lower bound and the least upper bound on the values potentially returned by $d$’s $\text{operator()}$, as determined by the current values of $d$’s parameters;

e) $p$ is a (possibly $\text{const}$) value of $P$;

f) $g$, $g1$, and $g2$ are $l$values of a type satisfying the requirements of a uniform random number generator [26.5.1.3];

g) $os$ is an $l$value of the type of some class template specialization $\text{basic_ostream}<\text{charT}, \text{traits}>$; and

h) $is$ is an $l$value of the type of some class template specialization $\text{basic_istream}<\text{charT}, \text{traits}>$;

where $\text{charT}$ and $\text{traits}$ are constrained according to Clause 21 and 27.

Table 115 — Random number distribution requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D::\text{result_type}$</td>
<td>$\text{T}$</td>
<td>$\text{T}$ is an arithmetic type (3.9.1).</td>
<td>compile-time</td>
</tr>
<tr>
<td>$D::\text{param_type}$</td>
<td>$\text{P}$</td>
<td>—</td>
<td>compile-time</td>
</tr>
<tr>
<td>$D()$</td>
<td>—</td>
<td>Creates a distribution whose behavior is indistinguishable from that of any other newly default-constructed distribution of type $D$.</td>
<td>constant</td>
</tr>
<tr>
<td>$D(p)$</td>
<td>—</td>
<td>Creates a distribution whose behavior is indistinguishable from that of a distribution newly constructed directly from the values used to construct $p$.</td>
<td>same as $p$’s construction</td>
</tr>
<tr>
<td>$d.\text{reset}()$</td>
<td>$\text{void}$</td>
<td>Subsequent uses of $d$ do not depend on values produced by any engine prior to invoking $\text{reset}$.</td>
<td>constant</td>
</tr>
<tr>
<td>$x.\text{param}()$</td>
<td>$\text{P}$</td>
<td>Returns a value $p$ such that $D(p).\text{param}() == p$.</td>
<td>no worse than the complexity of $D(p)$</td>
</tr>
<tr>
<td>$d.\text{param}(p)$</td>
<td>$\text{void}$</td>
<td>post: $d.\text{param}() == p$.</td>
<td>no worse than the complexity of $D(p)$</td>
</tr>
<tr>
<td>$d(g)$</td>
<td>$\text{T}$</td>
<td>With $p = d.\text{param}()$, the sequence of numbers returned by successive invocations with the same object $g$ is randomly distributed according to the associated $p(z</td>
<td>{p})$ or $P(z_i</td>
</tr>
</tbody>
</table>

§ 26.5.1.6
<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(d(g,p))</td>
<td>(T)</td>
<td>The sequence of numbers returned by successive invocations with the same objects (g) and (p) is randomly distributed according to the associated (p(z \mid {p})) or (P(z_i \mid {p})) function.</td>
<td>amortized constant number of invocations of (g)</td>
</tr>
<tr>
<td>(x.\text{min}())</td>
<td>(T)</td>
<td>Returns (\text{glb}).</td>
<td>constant</td>
</tr>
<tr>
<td>(x.\text{max}())</td>
<td>(T)</td>
<td>Returns (\text{lub}).</td>
<td>constant</td>
</tr>
<tr>
<td>(x == y)</td>
<td>(\text{bool})</td>
<td>This operator is an equivalence relation. Returns (\text{true}) if (x.\text{param}() == y.\text{param}()) and (S_1 = S_2), where (S_1) and (S_2) are the infinite sequences of values that would be generated, respectively, by repeated future calls to (x(g_1)) and (y(g_2)) whenever (g_1 == g_2). Otherwise returns (\text{false}).</td>
<td>constant</td>
</tr>
<tr>
<td>(x != y)</td>
<td>(\text{bool})</td>
<td>(! (x == y)).</td>
<td>same as (x == y).</td>
</tr>
<tr>
<td>(\text{os} \ll x)</td>
<td>reference to the type of (\text{os})</td>
<td>Writes to (\text{os}) a textual representation for the parameters and the additional internal data of (x).</td>
<td>—</td>
</tr>
<tr>
<td>(\text{is} \gg d)</td>
<td>reference to the type of (\text{is})</td>
<td>Restores from (\text{is}) the parameters and additional internal data of the lvalue (d). If bad input is encountered, ensures that (d) is unchanged by the operation and calls (\text{is}.\text{setstate}((\text{ios}::\text{failbit})) (which may throw (\text{ios}::\text{failure})) [27.5.4.3]).</td>
<td>—</td>
</tr>
</tbody>
</table>

4 \(d\) shall satisfy the requirements of CopyConstructible (Table 35) and CopyAssignable (Table 37) types.

5 The sequence of numbers produced by repeated invocations of \(d(g)\) shall be independent of any invocation.
of \texttt{os} \texttt{\ll} \texttt{d} or of any \texttt{const} member function of \texttt{D} between any of the invocations \texttt{d(g)}.

6 If a textual representation is written using \texttt{os} \texttt{\ll} \texttt{x} and that representation is restored into the same or a different object \texttt{y} of the same type using \texttt{is} \texttt{\gg} \texttt{y}, repeated invocations of \texttt{y(g)} shall produce the same sequence of numbers as would repeated invocations of \texttt{x(g)}.

7 It is unspecified whether \texttt{D::param_type} is declared as a (nested) \texttt{class} or via a \texttt{typedef}. In this subclause \texttt{26.5}, declarations of \texttt{D::param_type} are in the form of \texttt{typedef}s for convenience of exposition only.

8 \texttt{P} shall satisfy the requirements of \texttt{CopyConstructible} (Table 35), \texttt{CopyAssignable} (Table 37), and \texttt{Equality-Comparable} (Table 31) types.

9 For each of the constructors of \texttt{D} taking arguments corresponding to parameters of the distribution, \texttt{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 \texttt{D} that return values corresponding to parameters of the distribution, \texttt{P} shall have a corresponding member function with the identical name, type, and semantics.

10 \texttt{P} shall have a declaration of the form

\begin{verbatim}
typedef D distribution_type;
\end{verbatim}

\section{26.5.2 Header <random> synopsis \texttt{[rand.synopsis]}}

\begin{verbatim}
namespace std {
    #include <initializer_list>

    // 26.5.3.1 Class template linear_congruential_engine
    template<class UIntType, UIntType a, UIntType c, UIntType m>
    class linear_congruential_engine;

    // 26.5.3.2 Class template mersenne_twister_engine
    template<class UIntType, size_t w, size_t n, size_t m, size_t r,
             UIntType a, size_t u, UIntType d, size_t s,
             UIntType b, size_t t,
             UIntType c, size_t l, UIntType f>
    class mersenne_twister_engine;

    // 26.5.3.3 Class template subtract_with_carry_engine
    template<class UIntType, size_t w, size_t s, size_t r>
    class subtract_with_carry_engine;

    // 26.5.4.1 Class template discard_block_engine
    template<class Engine, size_t p, size_t r>
    class discard_block_engine;

    // 26.5.4.2 Class template independent_bits_engine
    template<class Engine, size_t w, class UIntType>
    class independent_bits_engine;

    // 26.5.4.3 Class template shuffle_order_engine
    template<class Engine, size_t k>
    class shuffle_order_engine;

    // 26.5.5 Engines and engine adaptors with predefined parameters
typedef see below minstd_rand0;
\end{verbatim}

\section{26.5.2 Header <random> synopsis \texttt{[rand.synopsis]}}

\begin{verbatim}
namespace std {
    #include <initializer_list>

    // 26.5.3.1 Class template linear_congruential_engine
    template<class UIntType, UIntType a, UIntType c, UIntType m>
    class linear_congruential_engine;

    // 26.5.3.2 Class template mersenne_twister_engine
    template<class UIntType, size_t w, size_t n, size_t m, size_t r,
             UIntType a, size_t u, UIntType d, size_t s,
             UIntType b, size_t t,
             UIntType c, size_t l, UIntType f>
    class mersenne_twister_engine;

    // 26.5.3.3 Class template subtract_with_carry_engine
    template<class UIntType, size_t w, size_t s, size_t r>
    class subtract_with_carry_engine;

    // 26.5.4.1 Class template discard_block_engine
    template<class Engine, size_t p, size_t r>
    class discard_block_engine;

    // 26.5.4.2 Class template independent_bits_engine
    template<class Engine, size_t w, class UIntType>
    class independent_bits_engine;

    // 26.5.4.3 Class template shuffle_order_engine
    template<class Engine, size_t k>
    class shuffle_order_engine;

    // 26.5.5 Engines and engine adaptors with predefined parameters
typedef see below minstd_rand0;
\end{verbatim}

\section{26.5.2 Header <random> synopsis \texttt{[rand.synopsis]}}

\begin{verbatim}
namespace std {
    #include <initializer_list>

    // 26.5.3.1 Class template linear_congruential_engine
    template<class UIntType, UIntType a, UIntType c, UIntType m>
    class linear_congruential_engine;

    // 26.5.3.2 Class template mersenne_twister_engine
    template<class UIntType, size_t w, size_t n, size_t m, size_t r,
             UIntType a, size_t u, UIntType d, size_t s,
             UIntType b, size_t t,
             UIntType c, size_t l, UIntType f>
    class mersenne_twister_engine;

    // 26.5.3.3 Class template subtract_with_carry_engine
    template<class UIntType, size_t w, size_t s, size_t r>
    class subtract_with_carry_engine;

    // 26.5.4.1 Class template discard_block_engine
    template<class Engine, size_t p, size_t r>
    class discard_block_engine;

    // 26.5.4.2 Class template independent_bits_engine
    template<class Engine, size_t w, class UIntType>
    class independent_bits_engine;

    // 26.5.4.3 Class template shuffle_order_engine
    template<class Engine, size_t k>
    class shuffle_order_engine;

    // 26.5.5 Engines and engine adaptors with predefined parameters
typedef see below minstd_rand0;
\end{verbatim}

\section{26.5.2 Header <random> synopsis \texttt{[rand.synopsis]}}

\begin{verbatim}
namespace std {
    #include <initializer_list>

    // 26.5.3.1 Class template linear_congruential_engine
    template<class UIntType, UIntType a, UIntType c, UIntType m>
    class linear_congruential_engine;

    // 26.5.3.2 Class template mersenne_twister_engine
    template<class UIntType, size_t w, size_t n, size_t m, size_t r,
             UIntType a, size_t u, UIntType d, size_t s,
             UIntType b, size_t t,
             UIntType c, size_t l, UIntType f>
    class mersenne_twister_engine;

    // 26.5.3.3 Class template subtract_with_carry_engine
    template<class UIntType, size_t w, size_t s, size_t r>
    class subtract_with_carry_engine;

    // 26.5.4.1 Class template discard_block_engine
    template<class Engine, size_t p, size_t r>
    class discard_block_engine;

    // 26.5.4.2 Class template independent_bits_engine
    template<class Engine, size_t w, class UIntType>
    class independent_bits_engine;

    // 26.5.4.3 Class template shuffle_order_engine
    template<class Engine, size_t k>
    class shuffle_order_engine;

    // 26.5.5 Engines and engine adaptors with predefined parameters
typedef see below minstd_rand0;
\end{verbatim}

\section{26.5.2 Header <random> synopsis \texttt{[rand.synopsis]}}

\begin{verbatim}
namespace std {
    #include <initializer_list>

    // 26.5.3.1 Class template linear_congruential_engine
    template<class UIntType, UIntType a, UIntType c, UIntType m>
    class linear_congruential_engine;

    // 26.5.3.2 Class template mersenne_twister_engine
    template<class UIntType, size_t w, size_t n, size_t m, size_t r,
             UIntType a, size_t u, UIntType d, size_t s,
             UIntType b, size_t t,
             UIntType c, size_t l, UIntType f>
    class mersenne_twister_engine;

    // 26.5.3.3 Class template subtract_with_carry_engine
    template<class UIntType, size_t w, size_t s, size_t r>
    class subtract_with_carry_engine;

    // 26.5.4.1 Class template discard_block_engine
    template<class Engine, size_t p, size_t r>
    class discard_block_engine;

    // 26.5.4.2 Class template independent_bits_engine
    template<class Engine, size_t w, class UIntType>
    class independent_bits_engine;

    // 26.5.4.3 Class template shuffle_order_engine
    template<class Engine, size_t k>
    class shuffle_order_engine;

    // 26.5.5 Engines and engine adaptors with predefined parameters
typedef see below minstd_rand0;
\end{verbatim}
typedef see below minstd_rand;
typedef see below mt19937;
typedef see below mt19937_64;
typedef see below ranlux24_base;
typedef see below ranlux48_base;
typedef see below ranlux24;
typedef see below ranlux48;
typedef see below knuth_b;
typedef see below default_random_engine;

// 26.5.6 Class random_device
class random_device;

// 26.5.7.1 Class seed_seq
class seed_seq;

// 26.5.7.2 Function template generate_canonical
template<class RealType, size_t bits, class URNG>
    RealType generate_canonical(URNG& g);

// 26.5.8.1.1 Class template uniform_int_distribution
template<class IntType = int>
    class uniform_int_distribution;

// 26.5.8.1.2 Class template uniform_real_distribution
template<class RealType = double>
    class uniform_real_distribution;

// 26.5.8.2.1 Class bernoulli_distribution
class bernoulli_distribution;

// 26.5.8.2.2 Class template binomial_distribution
template<class IntType = int>
    class binomial_distribution;

// 26.5.8.2.3 Class template geometric_distribution
template<class IntType = int>
    class geometric_distribution;

// 26.5.8.2.4 Class template negative_binomial_distribution
template<class IntType = int>
    class negative_binomial_distribution;

// 26.5.8.3.1 Class template poisson_distribution
template<class IntType = int>
    class poisson_distribution;

// 26.5.8.3.2 Class template exponential_distribution
template<class RealType = double>
    class exponential_distribution;

// 26.5.8.3.3 Class template gamma_distribution
template<class RealType = double>
    class gamma_distribution;
// 26.5.8.3.4 Class template weibull_distribution
template<class RealType = double>
class weibull_distribution;

// 26.5.8.3.5 Class template extreme_value_distribution
template<class RealType = double>
class extreme_value_distribution;

// 26.5.8.4.1 Class template normal_distribution
template<class RealType = double>
class normal_distribution;

// 26.5.8.4.2 Class template lognormal_distribution
template<class RealType = double>
class lognormal_distribution;

// 26.5.8.4.3 Class template chi_squared_distribution
template<class RealType = double>
class chi_squared_distribution;

// 26.5.8.4.4 Class template cauchy_distribution
template<class RealType = double>
class cauchy_distribution;

// 26.5.8.4.5 Class template fisher_f_distribution
template<class RealType = double>
class fisher_f_distribution;

// 26.5.8.4.6 Class template student_t_distribution
template<class RealType = double>
class student_t_distribution;

// 26.5.8.5.1 Class template discrete_distribution
template<class IntType = int>
class discrete_distribution;

// 26.5.8.5.2 Class template piecewise_constant_distribution
template<class RealType = double>
class piecewise_constant_distribution;

// 26.5.8.5.3 Class template piecewise_linear_distribution
template<class RealType = double>
class piecewise_linear_distribution;

} // namespace std

26.5.3  Random number engine class templates

1 Each type instantiated from a class template specified in this section 26.5.3 satisfies the requirements of a random number engine (26.5.1.4) type.

2 Except where specified otherwise, the complexity of each function specified in this section 26.5.3 is constant.

3 Except where specified otherwise, no function described in this section 26.5.3 throws an exception.
Descriptions are provided in this section 26.5.3 only for engine operations that are not described in 26.5.1.4 or for operations where there is additional semantic information. In particular, declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.

Each template specified in this section 26.5.3 requires one or more relationships, involving the value(s) of its non-type template parameter(s), to hold. A program instantiating any of these templates is ill-formed if any such required relationship fails to hold.

### 26.5.3.1 Class template `linear_congruential_engine`

A `linear_congruential_engine` random number engine produces unsigned integer random numbers. The state $x_i$ of a `linear_congruential_engine` object $x$ is of size 1 and consists of a single integer. The transition algorithm is a modular linear function of the form $TA(x_i) = (a \cdot x_i + c) \mod m$; the generation algorithm is $GA(x_i) = x_i + 1$.

```cpp
template<class UIntType, UIntType a, UIntType c, UIntType m>
class linear_congruential_engine
{
  public:
    // types
    typedef UIntType result_type;

    // engine characteristics
    static constexpr result_type multiplier = a;
    static constexpr result_type increment = c;
    static constexpr result_type modulus = m;
    static constexpr result_type min() { return c == 0u ? 1u: 0u; }
    static constexpr result_type max() { return m - 1u; }
    static constexpr result_type default_seed = 1u;

    // constructors and seeding functions
    explicit linear_congruential_engine(result_type s = default_seed);
    template<class Sseq> explicit linear_congruential_engine(Sseq& q);
    void seed(result_type s = default_seed);
    template<class Sseq> void seed(Sseq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);
};
```

If the template parameter $m$ is 0, the modulus $m$ used throughout this section 26.5.3.1 is `numeric_limits<result_type>::max()` plus 1. [Note: $m$ need not be representable as a value of type `result_type`. — end note]

If the template parameter $m$ is not 0, the following relations shall hold: $a < m$ and $c < m$.

The textual representation consists of the value of $x_i$.

```cpp
explicit linear_congruential_engine(result_type s = default_seed);
```

**Effects:** Constructs a `linear_congruential_engine` object. If $c \mod m$ is 0 and $s \mod m$ is 0, sets the engine’s state to 1, otherwise sets the engine’s state to $s \mod m$.

```cpp
template<class Sseq> explicit linear_congruential_engine(Sseq& q);
```
Effects: Constructs a `linear_congruential_engine` object. With \( k = \left\lfloor \log_2 m \right\rfloor \) and \( a \) an array (or equivalent) of length \( k + 3 \), invokes `q.generate(a + 0, a + k + 3)` and then computes \( S = (\sum_{j=0}^{k-1} a_{j+3} \cdot 2^{32j}) \mod m \). If \( c \mod m \) is 0 and \( S \) is 0, sets the engine’s state to 1, else sets the engine’s state to \( S \).

26.5.3.2 Class template `mersenne_twister_engine`  

A `mersenne_twister_engine` random number engine\(^{280}\) produces unsigned integer random numbers in the closed interval \([0, 2^w - 1]\). The state \( x_i \) of a `mersenne_twister_engine` object \( x \) is of size \( n \) and consists of a sequence \( X \) of \( n \) values of the type delivered by \( x \); all subscripts applied to \( X \) are to be taken modulo \( n \).

The transition algorithm employs a twisted generalized feedback shift register defined by shift values \( n \) and \( m \), a twist value \( r \), and a conditional xor-mask \( a \). To improve the uniformity of the result, the bits of the raw shift register are additionally tempered (i.e., scrambled) according to a bit-scrambling matrix defined by values \( u, d, s, b, t, c, \) and \( \ell \).

The state transition is performed as follows:

a) Concatenate the upper \( w - r \) bits of \( X_{i-n} \) with the lower \( r \) bits of \( X_{i+1-n} \) to obtain an unsigned integer value \( Y \).

b) With \( \alpha = a \cdot (Y \bith 1) \), set \( X_i \) to \( X_{i+m-n} \xor (Y \bith 1) \xor \alpha \).

The sequence \( X \) is initialized with the help of an initialization multiplier \( f \).

The generation algorithm determines the unsigned integer values \( z_1, z_2, z_3, z_4 \) as follows, then delivers \( z_4 \) as its result:

a) Let \( z_1 = X_i \xor ((X_i \bith u) \bith d) \).

b) Let \( z_2 = z_1 \xor ((z_1 \lith w s) \bith b) \).

c) Let \( z_3 = z_2 \xor ((z_2 \lith w t) \bith c) \).

d) Let \( z_4 = z_3 \xor (z_3 \bith \ell) \).

```
template<class UIntType, size_t w, size_t n, size_t m, size_t r,  
       UIntType a, size_t u, UIntType d, size_t s,  
       UIntType b, size_t t,  
       UIntType c, size_t l, UIntType f>  
class mersenne_twister_engine  
{  
  public:  
    typedef UIntType result_type;  
    // engine characteristics  
    static constexpr size_t word_size = w;  
    static constexpr size_t state_size = n;  
    static constexpr size_t shift_size = m;  
    static constexpr size_t mask_bits = r;  
    static constexpr UIntType xor_mask = a;  
    static constexpr size_t tempering_u = u;  
    static constexpr UIntType tempering_d = d;  

280) 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 constexpr size_t tempering_s = s;
static constexpr UIntType tempering_b = b;
static constexpr size_t tempering_t = t;
static constexpr UIntType tempering_c = c;
static constexpr size_t tempering_l = l;
static constexpr UIntType initialization_multiplier = f;
static constexpr result_type min () { return 0; }
static constexpr result_type max() { return 2w − 1; }
static constexpr result_type default_seed = 5489u;

// constructors and seeding functions
explicit mersenne_twister_engine(result_type value = default_seed);
template<class Sseq> explicit mersenne_twister_engine(Sseq& q);
void seed(result_type value = default_seed);
template<class Sseq> void seed(Sseq& q);

// generating functions
result_type operator()();
void discard(unsigned long long z);

Effects: Constructs a mersenne_twister_engine object. Sets X − n to value mod 2w. Then, iteratively for i = 1−n, ..., −1, sets Xi to 
\[ f \cdot (X_{i−1} \oplus (X_{i−1} \ll (w − 2))) + i \mod n \] mod 2w .

Complexity: \( O(n) \).

template<class Sseq> explicit mersenne_twister_engine(Sseq& q);

Effects: Constructs a mersenne_twister_engine object. With k = \( \lceil w/32 \rceil \) and a an array (or equivalent) of length n ∙ k, invokes q.generate(a + 0, a + n ∙ k) and then, iteratively for i = −n, ..., −1, sets Xi to \( \left( \sum_{j=0}^{k−1} a_{k(i+n)+j} \cdot 2^{32j} \right) \mod 2^w \). Finally, if the most significant w − r bits of X − n are zero, and if each of the other resulting Xi is 0, changes X − n to 2w−1.

26.5.3.3 Class template subtract_with_carry_engine

A subtract_with_carry_engine random number engine produces unsigned integer random numbers.

The state xi of a subtract_with_carry_engine object x is of size \( O(r) \), and consists of a sequence X of r integer values 0 \leq X_i < m = 2^w; all subscripts applied to X are to be taken modulo r. The state x_i additionally consists of an integer c (known as the carry) whose value is either 0 or 1.

The state transition is performed as follows:

a) Let \( Y = X_{i−s} − X_{i−r} − c \).
b) Set $X_i$ to $y = Y \mod m$. Set $c$ to 1 if $Y < 0$, otherwise set $c$ to 0.

[Note: This algorithm corresponds to a modular linear function of the form $TA(x_i) = (a \cdot x_i) \mod b$, where $b$ is of the form $m' - m'' + 1$ and $a = b - (b - 1)/m$. — end note]

The generation algorithm is given by $GA(x_i) = y$, where $y$ is the value produced as a result of advancing the engine’s state as described above.

```cpp
template<class UIntType, size_t w, size_t s, size_t r>
class subtract_with_carry_engine
{
    public:
    // types
    typedef UIntType result_type;

    // engine characteristics
    static constexpr size_t word_size = w;
    static constexpr size_t short_lag = s;
    static constexpr size_t long_lag = r;
    static constexpr result_type min() { return 0; }
    static constexpr result_type max() { return m - 1; }
    static constexpr result_type default_seed = 19780503u;

    // constructors and seeding functions
    explicit subtract_with_carry_engine(result_type value = default_seed);
    template<class Sseq> explicit subtract_with_carry_engine(Sseq& q);
    void seed(result_type value = default_seed);
    template<class Sseq> void seed(Sseq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);
}
```

The following relations shall hold: $0 < s, s < r, 0 < w$, and $w \leq \text{numeric_limits<UIntType>::digits}$.

The textual representation consists of the values of $X_{i-r}, \ldots, X_{i-1}$, in that order, followed by $c$.

```cpp
explicit subtract_with_carry_engine(result_type value = default_seed);
```

 Effects: Constructs a `subtract_with_carry_engine` object. Sets the values of $X_{i-r}, \ldots, X_{i-1}$, in that order, as specified below. If $X_{i-1}$ is then 0, sets $c$ to 1; otherwise sets $c$ to 0.

To set the values $X_k$, first construct $e$, a `linear_congruential_engine` object, as if by the following definition:

```cpp
    linear_congruential_engine<result_type, 40014u, 0u, 2147483563u> e(value == 0u ? default_seed : value);
```

Then, to set each $X_k$, obtain new values $z_0, \ldots, z_{n-1}$ from $n = \lceil w/32 \rceil$ successive invocations of $e$ taken modulo $2^{32}$. Set $X_k$ to $\left(\sum_{j=0}^{n-1} z_j \cdot 2^{32j}\right) \mod m$.

 Complexity: Exactly $n \cdot r$ invocations of $e$. 

```cpp
template<class Sseq> explicit subtract_with_carry_engine(Sseq& q);
```
Effects: Constructs a \texttt{subtract\_with\_carry\_engine} object. With \(k = \lceil w/32 \rceil\) and \(a\) an array (or equivalent) of length \(r \cdot k\), invokes \texttt{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.

26.5.4 Random number engine adaptor class templates

1 Each type instantiated from a class template specified in this section 26.5.3 satisfies the requirements of a random number engine adaptor (26.5.1.5) type.
2 Except where specified otherwise, the complexity of each function specified in this section 26.5.4 is constant.
3 Except where specified otherwise, no function described in this section 26.5.4 throws an exception.
4 Descriptions are provided in this section 26.5.4 only for adaptor operations that are not described in section 26.5.1.5 or for operations where there is additional semantic information. In particular, declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.
5 Each template specified in this section 26.5.4 requires one or more relationships, involving the value(s) of its non-type template parameter(s), to hold. A program instantiating any of these templates is ill-formed if any such required relationship fails to hold.

26.5.4.1 Class template \texttt{discard\_block\_engine}

1 A \texttt{discard\_block\_engine} random number engine adaptor produces random numbers selected from those produced by some base engine \(e\). The state \(x_i\) of a \texttt{discard\_block\_engine} engine adaptor object \(x\) consists of the state \(e_i\) of its base engine \(e\) and an additional integer \(n\). The size of the state is the size of \(e\)'s state plus 1.
2 The transition algorithm discards all but \(r > 0\) values from each block of \(p \geq r\) values delivered by \(e\). The state transition is performed as follows: If \(n \geq r\), advance the state of \(e\) from \(e_i\) to \(e_{i+p-r}\) and set \(n\) to 0. In any case, then increment \(n\) and advance \(e\)'s then-current state \(e_j\) to \(e_{j+1}\).
3 The generation algorithm yields the value returned by the last invocation of \(e()\) while advancing \(e\)'s state as described above.

    template<class Engine, size_t p, size_t r>
    class discard_block_engine
    {
    public:
        // types
        typedef typename Engine::result_type result_type;

        // engine characteristics
        static constexpr size_t block_size = p;
        static constexpr size_t used_block = r;
        static constexpr result_type min() { return Engine::min; }
        static constexpr result_type max() { return Engine::max; }

        // constructors and seeding functions
        discard_block_engine();
        explicit discard_block_engine(const Engine& e);
        explicit discard_block_engine(Engine&& e);
        explicit discard_block_engine(result_type s);
        template<class Sseq> explicit discard_block_engine(Sseq& q);
        void seed();
void seed(result_type s);
  template<class Sseq> void seed(Sseq& q);

  // generating functions
  result_type operator()();
  void discard(unsigned long long z);

  // property functions
  const Engine& base() const;

private:
  Engine e; // exposition only
  int n;    // exposition only
};

The following relations shall hold: 0 < r and r <= p.
The textual representation consists of the textual representation of e followed by the value of n.

In addition to its behavior pursuant to section 26.5.1.5, each constructor that is not a copy constructor sets n to 0.

### 26.5.4.2 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.

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 \( w_0 = \lfloor \frac{w}{n} \rfloor \), \( n_0 = n - w \mod n \), \( y_0 = 2^{w_0} \lfloor \frac{R}{2^{w_0}} \rfloor \), and \( y_1 = 2^{w_0+1} \lfloor \frac{R}{2^{w_0+1}} \rfloor \).

c) Let \( n = \lceil \frac{w}{m} \rceil \) if and only if the relation \( R - y_0 \leq \lfloor \frac{y_0}{n} \rfloor \) holds as a result. Otherwise let \( n = 1 + \lceil \frac{w}{m} \rceil \).

[Note: The relation \( w = n_0 w_0 + (n - n_0)(w_0 + 1) \) always holds. — end note]

The transition algorithm is carried out by invoking \( e() \) as often as needed to obtain \( n_0 \) values less than \( y_0 + e.\min() \) and \( n - n_0 \) values less than \( y_1 + e.\min() \).

The generation algorithm uses the values produced while advancing the state as described above to yield a quantity \( S \) obtained as if by the following algorithm:

\[
S = 0;
\]

\[
\text{for} \ (k = 0; \ k \neq n_0; \ k++ = 1) \ \{ \\
  \text{do} \ u = e() - e.\min(); \text{while} \ (u \geq y_0); \\
  \ S = 2^{w_0} \cdot S + u \mod 2^{w_0}; \\
\}
\]

\[
\text{for} \ (k = n_0; \ k \neq n; \ k++ = 1) \ \{ \\
  \text{do} \ u = e() - e.\min(); \text{while} \ (u \geq y_0); \\
  \ S = 2^{w_0+1} \cdot S + u \mod 2^{w_0+1}; \\
\}
\]
template<class Engine, size_t w, class UIntType>
class independent_bits_engine
{
public:
    // types
typedef UIntType result_type;

    // engine characteristics
    static constexpr result_type min() { return 0; }
    static constexpr result_type max() { return 2^w - 1; }

    // constructors and seeding functions
    independent_bits_engine();
    explicit independent_bits_engine(const Engine& e);
    explicit independent_bits_engine(Engine&& e);
    explicit independent_bits_engine(result_type s);
    template<class Sseq> explicit independent_bits_engine(Sseq& q);
    void seed();
    void seed(result_type s);
    template<class Sseq> void seed(Sseq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);

    // property functions
    const Engine& base() const;

private:
    Engine e;  // exposition only
};

The following relations shall hold: 0 < w and w <= numeric_limits<result_type>::digits.

The textual representation consists of the textual representation of e.

26.5.4.3 Class template shuffle_order_engine [rand.adapt.shuf]

A shuffle_order_engine random number engine adaptor produces the same random numbers that are
produced by some base engine e, but delivers them in a different sequence. The state x_i of a shuffle_order_engine engine adaptor object x consists of the state e_i of its base engine e, an additional value Y of
the type delivered by e, and an additional sequence V of k values also of the type delivered by e. The size
of the state is the size of e’s state plus k + 1.

The transition algorithm permutes the values produced by e. The state transition is performed as follows:

a) Calculate an integer \( j = \left\lfloor \frac{k \cdot (Y - e_{\min})}{e_{\max} - e_{\min} + 1} \right\rfloor \).

b) Set Y to V_j and then set V_j to e().

The generation algorithm yields the last value of Y produced while advancing e’s state as described above.
// types
typedef typename Engine::result_type result_type;

// engine characteristics
static constexpr size_t table_size = k;
static constexpr result_type min() { return Engine::min; }
static constexpr result_type max() { return Engine::max; }

// constructors and seeding functions
shuffle_order_engine();
explicit shuffle_order_engine(const Engine& e);
explicit shuffle_order_engine(Engine&& e);
extric explicit shuffle_order_engine(result_type s);
template<class Sseq> explicit shuffle_order_engine(Sseq& q);
void seed();
void seed(result_type s);
template<class Sseq> void seed(Sseq& q);

// generating functions
result_type operator()();
void discard(unsigned long long z);

// property functions
const Engine& base() const;

private:
    Engine e;          // exposition only
    result_type Y;     // exposition only
    result_type V[k];  // exposition only
};

4 The following relation shall hold: 0 < k.
5 The textual representation consists of the textual representation of e, followed by the k values of V, followed by the value of Y.
6 In addition to its behavior pursuant to section 26.5.1.5, each constructor that is not a copy constructor initializes V[0],...,V[k-1] and Y, in that order, with values returned by successive invocations of e().

26.5.5 Engines and engine adaptors with predefined parameters [rand.predef]

typedef linear_congruential_engine<uint_fast32_t, 16807, 0, 2147483647> minstd_rand0;

1 Required behavior: The 10000th consecutive invocation of a default-constructed object of type minstd_rand0 shall produce the value 1043618065.

typedef linear_congruential_engine<uint_fast32_t, 48271, 0, 2147483647> minstd_rand;

2 Required behavior: The 10000th consecutive invocation of a default-constructed object of type minstd_rand shall produce the value 399268537.

typedef mersenne_twister_engine<uint_fast32_t>;
Required behavior: The 10000th consecutive invocation of a default-constructed object of type `mt19937` shall produce the value 4123659995.

typedef mersenne_twister_engine<uint_fast64_t,
  64,312,156,31,0xb5026f5aa96619e9,29,
  0x5555555555555555,17,
  0x71d67fffedada60000,37,
  0xfffffee000000000,43,
  6364136223846793005>
  mt19937_64;

Required behavior: The 10000th consecutive invocation of a default-constructed object of type `mt19937_64` shall produce the value 9981545732273789042.

typedef subtract_with_carry_engine<uint_fast32_t, 24, 10, 24>
  ranlux24_base;

Required behavior: The 10000th consecutive invocation of a default-constructed object of type `ranlux24_base` shall produce the value 7937952.

typedef subtract_with_carry_engine<uint_fast64_t, 48, 5, 12>
  ranlux48_base;

Required behavior: The 10000th consecutive invocation of a default-constructed object of type `ranlux48_base` shall produce the value 61839128582725.

typedef discard_block_engine<ranlux24_base, 223, 23>
  ranlux24;

Required behavior: The 10000th consecutive invocation of a default-constructed object of type `ranlux24` shall produce the value 9901578.

typedef discard_block_engine<ranlux48_base, 389, 11>
  ranlux48;

Required behavior: The 10000th consecutive invocation of a default-constructed object of type `ranlux48` shall produce the value 249142670248501.

typedef shuffle_order_engine<minstd_rand0,256>
  knuth_b;

Required behavior: The 10000th consecutive invocation of a default-constructed object of type `knuth_b` shall produce the value 1112339016.

typedef implementation-defined
  default_random_engine;
Remark: The choice of engine type named by this typedef is implementation-defined. [Note: The implementation may select this type on the basis of performance, size, quality, or any combination of such factors, so as to provide at least acceptable engine behavior for relatively casual, inexpert, and/or lightweight use. Because different implementations may select different underlying engine types, code that uses this typedef need not generate identical sequences across implementations. —end note]

26.5.6 Class random_device

A random_device uniform random number generator produces non-deterministic random numbers.

If implementation limitations prevent generating non-deterministic random numbers, the implementation may employ a random number engine.

```cpp
class random_device
{
public:
  // types
  typedef unsigned int result_type;

  // generator characteristics
  static constexpr result_type min() { return numeric_limits<result_type>::min(); }  
  static constexpr result_type max() { return numeric_limits<result_type>::max(); }  

  // constructors
  explicit random_device(const string& token = implementation-defined);

  // generating functions
  result_type operator()();

  // property functions
  double entropy() const;

  // no copy functions
  random_device(const random_device&) = delete;
  void operator=(const random_device&) = delete;
};

explicit random_device(const string& token = implementation-defined);
```

Effects: Constructs a random_device non-deterministic uniform random number generator object. The semantics and default value of the token parameter are implementation-defined.281

Returns: If the implementation employs a random number engine, returns 0.0. Otherwise, returns an entropy estimate282 for the random numbers returned by operator(), in the range min() to \( \log_2(\text{max}() + 1) \).

281) The parameter is intended to allow an implementation to differentiate between different sources of randomness.
282) If a device has \( n \) states whose respective probabilities are \( P_0, \ldots, P_{n-1} \), the device entropy \( S \) is defined as

\[
S = -\sum_{i=0}^{n-1} P_i \cdot \log P_i.
\]
6  *Throws:* Nothing.

```cpp
result_type operator()();
```

7  *Returns:* A non-deterministic random value, uniformly distributed between \( \min() \) and \( \max() \), inclusive. It is implementation-defined how these values are generated.

8  *Throws:* A value of an implementation-defined type derived from `exception` if a random number could not be obtained.

### 26.5.7 Utilities

#### 26.5.7.1 Class `seed_seq` [rand.util.seedseq]

1  No function described in this section 26.5.7.1 throws an exception.

```cpp
class seed_seq
{
public:
  // types
  typedef uint_least32_t result_type;

  // constructors
  seed_seq();
  template<class T>
  seed_seq(initializer_list<T> il);
  template<class InputIterator>
  seed_seq(InputIterator begin, InputIterator end);

  // generating functions
  template<class RandomAccessIterator>
  void generate(RandomAccessIterator begin, RandomAccessIterator end);

  // property functions
  size_t size() const;
  template<class OutputIterator>
  void param(OutputIterator dest) const;

  // no copy functions
  seed_seq(const seed_seq&) = delete;
  void operator=(const seed_seq&) = delete;

private:
  vector<result_type> v;  // exposition only
};
```

2  *Effects:* Constructs a `seed_seq` object as if by default-constructing its member \( v \).

```cpp
template<class T>
seed_seq(initializer_list<T> il);
```

3  *Requires:* \( T \) shall be an integer type.
Effects: Same as seed_seq(il.begin(), il.end()).

```cpp
template<class InputIterator>
seed_seq(InputIterator begin, InputIterator end);
```

Requires: InputIterator shall satisfy the requirements of an input iterator (Table 104) type. Moreover, iterator_traits<InputIterator>::value_type shall denote an integer type.

Effects: Constructs a seed_seq object by the following algorithm:

```cpp
for( InputIterator s = begin; s != end; ++s)
v.push_back((*s) mod2 32);
```

```cpp
template<class RandomAccessIterator>
void generate(RandomAccessIterator begin, RandomAccessIterator end);
```

Requires: RandomAccessIterator shall meet the requirements of a mutable random access iterator (Table 108) type. Moreover, iterator_traits<RandomAccessIterator>::value_type shall denote an unsigned integer type capable of accommodating 32-bit quantities.

Effects: Does nothing if begin == end. Otherwise, with s = v.size() and n = end − begin, fills the supplied range [begin,end) according to the following algorithm in which each operation is to be carried out modulo 232, each indexing operator applied to begin is to be taken modulo n, and T(x) is defined as x xor (x rshift 27):

a) By way of initialization, set each element of the range to the value 0x8b8b8b8b. Additionally, for use in subsequent steps, let p = (n − t)/2 and let q = p + t, where

```cpp
t = (n ≥ 623) ? 11 : (n ≥ 68) ? 7 : (n ≥ 39) ? 5 : (n ≥ 7) ? 3 : (n − 1)/2;
```

b) With m as the larger of s + 1 and n, transform the elements of the range: iteratively for k = 0,...,m − 1, calculate values

```cpp
\begin{align*}
    r_1 &= 1664525 \cdot T(begin[k] \ xor \ begin[k+p] \ xor \ begin[k-1]) \\
    r_2 &= r_1 + \begin{cases} 
        s & k = 0 \\
        k \ mod \ n + v[k-1] & 0 < k \leq s \\
        k \ mod \ n & s < k
            \end{cases}
\end{align*}
```

and, in order, increment begin[k+p] by r1, increment begin[x+q] by r2, and set begin[k] to r2.

c) Transform the elements of the range three more times, beginning where the previous step ended: iteratively for k = m,...,m+n−1, calculate values

```cpp
\begin{align*}
    r_3 &= 1566083941 \cdot T(begin[k] + begin[k+p] + begin[k-1]) \\
    r_4 &= r_3 - (k \ mod \ n)
\end{align*}
```

and, in order, update begin[k+p] by xoring it with r4, update begin[k+q] by xoring it with r3, and set begin[k] to r4.
Returns: The number of 32-bit units that would be returned by a call to param().

Complexity: constant time.

```cpp
template<class OutputIterator>
void param(OutputIterator dest) const;
```

Requires: OutputIterator shall satisfy the requirements of an output iterator (Table 105) type. Moreover, the expression *dest = rt shall be valid for a value rt of type result_type.

Effects: Copies the sequence of prepared 32-bit units to the given destination, as if by executing the following statement:

```cpp
copy(v.begin(), v.end(), dest);
```

### 26.5.7.2 Function template `generate_canonical` [rand.util.canonical]

Each function instantiated from the template described in this section 26.5.7.2 maps the result of one or more invocations of a supplied uniform random number generator g to one member of the specified RealType such that, if the values $g_i$ produced by g are uniformly distributed, the instantiation’s results $t_j$, $0 \leq t_j < 1$, are distributed as uniformly as possible as specified below.

[Note: Obtaining a value in this way can be a useful step in the process of transforming a value generated by a uniform random number generator into a value that can be delivered by a random number distribution. — end note]

```cpp
template<class RealType, size_t bits, class URNG>
RealType generate_canonical(URNG& g);
```

Complexity: Exactly $k = \max(1, \lceil b / \log_2 R \rceil)$ invocations of g, where $b^{283}$ is the lesser of `numeric_limits<RealType>::digits` and `bits`, and $R$ is the value of `g.max() - g.min()` + 1.

Effects: Invokes g() $k$ times to obtain values $g_0, \ldots, g_{k-1}$, respectively. Calculates a quantity

$$S = \sum_{i=0}^{k-1} (g_i - g\text{.min()} \cdot R^i$$

using arithmetic of type RealType.

Returns: $S/R^k$.

Throws: What and when g throws.

### 26.5.8 Random number distribution class templates [rand.dist]

Each type instantiated from a class template specified in this section 26.5.8 satisfies the requirements of a random number distribution (26.5.1.6) type.

Descriptions are provided in this section 26.5.8 only for distribution operations that are not described in 26.5.1.6 or for operations where there is additional semantic information. In particular, declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.

\[283\] $b$ is introduced to avoid any attempt to produce more bits of randomness than can be held in RealType.

§ 26.5.8
The algorithms for producing each of the specified distributions are implementation-defined.

The value of each probability density function \( p(z) \) and of each discrete probability function \( P(z_i) \) specified in this section is 0 everywhere outside its stated domain.

### 26.5.8.1 Uniform distributions

#### 26.5.8.1.1 Class template `uniform_int_distribution`

A `uniform_int_distribution` random number distribution produces random integers \( i, a \leq i \leq b \), distributed according to the constant discrete probability function

\[
P(i | a, b) = \frac{1}{b - a + 1}.
\]

```cpp
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 URNG>
  result_type operator()(URNG& g);
  template<class URNG>
  result_type operator()(URNG& g, const param_type& parm);

  // property functions
  result_type a() const;
  result_type b() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
};
```

**Requires:** \( a \leq b \).

**Effects:** Constructs a `uniform_int_distribution` object; \( a \) and \( b \) correspond to the respective parameters of the distribution.

**Returns:** The value of the \( a \) parameter with which the object was constructed.

**Returns:** The value of the \( b \) parameter with which the object was constructed.
26.5.8.1.2 Class template uniform_real_distribution

A `uniform_real_distribution` random number distribution produces random numbers \( x, a \leq x < b \), distributed according to the constant probability density function

\[
p(x \mid a, b) = 1/(b - a).
\]

template<class RealType = double>
class uniform_real_distribution
{
public:
  // types
  typedef RealType result_type;
  typedef unspecified param_type;
  // constructors and reset functions
  explicit uniform_real_distribution(RealType a = 0.0, RealType b = 1.0);
  explicit uniform_real_distribution(const param_type& parm);
  void reset();
  // generating functions
  template<class URNG>
  result_type operator()(URNG& g);
  template<class URNG>
  result_type operator()(URNG& g, const param_type& parm);
  // property functions
  result_type a() const;
  result_type b() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
};

explicit uniform_real_distribution(RealType a = 0.0, RealType b = 1.0);

Requires: \( a \leq b \) and \( b - a \leq \text{numeric_limits<RealType>::max()} \).

Effects: Constructs a `uniform_real_distribution` object; \( a \) and \( b \) correspond to the respective parameters of the distribution.

result_type a() const;

Returns: The value of the \( a \) parameter with which the object was constructed.

result_type b() const;

Returns: The value of the \( b \) parameter with which the object was constructed.
26.5.8.2 Bernoulli distributions

26.5.8.2.1 Class bernoulli_distribution

A bernoulli_distribution random number distribution produces bool values \( b \) distributed according to the discrete probability function

\[
P(b \mid p) = \begin{cases} 
  p & \text{if } b = \text{true} \\
  1 - p & \text{if } b = \text{false}
\end{cases}
\]

```cpp
class bernoulli_distribution
{
  public:
    // types
    typedef bool result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit bernoulli_distribution(double p = 0.5);
    explicit bernoulli_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URNG>
    result_type operator()(URNG& g);
    template<class URNG>
    result_type operator()(URNG& g, const param_type& parm);

    // property functions
    double p() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

explicit bernoulli_distribution(double p = 0.5);

**Requires:** \( 0 \leq p \leq 1 \).

**Effects:** Constructs a bernoulli_distribution object; \( p \) corresponds to the parameter of the distribution.

double p() const;

**Returns:** The value of the \( p \) parameter with which the object was constructed.

26.5.8.2.2 Class template binomial_distribution

A binomial_distribution random number distribution produces integer values \( i \geq 0 \) distributed according to the discrete probability function

\[
P(i \mid t, p) = \binom{t}{i} \cdot p^i \cdot (1 - p)^{t-i}.
\]

§ 26.5.8.2.2
template<class IntType = int>
  class binomial_distribution
{
public:
  // types
  typedef IntType result_type;
  typedef unspecified param_type;

  // constructors and reset functions
  explicit binomial_distribution(IntType t = 1, double p = 0.5);
  explicit binomial_distribution(const param_type& parm);
  void reset();

  // generating functions
  template<class URNG>
  result_type operator()(URNG& g);
  template<class URNG>
  result_type operator()(URNG& g, const param_type& parm);

  // property functions
  IntType t() const;
  double p() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
};

explicit binomial_distribution(IntType t = 1, double p = 0.5);

2  Requires: 0 ≤ p ≤ 1 and 0 ≤ t.
3  Effects: Constructs a binomial_distribution object; t and p correspond to the respective parameters of the distribution.

IntType t() const;
4  Returns: The value of the t parameter with which the object was constructed.

double p() const;
5  Returns: The value of the p parameter with which the object was constructed.

26.5.8.2.3 Class template geometric_distribution [rand.dist.bern.geo]

A geometric_distribution random number distribution produces integer values \( i \geq 0 \) distributed according to the discrete probability function

\[
P(i \mid p) = p \cdot (1 - p)^i.
\]

template<class IntType = int>
  class geometric_distribution
{
public:

§ 26.5.8.2.3
// 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 URNG>
result_type operator()(URNG& g);

// property functions
double p() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;
};

explicit geometric_distribution(double p = 0.5);

2 Requires: $0 < p < 1$.
3 Effects: Constructs a geometric_distribution object; $p$ corresponds to the parameter of the distribution.

double p() const;

4 Returns: The value of the $p$ parameter with which the object was constructed.

26.5.8.2.4 Class template negative_binomial_distribution [rand.dist.bern.negbin]
1 A negative_binomial_distribution random number distribution produces random integers $i \geq 0$ distributed according to the discrete probability function

$$P(i|k,p) = \binom{k + i - 1}{i} \cdot p^k \cdot (1 - p)^i.$$
// generating functions
template<class URNG>
  result_type operator()(URNG& g);
template<class URNG>
  result_type operator()(URNG& g, const param_type& parm);

// property functions
IntType k() const;
double p() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;
};

explicit negative_binomial_distribution(IntType k = 1, double p = 0.5);

Requirements: 0 < p ≤ 1 and 0 < k.

Effects: Constructs a negative_binomial_distribution object; k and p correspond to the respective parameters of the distribution.

IntType k() const;

Returns: The value of the k parameter with which the object was constructed.

double p() const;

Returns: The value of the p parameter with which the object was constructed.

26.5.8.3 Poisson distributions [rand.dist.pois]

26.5.8.3.1 Class template poisson_distribution [rand.dist.pois.poisson]

A poisson_distribution random number distribution produces integer values i ≥ 0 distributed according to the discrete probability function

\[ P(i | \mu) = \frac{e^{-\mu} \mu^i}{i!} \]

The distribution parameter \( \mu \) is also known as this distribution's mean.

template<class IntType = int>
  class poisson_distribution
{
  public:
    // types
    typedef IntType result_type;
    typedef unspecified param_type;
    // constructors and reset functions
    explicit poisson_distribution(double mean = 1.0);
    explicit poisson_distribution(const param_type& parm);
    void reset();

\[ 26.5.8.3.1 926 \]
// generating functions
template<class URNG>
  result_type operator()(URNG& g);
template<class URNG>
  result_type operator()(URNG& g, const param_type& parm);

// property functions
double mean() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;
};

explicit poisson_distribution(double mean = 1.0);

2 Requires: 0 < mean.
3 Effects: Constructs a poisson_distribution object; mean corresponds to the parameter of the distribution.

double mean() const;

4 Returns: The value of the mean parameter with which the object was constructed.

26.5.8.3.2 Class template exponential_distribution [rand.dist.pois.exp]

An exponential_distribution random number distribution produces random numbers x > 0 distributed according to the probability density function

\[ p(x | \lambda) = \lambda e^{-\lambda x} . \]

template<class RealType = double>
class exponential_distribution
{
  public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit exponential_distribution(RealType lambda = 1.0);
    explicit exponential_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URNG>
      result_type operator()(URNG& g);
    template<class URNG>
      result_type operator()(URNG& g, const param_type& parm);

    // property functions
    RealType lambda() const;
    param_type param() const;

§ 26.5.8.3.2
void param(const param_type& parm);
result_type min() const;
result_type max() const;
};

explicit exponential_distribution(RealType lambda = 1.0);

Requires: 0 < lambda.
Effects: Constructs a exponential_distribution object; lambda corresponds to the parameter of the distribution.

RealType lambda() const;

Returns: The value of the lambda parameter with which the object was constructed.

26.5.8.3.3 Class template gamma_distribution

A gamma_distribution random number distribution produces random numbers \( x > 0 \) distributed according to the probability density function

\[
p(x | \alpha, \beta) = \frac{e^{-x/\beta}}{\beta^\alpha \cdot \Gamma(\alpha)} \cdot x^{\alpha - 1}.
\]

template<class RealType = double>
class gamma_distribution
{
public:
  // types
  typedef RealType result_type;
  typedef unspecified param_type;

  // constructors and reset functions
  explicit gamma_distribution(RealType alpha = 1.0, RealType beta = 1.0);
  explicit gamma_distribution(const param_type& parm);
  void reset();

  // generating functions
  template<class URNG>
  result_type operator()(URNG& g);
  template<class URNG>
  result_type operator()(URNG& g, const param_type& parm);

  // property functions
  RealType alpha() const;
  RealType beta() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
};

explicit gamma_distribution(RealType alpha = 1.0, RealType beta = 1.0);
Requires: $0 < \alpha$ and $0 < \beta$.

Effects: Constructs a \texttt{gamma\_distribution} object; \texttt{alpha} and \texttt{beta} correspond to the parameters of the distribution.

\begin{verbatim}
RealType alpha() const;
\end{verbatim}

\textit{Returns:} The value of the \texttt{alpha} parameter with which the object was constructed.

\begin{verbatim}
RealType beta() const;
\end{verbatim}

\textit{Returns:} The value of the \texttt{beta} parameter with which the object was constructed.

26.5.8.3.4 Class template \texttt{weibull\_distribution} \hfill [rand.dist.pois.weibull]

A \texttt{weibull\_distribution} random number distribution produces random numbers $x \geq 0$ distributed according to the probability density function

$$p(x \mid a, b) = \frac{a}{b} \cdot \left(\frac{x}{b}\right)^{a-1} \cdot \exp\left(-\left(\frac{x}{b}\right)^a\right).$$

\begin{verbatim}
template<class RealType = double>
    class weibull_distribution
    {
        public:
            // types
            typedef RealType result_type;
            typedef unspecified param_type;

            // constructor and reset functions
            explicit weibull_distribution(RealType a = 1.0, RealType b = 1.0);
            explicit weibull_distribution(const param_type& parm);
            void reset();

            // generating functions
            template<class URNG>
                result_type operator()(URNG& g);
            template<class URNG>
                result_type operator()(URNG& g, const param_type& parm);

            // property functions
            RealType a() const;
            RealType b() const;
            param_type param() const;
            void param(const param_type& parm);
            result_type min() const;
            result_type max() const;
    };
\end{verbatim}

explicit weibull_distribution(RealType a = 1.0, RealType b = 1.0);

Requires: $0 < a$ and $0 < b$.

Effects: Constructs a \texttt{weibull\_distribution} object; \texttt{a} and \texttt{b} correspond to the respective parameters of the distribution.
RealType a() const;

Returns: The value of the \( a \) parameter with which the object was constructed.

RealType b() const;

Returns: The value of the \( b \) parameter with which the object was constructed.

### 26.5.8.3.5 Class template \texttt{extreme\_value\_distribution} \[\text{[rand.dist.pois.extreme]}\]

An \texttt{extreme\_value\_distribution} random number distribution produces random numbers \( x \) distributed according to the probability density function\(^{284}\)

\[
p(x | a, b) = \frac{1}{b} \cdot \exp \left( \frac{a - x}{b} \right) - \exp \left( \frac{a - x}{b} \right).
\]

\(^{284}\) 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.
RealType a() const;

        Returns: The value of the a parameter with which the object was constructed.

RealType b() const;

        Returns: The value of the b parameter with which the object was constructed.

26.5.8.4 Normal distributions

26.5.8.4.1 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
template<class RealType = double>
class normal_distribution
{
public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit normal_distribution(RealType mean = 0.0, RealType stddev = 1.0);
    explicit normal_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URNG>
    result_type operator()(URNG& g);
    template<class URNG>
    result_type operator()(URNG& g, const param_type& parm);

    // property functions
    RealType mean() const;
    RealType stddev() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

explicit normal_distribution(RealType mean = 0.0, RealType stddev = 1.0);

        Requires: \( 0 < \text{stddev} \).

        Effects: Constructs a normal_distribution object; mean and stddev correspond to the respective parameters of the distribution.

§ 26.5.8.4.1
RealType mean() const;

Returns: The value of the mean parameter with which the object was constructed.

RealType stddev() const;

Returns: The value of the stddev parameter with which the object was constructed.

### 26.5.8.4.2 Class template lognormal_distribution

A lognormal_distribution random number distribution produces random numbers \( x > 0 \) distributed according to the probability density function

\[
p(x | m, s) = \frac{1}{sx\sqrt{2\pi}} \cdot \exp \left( -\frac{(\ln x - m)^2}{2s^2} \right).
\]

```cpp
template<class RealType = double>
class lognormal_distribution
{
public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    explicit lognormal_distribution(RealType m = 0.0, RealType s = 1.0);
    explicit lognormal_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URNG>
    result_type operator()(URNG& g);
    template<class URNG>
    result_type operator()(URNG& g, const param_type& parm);

    // property functions
    RealType m() const;
    RealType s() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

explicit lognormal_distribution(RealType m = 0.0, RealType s = 1.0);

Requires: \( 0 < s \).

Effects: Constructs a lognormal_distribution object; \( m \) and \( s \) correspond to the respective parameters of the distribution.

RealType m() const;

Returns: The value of the \( m \) parameter with which the object was constructed.
RealType s() const;

Returns: The value of the s parameter with which the object was constructed.

26.5.8.4.3 Class template chi_squared_distribution

A chi_squared_distribution random number distribution produces random numbers \( x > 0 \) distributed according to the probability density function

\[
p(x \mid n) = \frac{x^{(n/2)-1} \cdot e^{-x/2}}{\Gamma(n/2) \cdot 2^{n/2}}.
\]

template<class RealType = double>
class chi_squared_distribution
{
public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    explicit chi_squared_distribution(RealType n = 1);
    explicit chi_squared_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URNG>
    result_type operator()(URNG& g);
    template<class URNG>
    result_type operator()(URNG& g, const param_type& parm);

    // property functions
    RealType n() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};

explicit chi_squared_distribution(RealType n = 1);

Requires: 0 < n.

Effects: Constructs a chi_squared_distribution object; n corresponds to the parameter of the distribution.

RealType n() const;

Returns: The value of the n parameter with which the object was constructed.
26.5.8.4.4 Class template cauchy_distribution

1 A `cauchy_distribution` random number distribution produces random numbers \( x \) distributed according to the probability density function

\[
p(x \mid a, b) = \left( \pi b \left( 1 + \left( \frac{x - a}{b} \right)^2 \right) \right)^{-1}.
\]

```cpp
template<class RealType = double>
    class cauchy_distribution
    {
    public:
        // types
        typedef RealType result_type;
        typedef unspecified param_type;

        // constructor and reset functions
        explicit cauchy_distribution(RealType a = 0.0, RealType b = 1.0);
        explicit cauchy_distribution(const param_type& parm);
        void reset();

        // generating functions
        template<class URNG>
            result_type operator()(URNG& g);
        template<class URNG>
            result_type operator()(URNG& g, const param_type& parm);

        // property functions
        RealType a() const;
        RealType b() const;
        param_type param() const;
        void param(const param_type& parm);
        result_type min() const;
        result_type max() const;
    };
```

2 `explicit cauchy_distribution(RealType a = 0.0, RealType b = 1.0);`

3 **Requires:** \(0 < b\).

3 **Effects:** Constructs a `cauchy_distribution` object; \(a\) and \(b\) correspond to the respective parameters of the distribution.

4 **Returns:** The value of the \(a\) parameter with which the object was constructed.

5 **Returns:** The value of the \(b\) parameter with which the object was constructed.
26.5.8.4.5  Class template fisher_f_distribution

1 A fisher_f_distribution random number distribution produces random numbers \( x \geq 0 \) distributed according to the probability density function

\[
p(x \mid m, n) = \frac{\Gamma((m + n)/2)}{\Gamma(m/2) \Gamma(n/2)} \left( \frac{m}{n} \right)^{m/2} \cdot x^{(m/2)-1} \cdot \left( 1 + \frac{mx}{n} \right)^{-(m+n)/2}.
\]

template<class RealType = double>
class fisher_f_distribution
{
    public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    explicit fisher_f_distribution(RealType m = 1, RealType n = 1);
    explicit fisher_f_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URNG>
    result_type operator()(URNG& g);
    template<class URNG>
    result_type operator()(URNG& g, const param_type& parm);

    // property functions
    RealType m() const;
    RealType n() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};

explicit fisher_f_distribution(RealType m = 1, RealType n = 1);

2 Requires: \( 0 < m \) and \( 0 < n \).

3 Effects: Constructs a fisher_f_distribution object; \( m \) and \( n \) correspond to the respective parameters of the distribution.

RealType m() const;

4 Returns: The value of the \( m \) parameter with which the object was constructed.

RealType n() const;

5 Returns: The value of the \( n \) parameter with which the object was constructed.
Class template student_t_distribution

A student_t_distribution random number distribution produces random numbers \( x \) distributed according to the probability density function

\[
p(x \mid n) = \frac{1}{\sqrt{n\pi}} \frac{\Gamma((n+1)/2)}{\Gamma(n/2)} \left(1 + \frac{x^2}{n}\right)^{-(n+1)/2}.
\]

```
template<class RealType = double>
  class student_t_distribution
  {
    public:
      // types
      typedef RealType result_type;
      typedef unspecified param_type;

      // constructor and reset functions
      explicit student_t_distribution(RealType n = 1);
      explicit student_t_distribution(const param_type& parm);
      void reset();

      // generating functions
      template<class URNG>
        result_type operator()(URNG& g);
      template<class URNG>
        result_type operator()(URNG& g, const param_type& parm);

      // property functions
      RealType n() const;
      param_type param() const;
      void param(const param_type& parm);
      result_type min() const;
      result_type max() const;
  };
```

2

Requires: \( 0 < n \).

3

Effects: Constructs a student_t_distribution object; \( n \) corresponds to the parameter of the distribution.

RealType n() const;

4

Returns: The value of the \( n \) parameter with which the object was constructed.

Sampling distributions

Class template discrete_distribution

A discrete_distribution random number distribution produces random integers \( i, 0 \leq i < n \), distributed according to the discrete probability function

\[
P(i \mid p_0, \ldots, p_{n-1}) = p_i.
\]
Unless specified otherwise, the distribution parameters are calculated as: \( p_k = \frac{w_k}{S} \) for \( k = 0, \ldots, n-1 \), in which the values \( w_k \), commonly known as the *weights*, shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold: \( 0 < S = w_0 + \cdots + w_{n-1} \).

```cpp
template<class IntType = int>
class discrete_distribution
{
  public:
    // types
    typedef IntType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    discrete_distribution();
    template<class InputIterator>
    discrete_distribution(InputIterator firstW, InputIterator lastW);
    discrete_distribution(initializer_list<double> wl);
    template<class UnaryOperation>
    discrete_distribution(size_t nw, double xmin, double xmax, UnaryOperation fw);
    explicit discrete_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URNG>
    result_type operator()(URNG& g);
    template<class URNG>
    result_type operator()(URNG& g, const param_type& parm);

    // property functions
    vector<double> probabilities() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

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 UnaryOperation>
  discrete_distribution(size_t nw, double xmin, double xmax, UnaryOperation fw);

Requires: Each instance of type UnaryOperation shall be a function object (20.8) whose return type shall be convertible to double. Moreover, double shall be convertible to the type of UnaryOperation’s sole parameter. If nw = 0, let n = 1, otherwise let n = nw. The relation 0 < δ = (xmax − xmin)/n shall hold.

Effects: Constructs a discrete_distribution object with probabilities given by the formula above, using the following values: If nw = 0, let w0 = 1. Otherwise, let wk = fw(xmin + k · δ + δ/2) for k = 0, ..., n−1.

Complexity: The number of invocations of fw shall not exceed n.

vector<double> probabilities() const;

Returns: A vector<double> whose size member returns n and whose operator[] member returns pk when invoked with argument k for k = 0, ..., n−1.

26.5.8.5.2 Class template piecewise_constant_distribution

A piecewise_constant_distribution random number distribution produces random numbers x, b0 ≤ x < bn, uniformly distributed over each subinterval [bi, bi+1) according to the probability density function

\[ p(x | b_0, \ldots, b_n, \rho_0, \ldots, \rho_{n-1}) = \rho_i , \text{ for } b_i \leq x < b_{i+1} . \]

The n + 1 distribution parameters bi, also known as this distribution’s interval boundaries, shall satisfy the relation bi < bi+1 for i = 0, ..., n−1. Unless specified otherwise, the remaining n distribution parameters are calculated as:

\[ \rho_k = \frac{w_k}{S \cdot (b_{k+1} - b_k)} \text{ for } k = 0, ..., n-1, \]

in which the values wk, commonly known as the weights, shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold: 0 < S = w0 + ⋯ + wn−1.
// generating functions
template<class URNG>
result_type operator()(URNG& g);
// property functions
vector<result_type> 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 \texttt{piecewise} \texttt{constant} \texttt{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: InputIteratorB and InputIteratorW shall each satisfy the requirements of an input iterator (Table \ref{table:input-iterator-types}) type. Moreover, iterator_traits\texttt{<InputIteratorB>}::value_type and iterator_traits\texttt{<InputIteratorW>}::value_type shall each denote a type that is convertible to \texttt{double}. If firstB == lastB or ++firstB == lastB, let \( n = 1, w_0 = 1, b_0 = 0, \) and \( b_1 = 1. \) Otherwise, \([\texttt{firstB}, \texttt{lastB}]\) shall form a sequence \( b \) of length \( n + 1. \) the length of the sequence \( w \) starting from \( \texttt{firstW} \) shall be at least \( n, \) and any \( w_k \) for \( k \geq n \) shall be ignored by the distribution.

Effects: Constructs a \texttt{piecewise} \texttt{constant} \texttt{distribution} object with parameters as specified above.

template<class UnaryOperation>
piecewise_constant_distribution(initializer_list<RealType> bl, UnaryOperation fw);

Requires: Each instance of type \texttt{UnaryOperation} shall be a function object (20.8) whose return type shall be convertible to \texttt{double}. Moreover, \texttt{double} shall be convertible to the type of \texttt{UnaryOperation}'s sole parameter.

Effects: Constructs a \texttt{piecewise} \texttt{constant} \texttt{distribution} object with parameters taken or calculated from the following values: If \texttt{bl.size()} < 2, let \( n = 1, w_0 = 1, b_0 = 0, \) and \( b_1 = 1. \) Otherwise, let \([\texttt{bl.begin()}, \texttt{bl.end()}]\) form a sequence \( b_0, \ldots, b_n, \) and let \( w_k = \texttt{fw}\left(\frac{(b_{k+1}+b_k)}{2}\right) \) for \( k = 0, \ldots, n-1. \)

Complexity: The number of invocations of \texttt{fw} shall not exceed \( n. \)

template<class UnaryOperation>
piecewise_constant_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);

Requires: Each instance of type \texttt{UnaryOperation} shall be a function object (20.8) whose return type shall be convertible to \texttt{double}. Moreover, \texttt{double} shall be convertible to the type of \texttt{UnaryOperation}'s sole parameter. If \( \texttt{nw} = 0, \) let \( n = 1, \) otherwise let \( n = \texttt{nw}. \) The relation \( 0 < \delta = \texttt{(xmax} - \texttt{xmin})/n \) shall hold.
Effects: Constructs a piecewise_constant_distribution object with parameters taken or calculated from the following values: Let $b_k = x_{\text{min}} + k \cdot \delta$ for $k = 0, \ldots, n$, and $w_k = f_w(b_k + \delta/2)$ for $k = 0, \ldots, n-1$.

Complexity: The number of invocations of $f_w$ shall not exceed $n$.

```
vector<result_type> intervals() const;
```

Returns: A vector<result_type> whose size member returns $n+1$ and whose operator[] member returns $b_k$ when invoked with argument $k$ for $k = 0, \ldots, n$.

```
vector<double> densities() const;
```

Returns: A vector<result_type> whose size member returns $n$ and whose operator[] member returns $\rho_k$ when invoked with argument $k$ for $k = 0, \ldots, n-1$.

### 26.5.8.5.3 Class template piecewise_linear_distribution [rand.dist.samp.plinear]

A piecewise_linear_distribution random number distribution produces random numbers $x$, $b_0 \leq x < b_n$, distributed over each subinterval $[b_i, b_{i+1})$ according to the probability density function

$$p(x \mid b_0, \ldots, b_n, \rho_0, \ldots, \rho_n) = \rho_i \cdot \frac{b_{i+1} - x}{b_{i+1} - b_i} + \rho_{i+1} \cdot \frac{x - b_i}{b_{i+1} - b_i}, \text{ for } b_i \leq x < b_{i+1}.$$  

The $n+1$ distribution parameters $b_i$, also known as this distribution’s interval boundaries, shall satisfy the relation $b_i < b_{i+1}$ for $i = 0, \ldots, n-1$. Unless specified otherwise, the remaining $n+1$ distribution parameters are calculated as $\rho_k = w_k / S$ for $k = 0, \ldots, n$, in which the values $w_k$, commonly known as the weights at boundaries, shall be non-negative, non-\text{NaN}, and non-infinity. Moreover, the following relation shall hold:

$$0 < S = \frac{1}{2} \sum_{k=0}^{n-1} (w_k + w_{k+1}) \cdot (b_{k+1} - b_k).$$

```cpp
#include <functional>

template<class RealType = double>
class piecewise_linear_distribution
{
  public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    piecewise_linear_distribution();
    template<class InputIteratorB, class InputIteratorW>
    piecewise_linear_distribution(InputIteratorB firstB, InputIteratorB lastB,
                                  InputIteratorW firstW);
    template<class UnaryOperation>
    piecewise_linear_distribution(initializer_list<RealType> bl, UnaryOperation fw);
    template<class UnaryOperation>
    piecewise_linear_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);
    explicit piecewise_linear_distribution(const param_type& parm);
    void reset();

    // generating functions

    § 26.5.8.5.3
```
template<class URNG>
    result_type operator()(URNG& g);

template<class URNG>
    result_type operator()(URNG& g, const param_type& parm);

// property functions
vector<result_type> intervals() const;
vector<double> densities() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;

};

piecewise_linear_distribution();

Effects: Constructs a piecewise_linear_distribution object with n = 1, ρ₀ = ρ₁ = 1, b₀ = 0, and b₁ = 1.

template<class InputIteratorB, class InputIteratorW>
piecewise_linear_distribution(InputIteratorB firstB, InputIteratorB lastB,
                              InputIteratorW firstW);

Requires: InputIteratorB and InputIteratorW shall each satisfy the requirements of an input iterator (Table 104) type. Moreover, iterator_traits<InputIteratorB>::value_type and iterator_traits<InputIteratorW>::value_type shall each denote a type that is convertible to double. If firstB == lastB or ++firstB == lastB, let n = 1, ρ₀ = ρ₁ = 1, b₀ = 0, and b₁ = 1. Otherwise, [firstB, lastB) shall form a sequence b of length n + 1, the length of the sequence w starting from firstW shall be at least n + 1, and any wk for k ≥ n + 1 shall be ignored by the distribution.

Effects: Constructs a piecewise_linear_distribution object with parameters as specified above.

template<class UnaryOperation>
piecewise_linear_distribution(initializer_list<RealType> bl, UnaryOperation fw);

Requires: Each instance of type UnaryOperation shall be a function object (20.8) whose return type shall be convertible to double. Moreover, double shall be convertible to the type of UnaryOperation’s sole parameter.

Effects: Constructs a piecewise_linear_distribution object with parameters taken or calculated from the following values: If bl.size() < 2, let n = 1, ρ₀ = ρ₁ = 1, b₀ = 0, and b₁ = 1. Otherwise, let [bl.begin(), bl.end()) form a sequence b₀, ..., bₙ, and let wk = fw(bₙ) for k = 0, ..., n.

Complexity: The number of invocations of fw shall not exceed n + 1.

template<class UnaryOperation>
piecewise_linear_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);

Requires: Each instance of type UnaryOperation shall be a function object (20.8) whose return type shall be convertible to double. Moreover, double shall be convertible to the type of UnaryOperation’s sole parameter. If nw = 0, let n = 1, otherwise let n = nw. The relation 0 < δ = (xmax – xmin)/n shall hold.

§ 26.5.8.5.3
Effects: Constructs a `piecewise_linear_distribution` object with parameters taken or calculated from the following values: Let \( b_k = \text{xmin} + k \cdot \delta \) for \( k = 0, \ldots, n \), and \( w_k = f(w + \delta) \) for \( k = 0, \ldots, n \).

Complexity: The number of invocations of \( f \) shall not exceed \( n + 1 \).

```cpp
vector<result_type> intervals() const;
```

Returns: A `vector<result_type>` whose `size` member returns \( n + 1 \) and whose `operator[]` member returns \( b_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n \).

```cpp
vector<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 \( k = 0, \ldots, n \).
template<class T> valarray<T> operator^ (const valarray<T>&, const T&);
template<class T> valarray<T> operator^ (const T&, const valarray<T>&);

template<class T> valarray<T> operator& (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator& (const valarray<T>&, const T&);
template<class T> valarray<T> operator& (const T&, const valarray<T>&);

template<class T> valarray<T> operator| (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator| (const valarray<T>&, const T&);
template<class T> valarray<T> operator| (const T&, const valarray<T>&);

template<class T> valarray<T> operator<< (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator<< (const valarray<T>&, const T&);
template<class T> valarray<T> operator<< (const T&, const valarray<T>&);

template<class T> valarray<T> operator>> (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator>> (const valarray<T>&, const T&);
template<class T> valarray<T> operator>>() (const 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 T&);
template<class T> valarray<bool> operator&&(const T&, const valarray<T>&);

template<class T> valarray<bool> operator||(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator||(const valarray<T>&, const T&);
template<class T> valarray<bool> operator||(const T&, const valarray<T>&);

template<class T> valarray<bool> operator==(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator==(const valarray<T>&, const T&);
template<class T> valarray<bool> operator==(const T&, const valarray<T>&);

template<class T> valarray<bool> operator!=(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator!=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator!=(const T&, const valarray<T>&);

template<class T> valarray<bool> operator< (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator< (const valarray<T>&, const T&);
template<class T> valarray<bool> operator< (const T&, const valarray<T>&);

template<class T> valarray<bool> operator> (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator> (const valarray<T>&, const T&);
template<class T> valarray<bool> operator> (const T&, const valarray<T>&);

template<class T> valarray<bool> operator<=(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator<=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator<=(const T&, const valarray<T>&);

template<class T> valarray<bool> operator>=(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator>=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator>=(const T&, const valarray<T>&);

valarray<bool> operator==(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator==(const valarray<T>&, const T&);
template<class T> valarray<bool> operator==(const T&, const valarray<T>&);

valarray<bool> operator!=(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator!=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator!=(const T&, const valarray<T>&);

valarray<bool> operator< (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator< (const valarray<T>&, const T&);
template<class T> valarray<bool> operator< (const T&, const valarray<T>&);

valarray<bool> operator> (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator> (const valarray<T>&, const T&);
template<class T> valarray<bool> operator> (const T&, const valarray<T>&);

valarray<bool> operator<=(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator<=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator<=(const T&, const valarray<T>&);

valarray<bool> operator>=(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator>=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator>=(const T&, const valarray<T>&);

valarray<T> abs (const valarray<T>&);
template<class T> valarray<T> acos (const valarray<T>&);
template<class T> valarray<T> asin (const valarray<T>&);
template<class T> valarray<T> atan (const valarray<T>&);

template<class T> valarray<T> atan2(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> atan2(const valarray<T>&, const T&);
template<class T> valarray<T> atan2(const T&, const valarray<T>&);

template<class T> valarray<T> cos (const valarray<T>&);
template<class T> valarray<T> cosh (const valarray<T>&);

template<class T> valarray<T> exp (const valarray<T>&);

template<class T> valarray<T> log (const valarray<T>&);

template<class T> valarray<T> log10(const valarray<T>&);

template<class T> valarray<T> pow(const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> pow(const valarray<T>&, const T&);

template<class T> valarray<T> pow(const T&, const valarray<T>&);

template<class T> valarray<T> sin (const valarray<T>&);

template<class T> valarray<T> sinh (const valarray<T>&);

template<class T> valarray<T> sqrt (const valarray<T>&);

template<class T> valarray<T> tan (const valarray<T>&);

template<class T> valarray<T> tanh (const valarray<T>&);

template <class T> unspecified1 begin(valarray<T>& v);

template <class T> unspecified2 begin(const valarray<T>& v);

template <class T> unspecified1 end(valarray<T>& v);

template <class T> unspecified2 end(const valarray<T>& v);

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.\(^{285}\)

Implementations introducing such replacement types shall provide additional functions and operators as follows:

- for every function taking a `const valarray<T>&`, identical functions taking the replacement types shall be added;

- for every function taking two `const valarray<T>&` arguments, identical functions taking every combination of `const valarray<T>&` and replacement types shall be added.

In particular, an implementation shall allow a `valarray<T>` to be constructed from such replacement types and shall allow assignments and computed assignments of such types to `valarray<T>`, `slice_array<T>`, `gslice_array<T>`, `mask_array<T>` and `indirect_array<T>` objects.

\(^{285}\) Clause 18.3.1 recommends a minimum number of recursively nested template instantiations. This requirement thus indirectly suggests a minimum allowable complexity for valarray expressions.
These library functions are permitted to throw a bad_alloc (18.6.2.1) exception if there are not sufficient resources available to carry out the operation. Note that the exception is not mandated.

26.6.2 Class template valarray

```cpp
namespace std {
    template<class T> class valarray {
    public:
        typedef T value_type;

        // 26.6.2.1 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(initializer_list<T>);
        ~valarray();

        // 26.6.2.2 assignment:
        valarray<T>& operator=(const valarray<T>&);
        valarray<T>& operator=(valarray<T>&&);
        valarray<T>& operator=(initializer_list<T>);
        valarray<T>& operator=(const T&);
        valarray<T>& operator=(const slice_array<T>&);
        valarray<T>& operator=(const gslice_array<T>&);
        valarray<T>& operator=(const mask_array<T>&);
        valarray<T>& operator=(const indirect_array<T>&);

        // 26.6.2.3 element access:
        const T& operator[](size_t) const;
        T& operator[](size_t);

        // 26.6.2.4 subset operations:
        valarray<T> operator[](slice) const;
        slice_array<T> operator[](slice);
        valarray<T> operator[](const gslice&) const;
        gslice_array<T> operator[](const gslice&);
        valarray<T> operator[](const valarray<bool>&) const;
        mask_array<T> operator[](const valarray<bool>&);
        valarray<T> operator[](const valarray<br> size_t&) const;
        indirect_array<T> operator[](const valarray<br> size_t&);

        // 26.6.2.5 unary operators:
        valarray<T> operator+() const;
        valarray<T> operator-() const;
        valarray<T> operator~() const;
        valarray<bool> operator!() const;

        // 26.6.2.6 computed assignment:
    }
```
The class template `valarray<T>` is a one-dimensional smart array, with elements numbered sequentially from zero. It is a representation of the mathematical concept of an ordered set of values. The illusion of higher dimensionality may be produced by the familiar idiom of computed indices, together with the powerful subsetting capabilities provided by the generalized subscript operators.

An implementation is permitted to qualify any of the functions declared in `<valarray>` as `inline`.

### 26.6.2.1 valarray constructors

```cpp
valarray();
```
Effects: Constructs an object of class `valarray<T>` which has zero length.

```cpp
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 value-initialized (8.5).

```cpp
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.

```cpp
valarray(const T*, size_t);
```

The array created by this constructor has a length equal to the second argument \( n \). The values of the elements of the array are initialized with the first \( n \) values pointed to by the first argument. If the value of the second argument is greater than the number of values pointed to by the first argument, the behavior is undefined.

```cpp
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.

```cpp
valarray(valarray<T>&& v);
```

The array created by this constructor has the same length as the argument array. The elements are initialized with the values of the corresponding elements of the argument array. After construction, \( v \) is in a valid but unspecified state.

Complexity: Constant.

Throws: Nothing.

```cpp
valarray(initializer_list<T> il);
```

Effects: Same as `valarray(il.begin(), il.size())`.

```cpp
valarray(const slice_array<T>&);
valarray(const gslice_array<T>&);
valarray(const mask_array<T>&);
valarray(const indirect_array<T>&);
```

These conversion constructors convert one of the four reference templates to a `valarray`.

```cpp
~valarray();
```

The destructor is applied to every element of `*this`; an implementation may return all allocated memory.

---

287) For convenience, such objects are referred to as “arrays” throughout the remainder of 26.6.
288) This default constructor is essential, since arrays of `valarray` may be useful. After initialization, the length of an empty array can be increased with the `resize` member function.
289) This constructor is the preferred method for converting a C array to a `valarray` object.
290) This copy constructor creates a distinct array rather than an alias. Implementations in which arrays share storage are permitted, but they shall implement a copy-on-reference mechanism to ensure that arrays are conceptually distinct.
26.6.2.2 valarray assignment

valarray\langle T\rangle & \text{operator}=(\text{const} \ valarray\langle T\rangle & v);

1 Each element of the \*this array is assigned the value of the corresponding element of the argument array. If the length of v is not equal to the length of \*this, resizes \*this to make the two arrays the same length, as if by calling resize(v.size()), before performing the assignment.

2 \textit{Postcondition: size()} == v.size().

valarray\langle T\rangle & \text{operator}=(\text{valarray}\langle T\rangle &\& v);

3 \textit{Effects:} \*this obtains the value of v. After the assignment, v is in a valid but unspecified state. If the length of v is not equal to the length of \*this, resizes \*this to make the two arrays the same length, as if by calling resize(v.size()), before performing the assignment.

4 \textit{Complexity:} Constant.

5 \textit{Throws:} Nothing.

valarray\& \text{operator}=(\text{initializer_list}\langle T\rangle \ il);

6 \textit{Effects:} \*this = valarray(il).

7 \textit{Returns:} \*this.

valarray\langle T\rangle & \text{operator}=(\text{const} \ T\&);

8 The scalar assignment operator causes each element of the \*this array to be assigned the value of the argument.

valarray\langle T\rangle & \text{operator}=(\text{const} \ slice\_array\langle T\rangle &);
valarray\langle T\rangle & \text{operator}=(\text{const} \ gslice\_array\langle T\rangle &);
valarray\langle T\rangle & \text{operator}=(\text{const} \ mask\_array\langle T\rangle &);
valarray\langle T\rangle & \text{operator}=(\text{const} \ indirect\_array\langle T\rangle &);

9 \textit{Requires:} The length of the array to which the argument refers equals size().

10 These operators allow the results of a generalized subscripting operation to be assigned directly to a valarray.

11 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.

26.6.2.3 valarray element access

const T\& \text{operator}[] (size_t) const;
T\& \text{operator}[] (size_t);

1 The subscript operator returns a reference to the corresponding element of the array.

2 Thus, the expression (a[i] = q, a[i]) == q evaluates as true for any non-constant valarray\langle T\rangle a, any T q, and for any size_t i such that the value of i is less than the length of a.

3 The expression \&a[i+j] == \&a[i] + j evaluates as true for all size_t i and size_t j such that i+j is less than the length of the array a.
Likewise, the expression $\&a[i] \neq \&b[j]$ evaluates as true for any two arrays $a$ and $b$ and for any size_t $i$ and size_t $j$ such that $i$ is less than the length of $a$ and $j$ is less than the length of $b$. This property indicates an absence of aliasing and may be used to advantage by optimizing compilers.  

The reference returned by the subscript operator for an array is guaranteed to be valid until the member function `resize(size_t, T)` (26.6.2.7) is called for that array or until the lifetime of that array ends, whichever happens first.  

If the subscript operator is invoked with a size_t argument whose value is not less than the length of the array, the behavior is undefined.  

### 26.6.2.4 valarray subset operations

The member `operator[](slice slicearr)` is overloaded to provide several ways to select sequences of elements from among those controlled by `*this`. Each of these operations returns a subset of the array. The const-qualified versions return this subset as a new valarray object. The non-const versions return a class template object which has reference semantics to the original array, working in conjunction with various overloads of `operator=` and other assigning operators to allow selective replacement (slicing) of the controlled sequence. In each case the selected element(s) must exist.

```c
valarray<T> operator[](slice slicearr) const;
```

Returns: an object of class valarray<T> containing those elements of the controlled sequence designated by `slicearr`.  

[Example:]  
```c
const valarray<char> v0("abcdefghijklmnop", 16);
// v0[slice(2, 5, 3)] returns valarray<char>("cfilo", 5)
```

— end example

```c
slice_array<T> operator[](slice);
```

Returns: an object that holds references to elements of the controlled sequence selected by `slicearr`.  

[Example:]  
```c
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDE", 5);
v0[slice(2, 5, 3)] = v1;
// v0 == valarray<char>("abAdBghCjkDmnEp", 16);
```

— end example

```c
valarray<T> operator[](const gslice& gslicearr) const;
```

Returns: an object of class valarray<T> containing those elements of the controlled sequence designated by `gslicearr`.  

[Example:]  
```c
const valarray<char> v0("abcdefghijklmnop", 16);
const size_t lv[] = { 2, 3 };
const size_t dv[] = { 7, 2 };  
const valarray<size_t> len(lv, 2), str(dv, 2);
// v0[gslice(3, len, str)] returns
// valarray<char>("dfhkmno", 6)
```

291) Compilers may take advantage of inlining, constant propagation, loop fusion, tracking of pointers obtained from `operator new`, and other techniques to generate efficient valarrays.
gslice_array<T> operator[](const gslice& gslice_array);

Returns: an object that holds references to elements of the controlled sequence selected by gslicearr.

[Example:

valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDE", 5);
const size_t lv[] = { 2, 3 };
const size_t dv[] = { 7, 2 };
const valarray<size_t> len(lv, 2), str(dv, 2);
v0[gslice(3, len, str)] = v1;
// v0 == valarray<char>("abcAeBgCijDlEnFp", 16)

— end example]

valarray<T> operator[](const valarray<bool>& boolarr) const;

Returns: an object of class valarray<T> containing those elements of the controlled sequence designated by boolarr.

[Example:

cst valarray<char> v0("abcdefghijklmnop", 16);
cst bool vb[] = { false, false, true, true, false, true };
// v0[valarray<bool>(vb, 6)] returns
// valarray<char>("cdf", 3)

— end example]

mask_array<T> operator[](const valarray<bool>& boolarr);

Returns: an object that holds references to elements of the controlled sequence selected by boolarr.

[Example:

cst valarray<char> v0("abcdefghijklmnop", 16);
cst char v1("ABC", 3);
cst bool vb[] = { false, false, true, true, false, true };
v0[valarray<bool>(vb, 6)] = v1;
// v0 == valarray<char>("abABeGhijklmnop", 16)

— end example]

valarray<T> operator[](const valarray<size_t>& indarr) const;

Returns: an object of class valarray<T> containing those elements of the controlled sequence designated by indarr.

[Example:

cst valarray<char> v0("abcdefghijklmnop", 16);
cst size_t vi[] = { 7, 5, 2, 3, 8 };
// v0[valarray<size_t>(vi, 5)] returns
// valarray<char>("hfcdi", 5)

— end example]

indirect_array<T> operator[](const valarray<size_t>& indarr);

Returns: an object that holds references to elements of the controlled sequence selected by indarr.

[Example:
```cpp
valarray<char> v0("abcdefghijklmnopqrstuvwxyz", 16);
valarray<char> v1("ABCDEFGHIJKLMNOPQRSTUVWXYZ", 5);
const size_t vi[] = {7, 5, 2, 3, 8};
v0[valarray<size_t>(vi, 5)] = v1;
// v0 == valarray<char>("abCDeBgAEjklmnop", 16)
```

--- end example ---

### 26.6.2.5 valarray unary operators

```cpp
valarray<T> operator+() const;
valarray<T> operator-() const;
valarray<T> operator~() const;
valarray<bool> operator!() const;
```

1. Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `T` (bool for `operator!`) or which may be unambiguously implicitly converted to type `T` (bool for `operator!`).

2. Each of these operators returns an array whose length is equal to the length of the array. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array.

### 26.6.2.6 valarray computed assignment

```cpp
valarray<T>& operator*= (const valarray<T>&);
valarray<T>& operator/= (const valarray<T>&);
valarray<T>& operator%= (const valarray<T>&);
valarray<T>& operator+= (const valarray<T>&);
valarray<T>& operator-= (const valarray<T>&);
valarray<T>& operator^= (const valarray<T>&);
valarray<T>& operator&= (const valarray<T>&);
valarray<T>& operator|= (const valarray<T>&);
valarray<T>& operator<<= (const valarray<T>&);
valarray<T>& operator>>= (const valarray<T>&);
```

1. 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.

2. The array is then returned by reference.

3. 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.

4. If the value of an element in the left-hand side of a valarray computed assignment operator depends on the value of another element in that left hand side, the resulting behavior is undefined.

```cpp
valarray<T>& operator*= (const T&);
valarray<T>& operator/= (const T&);
valarray<T>& operator%= (const T&);
valarray<T>& operator+= (const T&);
valarray<T>& operator-= (const T&);
valarray<T>& operator^= (const T&);
valarray<T>& operator&= (const T&);
valarray<T>& operator|= (const T&);
valarray<T>& operator<<= (const T&);
valarray<T>& operator>>= (const T&);
```

§ 26.6.2.6
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.

26.6.2.7 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;

Returns: the number of elements in the array.
Complexity: constant time.
T sum() const;

This function may only be instantiated for a type T to which operator+= can be applied. This function returns the sum of all the elements of the array.
If the array has length 0, the behavior is undefined. If the array has length 1, sum() returns the value of element 0. Otherwise, the returned value is calculated by applying operator+= to a copy of an element of the array and all other elements of the array in an unspecified order.

T min() const;

This function returns the minimum value contained in *this. The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using operator<.

T max() const;

This function returns the maximum value contained in *this. The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using operator<.

valarray<T> shift(int n) const;

This function returns an object of class valarray<T> of length size(), each of whose elements I is (*this)[I + n] if I + n is non-negative and less than size(), otherwise T(). Thus if element zero is taken as the leftmost element, a positive value of n shifts the elements left n places, with zero fill.

§ 26.6.2.7
[Example: If the argument has the value -2, the first two elements of the result will be value-initialized (8.5); the third element of the result will be assigned the value of the first element of the argument; etc. — end example]

valarray<T> cshift(int n) const;

This function returns an object of class valarray<T> of length size() that is a circular shift of *this. If element zero is taken as the leftmost element, a non-negative value of n shifts the elements circularly left n places and a negative value of n shifts the elements circularly right -n places.

valarray<T> apply(T func(T)) const;
valarray<T> apply(T func(const T&)) const;

These functions return an array whose length is equal to the array. Each element of the returned array is assigned the value returned by applying the argument function to the corresponding element of the array.

void resize(size_t sz, T c = T());

This member function changes the length of the *this array to sz and then assigns to each element the value of the second argument. Resizing invalidates all pointers and references to elements in the array.

26.6.3 valarray non-member operations

26.6.3.1 valarray binary operators

template<class T> valarray<T> operator*(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator/(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator%(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator+(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator-(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator^(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator&(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator|(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator<<(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator>>(const valarray<T>&, const valarray<T>&);

Each of these operators may only be instantiated for a type T to which the indicated operator can be applied and for which the indicated operator returns a value which is of type T or which can be unambiguously 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.

§ 26.6.3.1
If the argument arrays do not have the same length, the behavior is undefined.

```cpp
template<class T> valarray<T> operator* (const valarray<T>&, const T&);
template<class T> valarray<T> operator* (const T&, const valarray<T>&);
template<class T> valarray<T> operator/ (const valarray<T>&, const T&);
template<class T> valarray<T> operator/ (const T&, const valarray<T>&);
template<class T> valarray<T> operator% (const valarray<T>&, const T&);
template<class T> valarray<T> operator% (const T&, const valarray<T>&);
template<class T> valarray<T> operator+ (const valarray<T>&, const T&);
template<class T> valarray<T> operator+ (const T&, const valarray<T>&);
template<class T> valarray<T> operator- (const valarray<T>&, const T&);
template<class T> valarray<T> operator- (const T&, const valarray<T>&);
template<class T> valarray<T> operator^ (const valarray<T>&, const T&);
template<class T> valarray<T> operator^ (const T&, const valarray<T>&);
template<class T> valarray<T> operator& (const valarray<T>&, const T&);
template<class T> valarray<T> operator& (const T&, const valarray<T>&);
template<class T> valarray<T> operator| (const valarray<T>&, const T&);
template<class T> valarray<T> operator| (const T&, const valarray<T>&);
template<class T> valarray<T> operator<< (const valarray<T>&, const T&);
template<class T> valarray<T> operator<< (const T&, const valarray<T>&);
template<class T> valarray<T> operator>> (const valarray<T>&, const T&);
template<class T> valarray<T> operator>> (const T&, const valarray<T>&);
```

Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `T` or which can be unambiguously 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.

### 26.6.3.2 `valarray` logical operators

```cpp
template<class T> valarray<bool> operator== (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator!= (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator< (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator> (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator<= (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator>= (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator&& (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator|| (const valarray<T>&, const valarray<T>&);
```

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`.

§ 26.6.3.2

954
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.

```cpp
template<class T> valarray<bool> operator==(const valarray<T>&, const T&);
template<class T> valarray<bool> operator==(const T&, const valarray<T>&);
template<class T> valarray<bool> operator!=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator!=(const T&, const valarray<T>&);
template<class T> valarray<bool> operator>(const valarray<T>&, const T&);
template<class T> valarray<bool> operator>(const T&, const valarray<T>&);
```

Each of these operators may only be instantiated for a type T to which the indicated operator can be applied and for which the indicated operator returns a value which is of type bool or which can be unambiguously 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.

### 26.6.3.3 valarray transcendentals

```cpp
template<class T> valarray<T> abs (const valarray<T>&);
template<class T> valarray<T> acos (const valarray<T>&);
template<class T> valarray<T> asin (const valarray<T>&);
template<class T> valarray<T> atan (const valarray<T>&);
```
Each of these functions may only be instantiated for a type T to which a unique function with the indicated name can be applied (unqualified). This function shall return a value which is of type T or which can be unambiguously implicitly converted to type T.

26.6.3.4 valarray specialized algorithms

```
template <class T> void swap(valarray<T>& x, valarray<T>& y);
```

Effects: \( x \text{.} swap(y) \).

26.6.4 Class slice

```
namespace std {

class slice {
  public:
    slice();
    slice(size_t, size_t, size_t);

    size_t start() const;
    size_t size() const;
    size_t stride() const;
  }
};
```

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.\(^{(292)}\)

26.6.4.1 slice constructors

```
slice();
slice(size_t start, size_t length, size_t stride);
slice(const slice&);
```

The default constructor is equivalent to `slice(0, 0, 0)`. A default constructor is provided only to permit the 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]
```

26.6.4.2 slice access functions

```
size_t start() const;
size_t size() const;
size_t stride() const;
```

Returns: the start, length, or stride specified by a `slice` object.

Complexity: constant time.

\(^{(292)}\) BLAS stands for Basic Linear Algebra Subprograms. C++ programs may instantiate this class. See, for example, Dongarra, Du Croz, Duff, and Hammerling: A set of Level 3 Basic Linear Algebra Subprograms; Technical Report MCS-P1-0888, Argonne National Laboratory (USA), Mathematics and Computer Science Division, August, 1988.
26.6.5 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]

26.6.5.1 slice_array assignment

These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the valarray<T> object to which the slice_array object refers.

26.6.5.2 slice_array computed assignment
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.

26.6.5.3 slice_array fill function

void operator=(const T&) const;

This function has reference semantics, assigning the value of its argument to the elements of the valarray<T> object to which the slice_array object refers.

26.6.6 The gslice class

namespace std {
  class gslice {
    public:
      gslice();
      gslice(size_t s, const valarray<size_t>& l, const valarray<size_t>& d);
      size_t start() const;
      valarray<size_t> size() const;
      valarray<size_t> stride() const;
    };
  }

This class represents a generalized slice out of an array. A gslice is defined by a starting offset \( s \), a set of lengths \( l_j \), and a set of strides \( d_j \). The number of lengths shall equal the number of strides.

A gslice represents a mapping from a set of indices \( i_j \), equal in number to the number of strides, to a single index \( k \). It is useful for building multidimensional array classes using the valarray template, which is one-dimensional. The set of one-dimensional index values specified by a gslice are

\[
k = s + \sum_j i_j d_j
\]

where the multidimensional indices \( i_j \) range in value from 0 to \( l_{ij} - 1 \).

[Example: The gslice specification]

\[
\text{start} = 3 \\
\text{length} = \{2, 4, 3\} \\
\text{stride} = \{19, 4, 1\}
\]

yields the sequence of one-dimensional indices

\[
k = 3 + (0, 1) \times 19 + (0, 1, 2, 3) \times 4 + (0, 1, 2) \times 1
\]

which are ordered as shown in the following table:

\[
\begin{array}{cccc}
  (i_0, & i_1, & i_2, & k) = \\
  (0, & 0, & 0, & 3), \\
  (0, & 0, & 1, & 4), \\
  (0, & 0, & 2, & 5),
\end{array}
\]
(0, 1, 0, 7),
(0, 1, 1, 8),
(0, 1, 2, 9),
(0, 2, 0, 11),
(0, 2, 1, 12),
(0, 2, 2, 13),
(0, 3, 0, 15),
(0, 3, 1, 16),
(0, 3, 2, 17),
(1, 0, 0, 22),
(1, 0, 1, 23),
...  
(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),
...

— 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.

26.6.6.1 gslice constructors

[gslice.cons]

gslice();
gslice(size_t start, const valarray<size_t>& lengths,
    const valarray<size_t>& strides);
gslice(const gslice&);

1 The default constructor is equivalent to gslice(0, valarray<size_t>(), valarray<size_t>().
The constructor with arguments builds a gslice based on a specification of start, lengths, and strides, as explained in the previous section.

26.6.6.2 gslice access functions

[gslice.access]

size_t start() const;
valarray<size_t> size() const;
valarray<size_t> stride() const;

1 Returns: the representation of the start, lengths, or strides specified for the gslice.

2 Complexity: start() is constant time. size() and stride() are linear in the number of strides.
26.6.7 Class template gslice_array

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

gslice_array<T> valarray<T>::operator[](const gslice&);

It has reference semantics to a subset of an array specified by a gslice object.

Thus, the expression a[gslice(1, length, stride)] = b has the effect of assigning the elements of b to a generalized slice of the elements in a.

26.6.7.1 gslice_array assignment

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 refers.

26.6.7.2 gslice_array

void operator**(const valarray<T>&) const;
void operator%/=(const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;

§ 26.6.7.2
These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `gslice_array` object refers.

### 26.6.7.3 `gslice_array` fill function

This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `gslice_array` object refers.

### 26.6.8 Class template `mask_array`

This template is a helper template used by the mask subscript operator:

```cpp
mask_array<T> valarray<T>::operator[](const valarray<bool>&).
```

It has reference semantics to a subset of an array specified by a boolean mask. Thus, the expression `a[mask] = b;` has the effect of assigning the elements of `b` to the masked elements in `a` (those for which the corresponding element in `mask` is `true`.)

#### 26.6.8.1 `mask_array` assignment

```cpp
void operator=(const valarray<T>&) const;
const mask_array& operator=(const mask_array&) const;
```
These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which it refers.

### 26.6.8.2 mask_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 mask object refers.

### 26.6.8.3 mask_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 `mask_array` object refers.

### 26.6.9 Class template indirect_array

```c
namespace std {
    template <class T> class indirect_array {
        public:
            typedef T value_type;

            void operator= (const valarray<T>&) const;
            void operator*= (const valarray<T>&) const;
            void operator/= (const valarray<T>&) const;
            void operator%= (const valarray<T>&) const;
            void operator+= (const valarray<T>&) const;
            void operator-= (const valarray<T>&) const;
            void operator^= (const valarray<T>&) const;
            void operator&= (const valarray<T>&) const;
            void operator|= (const valarray<T>&) const;
            void operator<<=(const valarray<T>&) const;
            void operator>>=(const valarray<T>&) const;

            indirect_array(const indirect_array&);
            ~indirect_array();
            const indirect_array& operator=(const indirect_array&) const;
            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

\[
\text{indirect\_array}\langle T\rangle \text{ valarray}\langle T\rangle::\text{operator[]}\langle\text{const valarray}\langle\text{size\_t}\rangle&\rangle.
\]

It has reference semantics to a subset of an array specified by an \text{indirect\_array}. Thus the expression \text{a}[\text{indirect}] = \text{b}; has the effect of assigning the elements of \text{b} to the elements in \text{a} whose indices appear in \text{indirect}.

26.6.9.1 \text{indirect\_array} assignment

\begin{verbatim}
void operator=(\text{const valarray}\langle T\rangle&) const;
const indirect\_array\& operator=(\text{const indirect\_array\&}) const;
\end{verbatim}

1 These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the \text{valarray}\langle T\rangle object to which it refers.

2 If the \text{indirect\_array} specifies an element in the \text{valarray}\langle T\rangle object to which it refers more than once, the behavior is undefined.

[Example:

\begin{verbatim}
int addr[] = \{2, 3, 1, 4, 4\};
valarray\langle\text{size\_t}\rangle indirect(addr, 5);
valarray\langle\text{double}\rangle a(0., 10), b(1., 5);
a[\text{indirect}] = b;
\end{verbatim}

results in undefined behavior since element 4 is specified twice in the indirection. — end example]

26.6.9.2 \text{indirect\_array} computed assignment

\begin{verbatim}
void operator*= (\text{const valarray}\langle T\rangle&) const;
void operator/= (\text{const valarray}\langle T\rangle&) const;
void operator%= (\text{const valarray}\langle T\rangle&) const;
void operator+= (\text{const valarray}\langle T\rangle&) const;
void operator-= (\text{const valarray}\langle T\rangle&) const;
void operator^= (\text{const valarray}\langle T\rangle&) const;
void operator&= (\text{const valarray}\langle T\rangle&) const;
void operator|= (\text{const valarray}\langle T\rangle&) const;
void operator<<=(\text{const valarray}\langle T\rangle&) const;
void operator>>=(\text{const valarray}\langle T\rangle&) const;
\end{verbatim}

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the \text{valarray}\langle T\rangle object to which the \text{indirect\_array} object refers.

2 If the \text{indirect\_array} specifies an element in the \text{valarray}\langle T\rangle object to which it refers more than once, the behavior is undefined.

26.6.9.3 \text{indirect\_array} fill function

\begin{verbatim}
void operator=(\text{const T\&}) const;
\end{verbatim}

1 This function has reference semantics, assigning the value of its argument to the elements of the \text{valarray}\langle T\rangle object to which the \text{indirect\_array} object refers.
26.6.10  valarray range access

In the begin and end function templates that follow, unspecified1 is a type that meets the requirements of a mutable random access iterator (24.2.7) whose value_type is the template parameter T and whose reference type is T&. unspecified2 is a type that meets the requirements of a constant random access iterator (24.2.7) whose value_type is the template parameter T and whose reference type is const T&.

template <class T> unspecified1 begin(valarray<T>& v);
template <class T> unspecified2 begin(const valarray<T>& v);

Returns: an iterator referencing the first value in the numeric array.

template <class T> unspecified1 end(valarray<T>& v);
template <class T> unspecified2 end(const valarray<T>& v);

Returns: an iterator referencing one past the last value in the numeric array.

26.7  Generalized numeric operations

Header <numeric> synopsis

namespace std {
    template <class InputIterator, class T>
    T accumulate(InputIterator first, InputIterator last, T init);
template <class InputIterator, class T, class BinaryOperation>
    T accumulate(InputIterator first, InputIterator last, T init,
                  BinaryOperation binary_op);

template <class InputIterator1, class InputIterator2, class T>
    T inner_product(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, T init);

template <class InputIterator1, class InputIterator2, class T, class BinaryOperation1, class BinaryOperation2>
    T inner_product(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, T init,
                     BinaryOperation1 binary_op1,
                     BinaryOperation2 binary_op2);

template <class InputIterator, class OutputIterator>
    OutputIterator partial_sum(InputIterator first, InputIterator last,
                                OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
    OutputIterator partial_sum(InputIterator first, InputIterator last,
                                OutputIterator result,
                                BinaryOperation binary_op);

template <class InputIterator, class OutputIterator>
    OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                          OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
    OutputIterator adjacent_difference(InputIterator first,
```cpp
template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init);

template <class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init,
             BinaryOperation binary_op);
```

**Effects:** Computes its result by initializing the accumulator acc with the initial value init and then modifies it with acc = acc + *i or acc = binary_op(acc, *i) for every iterator i in the range [first,last) in order.\(^{293}\)

**Requires:** T shall meet the requirements of CopyConstructible (35) and Assignable (23.2) types. In the range [first,last], binary_op shall neither modify elements nor invalidate iterators or subranges.\(^{294}\)

### 26.7.3 Partial sum

```cpp
template <class InputIterator1, class InputIterator2, class T>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init);

template <class InputIterator1, class InputIterator2, class T,
          class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init,
                BinaryOperation1 binary_op1,
                BinaryOperation2 binary_op2);
```

**Effects:** Computes its result by initializing the accumulator acc with the initial value init and then modifying it with acc = acc + (*i1) * (*i2) or acc = binary_op1(acc, binary_op2(*i1, *i2)) for every iterator i1 in the range [first, last) and iterator i2 in the range [first2, first2 + (last - first)) in order.

**Requires:** T shall meet the requirements of CopyConstructible (35) and Assignable (23.2) 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.\(^{295}\)

---

\(^{293}\) `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.

\(^{294}\) The use of fully closed ranges is intentional

\(^{295}\) The use of fully closed ranges is intentional

§ 26.7.3 965
template <class InputIterator, class OutputIterator>
OutputIterator partial_sum(
    InputIterator first, InputIterator last,
    OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator partial_sum(
    InputIterator first, InputIterator last,
    OutputIterator result, BinaryOperation binary_op);

Effects: For a non-empty range, the function creates an accumulator acc whose type is InputIterator’s value type, initializes it with *first, and assigns the result to *result. For every iterator i in \([first + 1, last)\) in order, acc is then modified by acc = acc + *i or acc = binary_op(acc, *i) and the result is assigned to *(result + (i - first)).

Returns: \(result + (last - first)\).

Complexity: Exactly \((last - first) - 1\) applications of the binary operation.

Requires: InputIterator’s value type shall be constructible from the type of *first. The result of the expression acc + *i or binary_op(acc, *i) shall be implicitly convertible to InputIterator’s value type. acc shall be writable to the result output iterator. In the ranges \([first, last]\) and \([result, result + (last - first)]\), binary_op shall neither modify elements nor invalidate iterators or subranges.\(^{296}\)

Remarks: result may be equal to first.

26.7.4 Adjacent difference

\[\text{adjacent.difference}\]

template <class InputIterator, class OutputIterator>
OutputIterator adjacent_difference(
    InputIterator first, InputIterator last,
    OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator adjacent_difference(
    InputIterator first, InputIterator last,
    OutputIterator result,
    BinaryOperation binary_op);

Effects: For a non-empty range, the function creates an accumulator acc whose type is InputIterator’s value type, initializes it with *first, and assigns the result to *result. For every iterator i in \([first + 1, last)\) in order, creates an object val whose type is InputIterator’s value type, initializes it with *i, computes val - acc or binary_op(val, acc), assigns the result to *(result + (i - first)), and move assigns from val to acc.

Requires: InputIterator’s value type shall be MoveAssignable \(^{36}\) and shall be constructible from the type of *first. acc shall be writable to the result output iterator. The result of the expression val - acc or binary_op(val, acc) shall be writable to the result output iterator. In the ranges \([first, last]\) and \([result, result + (last - first)]\), binary_op shall neither modify elements nor invalidate iterators or subranges.\(^{297}\)

Remarks: result may be equal to first.

Returns: \(result + (last - first)\).

\(^{296}\) The use of fully closed ranges is intentional.
\(^{297}\) The use of fully closed ranges is intentional.
Complexity: Exactly \( \text{last} - \text{first} \) - 1 applications of the binary operation.

26.7.5 Iota

```cpp
template <class ForwardIterator, class T>
void iota(ForwardIterator first, ForwardIterator last, T value);
```

 Requires: \( T \) shall be convertible to `ForwardIterator`’s value type. The expression `++val`, where `val` has type \( T \), shall be well formed.

 Effects: For each element referred to by the iterator \( i \) in the range \( \text{[first, last)} \), assigns \( *i = value \) and increments \( value \) as if by `++value`.

 Complexity: Exactly \( \text{last} - \text{first} \) increments and assignments.

26.8 C Library

The header `<ctgmath>` simply includes the headers `<ccomplex>` and `<cmath>`.

[Note: The overloads provided in C by magic macros are already provided in `<ccomplex>` and `<cmath>` by “sufficient” additional overloads. — end note]

Tables 116 and 117 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_ZERO</td>
</tr>
<tr>
<td>FP_ILOGBO</td>
<td>FP_NORMAL</td>
</tr>
<tr>
<td>Types:</td>
<td></td>
</tr>
<tr>
<td>double_t</td>
<td>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>exp2</td>
</tr>
<tr>
<td>asinh</td>
<td>exp</td>
</tr>
<tr>
<td>atan</td>
<td>expm1</td>
</tr>
<tr>
<td>atan2</td>
<td>fabs</td>
</tr>
<tr>
<td>atanh</td>
<td>fdim</td>
</tr>
<tr>
<td>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>isgreatequal</td>
</tr>
<tr>
<td>isfinite</td>
<td>isgreaterequal</td>
</tr>
<tr>
<td>isinf</td>
<td>islessequal</td>
</tr>
<tr>
<td>isnan</td>
<td>islesssgreater</td>
</tr>
<tr>
<td>isnormal</td>
<td>isnormal</td>
</tr>
<tr>
<td>isunordered</td>
<td>isunordered</td>
</tr>
<tr>
<td>signbit</td>
<td>isunordered</td>
</tr>
<tr>
<td>isgreater</td>
<td>isless</td>
</tr>
<tr>
<td>isless</td>
<td>islessgreater</td>
</tr>
</tbody>
</table>

The contents of these headers are the same as the Standard C library headers `<math.h>` and `<stdlib.h>` respectively, with the following changes:
The `rand` function has the semantics specified in the C standard, except that the implementation may specify that particular library functions may call `rand`. It is implementation-defined whether the `rand` function may introduce data races (17.6.4.8). [Note: The random number generation (26.5) facilities in this standard are often preferable to `rand`. — end note]

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.

The added signatures are:

```c
long abs(long); // labs()
long long abs(long long); // llabs()
ldiv_t div(long, long); // ldiv()
lldiv_t div(long long, long long); // lldiv()
```

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 exp10(float);
float fabs(float);
float fabsl(float);
float floor(float);
float fma(float, float, float);
float fmax(float, float);
float fmin(float, float);
```
float fmod(float, float);
float frexp(float, int*);
float hypot(float, float);
int ilogb(float);
float ldexp(float, int);
float lgamma(float);
long long llrint(float);
long long llround(float);
float log(float);
float log10(float);
float log1p(float);
float log2(float);
float logb(float);
long lrint(float);
long lround(float);
float modf(float, float*);
float nearbyint(float);
float nextafter(float, float);
float nexttoward(float, long double);
float pow(float, float);
float remainder(float, float);
float remquo(float, float, int *);
float rint(float);
float round(float);
float scalbln(float, long);
float scalbn(float, int);
float sin(float);
float sinh(float);
float sqrt(float);
float tan(float);
float tanh(float);
float tgamma(float);
float trunc(float);

double abs(double);    // fabs()

long double abs(long double);
long double acos(long double);
long double acosh(long double);
long double asin(long double);
long double asinh(long double);
long double atan(long double);
long double atan2(long double, long double);
long double atanh(long double);
long double cbrt(long double);
long double ceil(long double);
long double copysign(long double, long double);
long double cos(long double);
long double cosh(long double);
long double erf(long double);
long double erfc(long double);
long double exp(long double);
long double exp2(long double);
long double expm1(long double);
long double fabs(long double);

§ 26.8
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 C macros with the same names. The templates have the following declarations:

```cpp
namespace std {
    template <class T> bool signbit(T x);
    template <class T> int fpclassify(T x);
}
```
template <class T> bool isfinite(T x);
template <class T> bool isinf(T x);
template <class T> bool isnan(T x);
template <class T> bool isnormal(T x);

template <class T> bool isgreater(T x, T y);
template <class T> bool isgreaterequal(T x, T y);
template <class T> bool islessequal(T x, T y);
template <class T> bool isless(T x, T y);
template <class T> bool islessgreater(T x, T y);
template <class T> bool isunordered(T x, T y);
}

12 The templates behave the same as the C macros with corresponding names defined in 7.12.3, Classification macros, and 7.12.14, Comparison macros in the C standard.

See also: ISO C 7.5, 7.10.2, 7.10.6.
27  Input/output library  [input.output]

27.1  General  [input.output.general]

1  This Clause describes components that C++ programs may use to perform input/output operations.

2  The following subclauses describe requirements for stream parameters, and components for forward declarations of istreams, predefined istreams objects, base istreams classes, stream buffering, stream formatting and manipulators, string streams, and file streams, as summarized in Table 118.

Table 118 — Input/output library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.2 Requirements</td>
<td>&lt;iosfwd&gt;</td>
</tr>
<tr>
<td>27.3 Forward declarations</td>
<td>&lt;iostream&gt;</td>
</tr>
<tr>
<td>27.4 Standard istream objects</td>
<td>&lt;ios&gt;</td>
</tr>
<tr>
<td>27.5 Iostreams base classes</td>
<td>&lt;streambuf&gt;</td>
</tr>
<tr>
<td>27.6 Stream buffers</td>
<td>&lt;iomanip&gt;</td>
</tr>
<tr>
<td>27.7 Formatting and manipulators</td>
<td>&lt;iostream&gt;</td>
</tr>
<tr>
<td>27.8 String streams</td>
<td>&lt;sstream&gt;</td>
</tr>
<tr>
<td>27.9 File streams</td>
<td>&lt;fstream&gt;</td>
</tr>
</tbody>
</table>

3  Figure 7 illustrates relationships among various types described in this clause. A line from A to B indicates that A is an alias (e.g. a typedef) for B or that A is defined in terms of B.

Figure 7 — Stream position, offset, and size types [non-normative]
27.2 Iostreams requirements

27.2.1 Imbue Limitations

1 No function described in Clause 27 except for \texttt{ios\_base::imbue} and \texttt{basic\_filebuf::pubimbue} causes any instance of \texttt{basic\_ios::imbue} or \texttt{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 \texttt{imbue}, the behavior is undefined.

27.2.2 Positioning Type Limitations

1 The classes of Clause 27 with template arguments \texttt{charT} and \texttt{traits} behave as described if \texttt{traits::pos\_type} and \texttt{traits::off\_type} are \texttt{streampos} and \texttt{streamoff} respectively. Except as noted explicitly below, their behavior when \texttt{traits::pos\_type} and \texttt{traits::off\_type} are other types is implementation-defined.

2 In the classes of Clause 27, a template formal parameter with name \texttt{charT} represents a member of the set of types containing \texttt{char}, \texttt{wchar\_t}, and any other implementation-defined character types that satisfy the requirements for a character on which any of the iostream components can be instantiated.

27.2.3 Thread safety

1 Concurrent access to a stream object (27.8, 27.9), stream buffer object (27.6), or C Library stream (27.9.2) by multiple threads may result in a data race (1.10) unless otherwise specified (27.4). [\textit{Note: data races result in undefined behavior (1.10).} — end note]

2 If one thread makes a library call \texttt{a} that writes a value to a stream and, as a result, another thread reads this value from the stream through a library call \texttt{b} such that this does not result in a data race, then \texttt{a} happens before \texttt{b}.

27.3 Forward declarations

Header \texttt{<iosfwd>} synopsis

\begin{verbatim}
namespace std {
    template<class charT> class char_traits;
    template<> class char_traits<char>;
    template<> class char_traits<char16_t>;
    template<> class char_traits<char32_t>;
    template<class charT> class allocator;

    template <class charT, class traits = char_traits<charT> >
        class basic_ios;
    template <class charT, class traits = char_traits<charT> >
        class basic_streambuf;
    template <class charT, class traits = char_traits<charT> >
        class basic_istream;
    template <class charT, class traits = char_traits<charT> >
        class basic_ostream;
    template <class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
        class basic_iostream;

    template <class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
        class basic_stringbuf;
\end{verbatim}

§ 27.3
template <class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_istringstream;
template <class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_ostringstream;
template <class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_stringstream;

template <class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_filebuf;
template <class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_ifstream;
template <class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_ofstream;
template <class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_fstream;

template <class charT, class traits = char_traits<charT> >
class istreambuf_iterator;
template <class charT, class traits = char_traits<charT> >
class ostreambuf_iterator;

typedef basic_ios<char> ios;
typedef basic_ios<wchar_t> wios;

typedef basic_streambuf<char> streambuf;
typedef basic_istream<char> istream;
typedef basic_ostream<char> ostream;
typedef basic_iostream<char> iostream;

typedef basic_stringbuf<char> stringbuf;
typedef basic_istream<wchar_t> wistream;
typedef basic_ostream<wchar_t> wostream;
typedef basic_iostream<wchar_t> wiostream;

typedef basic_stringbuf<wchar_t> wstringbuf;
typedef basic_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;
typedef basic_ofstream<wchar_t> wofstream;

§ 27.3 974
typedef basic_fstream<wchar_t> wfstream;

template <class state> class fpos;
typedef fpos<char_traits<char>::state_type> streampos;
typedef fpos<char_traits<wchar_t>::state_type> wstreampos;

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.\(^{298}\)

[Note: The class template specialization basic_ios<charT,traits> serves as a virtual base class for the class templates basic_istream, basic_ostream, and class templates derived from them. basic_iostream is a class template derived from both basic_istream<charT,traits> and basic_ostream<charT,traits>.]  
The class template specialization basic_streambuf<charT,traits> serves as a base class for template classes basic_stringbuf and basic_filebuf.

The class template specialization basic_istream<charT,traits> serves as a base class for template classes basic_istringstream and basic_ifstream.

The class template specialization basic_ostream<charT,traits> serves as a base class for template classes basic_ostringstream and basic_ofstream.

The class template specialization basic_iostream<charT,traits> serves as a base class for template classes basic_stringstream and basic_fstream.

Other typedefs define instances of class templates specialized for char or wchar_t types.

Specializations of the class template fpos are used for specifying file position information.

The types streampos and wstreampos are used for positioning streams specialized on char and wchar_t respectively.

This synopsis suggests a circularity between streampos and char_traits<char>. An implementation can avoid this circularity by substituting equivalent types. One way to do this might be

```cpp
template<class stateT> class fpos { ... }; // depends on nothing
typedef ... _STATE; // implementation private declaration of stateT

typedef fpos<_STATE> streampos;

template<> struct char_traits<char> {
  typedef streampos
  pos_type;
};

— end note]

27.4 Standard iostream objects

[iostream.objects]

Header <iostream> synopsis

```
#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;
}

The header `<iostream>` declares objects that associate objects with the standard C streams provided for by
the functions declared in `<cstdio>` (27.9.2), and includes all the headers necessary to use these objects.

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 execu-
tion.\footnote{If it is possible for them to do so, implementations are encouraged to initialize the objects earlier than required.} The objects are not destroyed during program execution.\footnote{Constructors and destructors for static objects can access these objects to read input from `stdin` or write output to `stdout` or `stderr`.} The results of including `<iostream>` in a translation unit shall be as if `<iostream>` defined an instance of `ios_base::Init` with static storage duration. Similarly, the entire program shall behave as if there were at least one instance of `ios_base::Init` with static storage duration.

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.

Concurrent access to a synchronized (27.5.2.4) standard iostream object’s formatted and unformatted input
(27.7.1.1) and output (27.7.2.1) functions or a standard C stream by multiple threads shall not result in a data race (1.10). \[Note: users must still synchronize concurrent use of these objects and streams by multiple threads if they wish to avoid interleaved characters. \textit{— end note}\]}

### 27.4.1 Narrow stream objects

**istream cin;**

The object `cin` controls input from a stream buffer associated with the object `stdin`, declared in
`<cstdio>`.\footnote{Constructors and destructors for static objects can access these objects to read input from `stdin` or write output to `stdout` or `stderr`.}

After the object `cin` is initialized, `cin.tie()` returns `%cout`. Its state is otherwise the same as required for `basic_ios<char>::init` (27.5.4.1).

**ostream cout;**

The object `cout` controls output to a stream buffer associated with the object `stdout`, declared in
`<cstdio>` (27.9.2).

**ostream cerr;**

The object `cerr` controls output to a stream buffer associated with the object `stderr`, declared in
`<cstdio>` (27.9.2).

After the object `cerr` is initialized, `cerr.flags()` & `unitbuf` is nonzero and `cerr.tie()` returns
`%cout`. Its state is otherwise the same as required for `basic_ios<char>::init` (27.5.4.1).
The object `clog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdlib>` (27.9.2).

27.4.2 Wide stream objects [wide.stream.objects]

The object `wcin` controls input from a stream buffer associated with the object `stdin`, declared in `<cstdlib>`.

After the object `wcin` is initialized, `wcin.tie()` returns `&wcout`. Its state is otherwise the same as required for `basic_ios<wchar_t>::init` (27.5.4.1).

The object `wcout` controls output to a stream buffer associated with the object `stdout`, declared in `<cstdlib>` (27.9.2).

The object `wcerr` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdlib>` (27.9.2).

After the object `wcerr` is initialized, `wcerr.flags() & unitbuf` is nonzero and `wcerr.tie()` returns `&wcout`. Its state is otherwise the same as required for `basic_ios<wchar_t>::init` (27.5.4.1).

The object `wclog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdlib>` (27.9.2).

27.5 Iostreams base classes [iostreams.base]

Header `<ios>` synopsis

```cpp
#include <iosfwd>

namespace std {
    typedef implementation-defined streamoff;
    typedef implementation-defined streamsize;
    template <class stateT> class fpos;

    class ios_base;
    template <class charT, class traits = char_traits<charT>>
        class basic_ios;

    // 27.5.5, 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);

// 27.5.5.2 adjustfield:
ios_base& internal (ios_base& str);
ios_base& left    (ios_base& str);
ios_base& right   (ios_base& str);

// 27.5.5.3 basefield:
ios_base& dec    (ios_base& str);
ios_base& hex    (ios_base& str);
ios_base& oct    (ios_base& str);

// 27.5.5.4 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);

// 27.5.5.5 error reporting:
enum class io_errc {
    stream = 1
};

template <> struct is_error_code_enum<io_errc> : public true_type { };
error_code make_error_code(io_errc e);
error_condition make_error_condition(io_errc e);
const error_category& iostream_category();
}

27.5.1 Types
[stream.types]

typedef implementation-defined streamoff;

1 The type streamoff is a synonym for one of the signed basic integral types of sufficient size to represent the maximum possible file size for the operating system.\textsuperscript{301}

typedef implementation-defined streamsize;

2 The type streamsize is a synonym for one of the signed basic integral types. It is used to represent the number of characters transferred in an I/O operation, or the size of I/O buffers.\textsuperscript{302}

\footnote{Typically \texttt{long long}.}
\footnote{streamsize is used in most places where ISO C would use \texttt{size_t}. Most of the uses of streamsize could use \texttt{size_t},...}
27.5.2 Class ios_base

namespace std {
    class ios_base {
    public:
        class failure;

        // 27.5.2.1.2 fmtflags
        enum fmtflags {
            boolalpha = unspecified,
            dec = unspecified,
            fixed = unspecified,
            hex = unspecified,
            internal = unspecified,
            left = unspecified,
            oct = unspecified,
            right = unspecified,
            scientific = unspecified,
            showbase = unspecified,
            showpoint = unspecified,
            showpos = unspecified,
            skipws = unspecified,
            unitbuf = unspecified,
            uppercase = unspecified,
            adjustfield = unspecified,
            basefield = unspecified,
            floatfield = unspecified,
        };

        constexpr fmtflags operator~(fmtflags f);
        constexpr fmtflags operator&(fmtflags lhs, fmtflags rhs);
        constexpr fmtflags operator|(fmtflags lhs, fmtflags rhs);

        // 27.5.2.1.3 iostate
        enum iostate {
            badbit = unspecified,
            eofbit = unspecified,
            failbit = unspecified,
            goodbit = unspecified,
        };

        constexpr iostate operator~(iostate f);
        constexpr iostate operator&(iostate lhs, iostate rhs);
        constexpr iostate operator|(iostate lhs, iostate rhs);

        // 27.5.2.1.4 openmode
        enum openmode {
            app = unspecified,
            ate = unspecified,
            binary = unspecified,
            in = unspecified,
            out = unspecified,
            trunc = unspecified,

        }
    };
}

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).
constexpr openmode operator~(openmode f);
constexpr openmode operator&(openmode lhs, openmode rhs);
constexpr openmode operator|(openmode lhs, openmode rhs);

// 27.5.2.1.5 seekdir
enum seekdir {
    beg = unspecified,
    cur = unspecified,
    end = unspecified,
};

constexpr seekdir operator~(seekdir f);
constexpr seekdir operator&(seekdir lhs, seekdir rhs);
constexpr seekdir operator|(seekdir lhs, seekdir rhs);

class Init;

// 27.5.2.2 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);

// 27.5.2.3 locales:
locale imbue(const locale& loc);
locale getloc() const;

// 27.5.2.5 storage:
static int xalloc();
long& iword(int index);
void*& pword(int index);

// destructor
virtual ~ios_base();

// 27.5.2.6 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;  // exposition only
    long* iarray;     // exposition only
    void** parray;    // exposition only
};
}

ios_base defines several member types:

- a class failure derived from system_error;
- 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]

27.5.2.1 Types [ios.types]

27.5.2.1.1 Class ios_base::failure [ios::failure]

namespace std {
    class ios_base::failure : public system_error {
    public:
        explicit failure(const string& msg, const error_code& ec = io_errc::stream);
        explicit failure(const char* msg, const error_code& ec = io_errc::stream);
    }
}

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.

When throwing ios_base::failure exceptions, implementations should provide values of ec that identify the specific reason for the failure. [Note: Errors arising from the operating system would typically be reported as system_category() errors with an error value of the error number reported by the operating system. Errors arising from within the stream library would typically be reported as error_code(io_errc::stream, iostream_category()). — end note]
explicit failure(const string& msg, const error_code& ec = io_errc::stream);

Effects: Constructs an object of class failure by constructing the base class with msg and ec.

explicit failure(const char* msg, const error_code& ec = io_errc::stream);

Effects: Constructs an object of class failure by constructing the base class with msg and ec.

27.5.2.1.2 Type ios_base::fmtflags

enum fmtflags;

The type fmtflags is a bitmask type (17.5.2.1.3). Setting its elements has the effects indicated in Table 119.

Table 119 — 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</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</td>
</tr>
<tr>
<td>internal</td>
<td>adds fill characters at a designated internal point</td>
</tr>
<tr>
<td></td>
<td>in certain generated output, or identical to right if</td>
</tr>
<tr>
<td></td>
<td>no such point is designated</td>
</tr>
<tr>
<td>left</td>
<td>adds fill characters on the right (final positions)</td>
</tr>
<tr>
<td>oct</td>
<td>converts integer input or generates integer output in</td>
</tr>
<tr>
<td>right</td>
<td>adds fill characters on the left (initial positions)</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</td>
</tr>
<tr>
<td></td>
<td>generated integer output</td>
</tr>
<tr>
<td>showpoint</td>
<td>generates a decimal-point character unconditionally</td>
</tr>
<tr>
<td></td>
<td>in generated floating-point output</td>
</tr>
<tr>
<td>showpos</td>
<td>generates a + sign in non-negative generated numeric</td>
</tr>
<tr>
<td>skipws</td>
<td>skips leading whitespace before certain input</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</td>
</tr>
<tr>
<td></td>
<td>uppercase equivalents in generated output</td>
</tr>
</tbody>
</table>

Type fmtflags also defines the constants indicated in Table 120.

Table 120 — 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>

constexpr fmtflags ios_base::operator~(fmtflags f);

Returns: fmtflags( f).

constexpr fmtflags ios_base::operator&(fmtflags lhs, fmtflags rhs);

Returns: fmtflags(int(lhs) & int(rhs)).
constexpr fmtflags ios_base::operator|(fmtflags lhs, fmtflags rhs);

Returns: fmtflags(int(lhs) | int(rhs)).

### 27.5.2.1.3 Type `ios_base::iostate`

enum iostate;

The type `iostate` is a bitmask type (17.5.2.1.3) that contains the elements indicated in Table 121.

<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>

Type `iostate` also defines the constant:
— goodbit, the value zero.

constexpr iostate ios_base::operator~(iostate f);

Returns: iostate( f).

constexpr iostate ios_base::operator&(iostate lhs, iostate rhs);

Returns: iostate(int(lhs) & int(rhs)).

constexpr iostate ios_base::operator|(iostate lhs, iostate rhs);

Returns: iostate(int(lhs) | int(rhs)).

### 27.5.2.1.4 Type `ios_base::openmode`

denum openmode;

The type `openmode` is a bitmask type (17.5.2.1.3). It contains the elements indicated in Table 122.

<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>at</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>

constexpr openmode ios_base::operator~(openmode f);

Returns: openmode( f).

§ 27.5.2.1.4 983
constexpr openmode ios_base::operator&(openmode lhs, openmode rhs);
3  Returns: openmode(int(lhs) & int(rhs)).

constexpr openmode ios_base::operator|(openmode lhs, openmode rhs);
4  Returns: openmode(int(lhs) | int(rhs)).

27.5.2.1.5 Type ios_base::seekdir

enum seekdir;
1  The type seekdir is an enumerated type (17.5.2.1.2) that contains the elements indicated in Table 123.

<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>

constexpr seekdir ios_base::operator~(seekdir f);
2  Returns: seekdir( f).

constexpr seekdir ios_base::operator&(seekdir lhs, seekdir rhs);
3  Returns: seekdir(int(lhs) & int(rhs)).

constexpr seekdir ios_base::operator|(seekdir lhs, seekdir rhs);
4  Returns: seekdir(int(lhs) | int(rhs)).

27.5.2.1.6 Class ios_base::Init

namespace std {
   class ios_base::Init {
      public:
         Init();
         ~Init();
      private:
         static int init_cnt; // exposition only
   };
}
1  The class Init describes an object whose construction ensures the construction of the eight objects declared in <iostream> (27.4) that associate file stream buffers with the standard C streams provided for by the functions declared in <cstdio> (27.9.2).
2  For the sake of exposition, the maintained data is presented here as:

        — static int init_cnt, counts the number of constructor and destructor calls for class Init, initialized to zero.

Init();

§ 27.5.2.1.6
Effects: Constructs an object of class Init. Constructs and initializes the objects cin, cout, cerr, clog, wcin, wcout, wcerr, and wclog if they have not already been constructed and initialized.

~Init();

Effects: Destroys an object of class Init. If there are no other instances of the class still in existence, calls cout.flush(), cerr.flush(), clog.flush(), wcout.flush(), wcerr.flush(), wclog.flush().

27.5.2.2 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().
27.5.2.3  ios_base functions

locale imbue(const locale& loc);

1. Effects: Calls each registered callback pair \((fn, index)\) as \((\ast fn)(\text{imbuelongskip_event}, \ast \text{this}, index)\) at such a time that a call to \texttt{ios\_base::getloc()} from within \texttt{fn} returns the new locale value \texttt{loc}.

2. Returns: The previous value of \texttt{getloc()}.  

3. Postcondition: \texttt{loc == getloc()}.  

locale getloc() const;

4. Returns: If no locale has been imbued, a copy of the global C++ locale, \texttt{locale()}, in effect at the time of construction. Otherwise, returns the imbued locale, to be used to perform locale-dependent input and output operations.

27.5.2.4  ios_base static members

bool sync_with_stdio(bool sync = true);

1. Returns: \texttt{true} if the previous state of the standard iostream objects (27.4) was synchronized and otherwise returns \texttt{false}. The first time it is called, the function returns \texttt{true}.

2. 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.

3. When a standard iostream object \texttt{str} is \texttt{synchronized} with a standard stdio stream \texttt{f}, the effect of inserting a character \texttt{c} by

\begin{verbatim}
  fputc(f, c);
\end{verbatim}

is the same as the effect of

\begin{verbatim}
  str.rdbuf()->sputc(c);
\end{verbatim}

for any sequences of characters; the effect of extracting a character \texttt{c} by

\begin{verbatim}
  c = fgetc(f);
\end{verbatim}

is the same as the effect of

\begin{verbatim}
  c = str.rdbuf()->sbumpc(c);
\end{verbatim}

for any sequences of characters; and the effect of pushing back a character \texttt{c} by

\begin{verbatim}
  ungetc(c, f);
\end{verbatim}

is the same as the effect of

\begin{verbatim}
  str.rdbuf()->sputbackc(c);
\end{verbatim}

for any sequence of characters.\footnote{This implies that operations on a standard iostream object can be mixed arbitrarily with operations on the corresponding stdio stream. In practical terms, synchronization usually means that a standard iostream object and a standard stdio object share a buffer.}

§ 27.5.2.4
27.5.2.5 ios_base storage functions

static int xalloc();

Returns: index ++.

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. However, the value of the storage referred to is retained, so that until the next call to copyfmt, calling iword with the same index yields another reference to the same value. If the function fails and *this is a base subobject of a basic_ios<> object or subobject, the effect is equivalent to calling basic_ios<>::setstate(badbit) on the derived object (which may throw failure).

Returns: On success iarray[idx]. On failure, a valid long& initialized to 0.

void*& pword(int idx);

Effects: If parray is a null pointer, allocates an array of pointers to void of unspecified size and stores a pointer to its first element in parray. The function then extends the array pointed at by parray as necessary to include the element parray[idx]. Each newly allocated element of the array is initialized to a null pointer. The reference returned is invalid after any other operations on the object. However, the value of the storage referred to is retained, so that until the next call to copyfmt, calling pword with the same index yields another reference to the same value. If the function fails and *this is a base subobject of a basic_ios<> object or subobject, the effect is equivalent to calling basic_ios<>::setstate(badbit) on the derived object (which may throw failure).

Returns: On success parray[idx]. On failure a valid void*& initialized to 0.

Remarks: After a subsequent call to pword(int) for the same object, the earlier return value may no longer be valid.

27.5.2.6 ios_base callbacks

void register_callback(event_callback fn, int index);

Effects: Registers the pair (fn,index) such that during calls to imbue() (27.5.2.3), copyfmt(), or ~ios_base() (27.5.2.7), 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.

27.5.2.7 ios_base constructors/destructor

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) (27.5.2.6) as (**fn)(erase_event, *this, index) at such time that any ios_base member function called from within fn has well defined results.

27.5.3 Class template fpos

namespace std {
    template <class stateT> class fpos {
        public:
            // 27.5.3.1 Members
            stateT state() const;
            void state(stateT);
        private;
            stateT st; // exposition only
    };
}

27.5.3.1 fpos Members

    void state(stateT s);

Effects: Assign s to st.

stateT state() const;

Returns: Current value of st.

27.5.3.2 fpos requirements

Operations specified in Table 124 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 124 — Position type requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(i)</td>
<td></td>
<td></td>
<td>p == P(i)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>note: a destructor is assumed.</td>
</tr>
</tbody>
</table>

§ 27.5.3.2
### Table 124 — Position type requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>P p(i); P p = i;</td>
<td></td>
<td></td>
<td>post: ( p = P(i) ).</td>
</tr>
<tr>
<td>P(o)</td>
<td>fpos</td>
<td>converts from offset</td>
<td></td>
</tr>
<tr>
<td>0(p)</td>
<td>OFF_T</td>
<td>converts to offset</td>
<td>( P(0(p)) ).</td>
</tr>
<tr>
<td>p == q</td>
<td>convertible to bool</td>
<td>( == ) is an equivalence relation</td>
<td></td>
</tr>
<tr>
<td>p != q</td>
<td>convertible to bool</td>
<td>( !(p == q) )</td>
<td></td>
</tr>
<tr>
<td>q = p + o</td>
<td>fpos</td>
<td>+ offset</td>
<td>( q - o ).</td>
</tr>
<tr>
<td>p += o</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q = p - o</td>
<td>fpos</td>
<td>- offset</td>
<td>( q + o ).</td>
</tr>
<tr>
<td>p -= o</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o = p - q</td>
<td>OFF_T</td>
<td>distance</td>
<td>( q + o ).</td>
</tr>
<tr>
<td>streamsize(o)</td>
<td>streamsize</td>
<td>converts</td>
<td>( \text{streamsize}(O(sz)) ).</td>
</tr>
<tr>
<td>0(sz)</td>
<td>OFF_T</td>
<td>converts</td>
<td>( \text{streamsize}(O(sz)) ).</td>
</tr>
</tbody>
</table>

2 [Note: Every implementation is required to supply overloaded operators on `fpos` objects to satisfy the requirements of 27.5.3.2. It is unspecified whether these operators are members of `fpos`, global operators, or provided in some other way. — end note]

3 Stream operations that return a value of type `traits::pos_type` return `P(O(-1))` as an invalid value to signal an error. If this value is used as an argument to any `istream`, `ostream`, or `streambuf` member that accepts a value of type `traits::pos_type` then the behavior of that function is undefined.

#### 27.5.4 Class template `basic_ios`

```cpp
namespace std {
    template <class charT, class traits = char_traits<charT>>
    class basic_ios : public ios_base {
        public:

        // types:
        typedef charT char_type;
        typedef typename traits::int_type int_type;
        typedef typename traits::pos_type pos_type;
        typedef typename traits::off_type off_type;
        typedef traits traits_type;

        explicit operator bool() const;
        bool operator!() const;
        iostate rdstate() const;
        void clear(iostate state = goodbit);
        void setstate(iostate state);
        bool good() const;
        bool eof() const;
        bool fail() const;
        bool bad() const;

        iostate exceptions() const;
    }
}
```

§ 27.5.4 989
27.5.4.1 Constructor/destructor:

```cpp
explicit basic_ios(basic_streambuf<charT,traits>* sb);
virtual ~basic_ios();
```

27.5.4.2 Members:

```cpp
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;
```

```cpp
protected:
basic_ios();
void init(basic_streambuf<charT,traits>* sb);
void move(basic_ios& rhs);
void move(basic_ios&& rhs);
void swap(basic_ios& rhs);
void set_rdbuf(basic_streambuf<charT,traits>* sb);

};
```

27.5.4.1 basic_ios constructors [basic.ios.cons]

```cpp
explicit basic_ios(basic_streambuf<charT,traits>* sb);
```

1. **Effects:** Constructs an object of class `basic_ios`, assigning initial values to its member objects by calling `init(sb)`.

```cpp
basic_ios();
```

2. **Effects:** Constructs an object of class `basic_ios` (27.5.2.7) 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.

```cpp
~basic_ios();
```

3. **Remarks:** The destructor does not destroy `rdbuf()`.

```cpp
void init(basic_streambuf<charT,traits>* sb);
```

**Postconditions:** The postconditions of this function are indicated in Table 125.

§ 27.5.4.1
Table 125 — `basic_ios::init()` effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rdbuf()</code></td>
<td><code>sb</code></td>
</tr>
<tr>
<td><code>tie()</code></td>
<td>0</td>
</tr>
<tr>
<td><code>rdstate()</code></td>
<td><code>goodbit</code> if <code>sb</code> is not a null pointer, otherwise <code>badbit</code>.</td>
</tr>
<tr>
<td><code>exceptions()</code></td>
<td><code>goodbit</code></td>
</tr>
<tr>
<td><code>flags()</code></td>
<td>`skipws</td>
</tr>
<tr>
<td><code>width()</code></td>
<td>0</td>
</tr>
<tr>
<td><code>precision()</code></td>
<td>6</td>
</tr>
<tr>
<td><code>fill()</code></td>
<td><code>widen(' ')</code>;</td>
</tr>
<tr>
<td><code>getloc()</code></td>
<td>a copy of the value returned by <code>locale()</code></td>
</tr>
<tr>
<td><code>iarray</code></td>
<td>a null pointer</td>
</tr>
<tr>
<td><code>parray</code></td>
<td>a null pointer</td>
</tr>
</tbody>
</table>

27.5.4.2 Member functions

```cpp
basic_ostream<charT,traits>* tie() const;
```

1. **Returns:** An output sequence that is tied to (synchronized with) the sequence controlled by the stream buffer.

```cpp
basic_ostream<charT,traits>* tie(basic_ostream<charT,traits>* tiestr);
```

2. **Requires:** If `tiestr` is not null, `tiestr` must not be reachable by traversing the linked list of tied stream objects starting from `tiestr->tie()`.
3. **Postcondition:** `tiestr == tie()`.
4. **Returns:** The previous value of `tie()`.

```cpp
basic_streambuf<charT,traits>* rdbuf() const;
```

5. **Returns:** A pointer to the streambuf associated with the stream.

```cpp
basic_streambuf<charT,traits>* rdbuf(basic_streambuf<charT,traits>* sb);
```

6. **Postcondition:** `sb == rdbuf()`.
7. **Effects:** Calls `clear()`.
8. **Returns:** The previous value of `rdbuf()`.

```cpp
locale imbue(const locale& loc);
```

9. **Effects:** Calls `ios_base::imbue(loc)` (27.5.2.3) and if `rdbuf() != 0 then rdbuf()->pubimbue(loc)` (27.6.2.2.1).
10. **Returns:** The prior value of `ios_base::imbue()`.

```cpp
char narrow(char_type c, char dfault) const;
```

11. **Returns:** `use_facet<ctype<char_type>>>(getloc()).narrow(c,dfault)`
12  

  Returns: use_facet< ctype<char_type> >(getloc()).widen(c)

  char_type fill() const;

13  

  Returns: The character used to pad (fill) an output conversion to the specified field width.

  char_type fill(char_type fillch);

14  

  Postcondition: traits::eq(fillch, fill())

  Returns: The previous value of fill().

basic_ios& copyfmt(const basic_ios& rhs);

16  

  Effects: If (this == &rhs) does nothing. Otherwise assigns to the member objects of *this the corresponding member objects of rhs as follows:

  1. calls each registered callback pair (fn, index) as (*fn)(erase_event, *this, index);
  2. assigns to the member objects of *this the corresponding member objects of rhs, except that
     — rdstate(), rdbuf(), and exceptions() are left unchanged;
     — the contents of arrays pointed at by pword and iword are copied, not the pointers themselves;
     — 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;
  3. calls each callback pair that was copied from rhs as (*fn)(copyfmt_event, *this, index);
  4. calls exceptions(rhs.except()).

17  

  Note: The second pass through the callback pairs permits a copied pword value to be zeroed, or to
  have its referent deep copied or reference counted, or to have other special action taken.

18  

  Postconditions: The postconditions of this function are indicated in Table 126.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdbuf()</td>
<td>unchanged</td>
</tr>
<tr>
<td>tie()</td>
<td>rhs.tie()</td>
</tr>
<tr>
<td>rdstate()</td>
<td>unchanged</td>
</tr>
<tr>
<td>exceptions()</td>
<td>rhs.exceptions()</td>
</tr>
<tr>
<td>flags()</td>
<td>rhs.flags()</td>
</tr>
<tr>
<td>width()</td>
<td>rhs.width()</td>
</tr>
<tr>
<td>precision()</td>
<td>rhs.precision()</td>
</tr>
<tr>
<td>fill()</td>
<td>rhs.fill()</td>
</tr>
<tr>
<td>getloc()</td>
<td>rhs.getloc()</td>
</tr>
</tbody>
</table>

19  

  Returns: *this.

307) This suggests an infinite amount of copying, but the implementation can keep track of the maximum element of the arrays that is non-zero.
void move(basic_ios& rhs);
void move(basic_ios& rhs);

Postconditions: *this shall have the state that rhs had before the function call, except that rdbuf() shall return 0. rhs shall be in a valid but unspecified state, except that rhs.rdbuf() shall return the same value as it returned before the function call, and rhs.tie() shall return 0.

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.

27.5.4.3 basic_ios flags functions

explicit operator bool() const;

Returns: !fail().

bool operator!() const;

Returns: fail().

iostate rdstate() const;

Returns: The error state of the stream buffer.

void clear(iostate state = goodbit);

Postcondition: If rdbuf()! = 0 then state == rdstate(); otherwise rdstate()=(state | ios_base::badbit).

Effects: If ((state | (rdbuf() ? goodbit : badbit)) & exceptions()) == 0, returns. Otherwise, the function throws an object fail of class basic_ios::failure (27.5.2.1.1), constructed with implementation-defined argument values.

void setstate(iostate state);

Effects: Calls clear(rdstate() | state) (which may throw basic_ios::failure (27.5.2.1.1)).

bool good() const;

Returns: rdstate() == 0

bool eof() const;

Returns: true if eofbit is set in rdstate().

§ 27.5.4.3
bool fail() const;

   Returns: true if failbit or badbit is set in rdstate().308

bool bad() const;

   Returns: true if badbit is set in rdstate().

iostate exceptions() const;

   Returns: A mask that determines what elements set in rdstate() cause exceptions to be thrown.

void exceptions(iostate except);

   Postcondition: except == exceptions().

   Effects: Calls clear(rdstate()).

27.5.5  ios_base manipulators  [std.ios.manip]

27.5.5.1 fmtflags manipulators  [fmtflags.manip]

ios_base& boolalpha(ios_base& str);

1   Effects: Calls str.setf(ios_base::boolalpha).

2   Returns: str.

ios_base& noboolalpha(ios_base& str);

3   Effects: Calls str.unsetf(ios_base::boolalpha).

4   Returns: str.

ios_base& showbase(ios_base& str);

5   Effects: Calls str.setf(ios_base::showbase).

6   Returns: str.

ios_base& noshowbase(ios_base& str);

7   Effects: Calls str.unsetf(ios_base::showbase).

8   Returns: str.

ios_base& showpoint(ios_base& str);

9   Effects: Calls str.setf(ios_base::showpoint).

10  Returns: str.

ios_base& noshownpoint(ios_base& str);

11  Effects: Calls str.unsetf(ios_base::showpoint).

12  Returns: str.

308) Checking badbit also for fail() is historical practice.
ios_base& showpos(ios_base& str);
  \textit{Effects:} Calls \texttt{str.setf(ios\_base::showpos)}.
  \textit{Returns:} \texttt{str}.

ios_base& noshowpos(ios_base& str);
  \textit{Effects:} Calls \texttt{str.unsetf(ios\_base::showpos)}.
  \textit{Returns:} \texttt{str}.

ios_base& skipws(ios_base& str);
  \textit{Effects:} Calls \texttt{str.setf(ios\_base::skipws)}.
  \textit{Returns:} \texttt{str}.

ios_base& noskipws(ios_base& str);
  \textit{Effects:} Calls \texttt{str.unsetf(ios\_base::skipws)}.
  \textit{Returns:} \texttt{str}.

ios_base& uppercase(ios_base& str);
  \textit{Effects:} Calls \texttt{str.setf(ios\_base::uppercase)}.
  \textit{Returns:} \texttt{str}.

ios_base& nouppercase(ios_base& str);
  \textit{Effects:} Calls \texttt{str.unsetf(ios\_base::uppercase)}.
  \textit{Returns:} \texttt{str}.

ios_base& unitbuf(ios_base& str);
  \textit{Effects:} Calls \texttt{str.setf(ios\_base::unitbuf)}.
  \textit{Returns:} \texttt{str}.

ios_base& nounitbuf(ios_base& str);
  \textit{Effects:} Calls \texttt{str.unsetf(ios\_base::unitbuf)}.
  \textit{Returns:} \texttt{str}.

\textbf{27.5.5.2 \textit{adjustfield} manipulators} [adjustfield.manip]

ios_base& internal(ios_base& str);
  \textit{Effects:} Calls \texttt{str.setf(ios\_base::internal, ios\_base::adjustfield)}.
  \textit{Returns:} \texttt{str}.

ios_base& left(ios_base& str);
  \textit{Effects:} Calls \texttt{str.setf(ios\_base::left, ios\_base::adjustfield)}.
  \textit{Returns:} \texttt{str}.

\S 27.5.5.2
ios_base& right(ios_base& str);
5  Effects: Calls str.setf(ios_base::right, ios_base::adjustfield).
6  Returns: str.

27.5.5.3 basefield manipulators

ios_base& dec(ios_base& str);
1  Effects: Calls str.setf(ios_base::dec, ios_base::basefield).
2  Returns: str\(^309\).

ios_base& hex(ios_base& str);
3  Effects: Calls str.setf(ios_base::hex, ios_base::basefield).
4  Returns: str.

ios_base& oct(ios_base& str);
5  Effects: Calls str.setf(ios_base::oct, ios_base::basefield).
6  Returns: str.

27.5.5.4 floatfield manipulators

ios_base& fixed(ios_base& str);
1  Effects: Calls str.setf(ios_base::fixed, ios_base::floatfield).
2  Returns: str.

ios_base& scientific(ios_base& str);
3  Effects: Calls str.setf(ios_base::scientific, ios_base::floatfield).
4  Returns: str.

ios_base& hexfloat(ios_base& str);
5  Effects: Calls str.setf(ios_base::fixed | ios_base::scientific, ios_base::floatfield).
6  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);
8  Effects: Calls str.unsetf(ios_base::floatfield).
9  Returns: str.

\(^{309}\) 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.
27.5.5.5 Error reporting

error_code make_error_code(io_errc e);

Returns: error_code(static_cast<int>(e), iostream_category()).

error_condition make_error_condition(io_errc e);

Returns: error_condition(static_cast<int>(e), iostream_category()).

const error_category& iostream_category();

Returns: a reference to an object of a type derived from class error_category.

The object’s default_error_condition and equivalent virtual functions shall behave as specified for the class error_category. The object’s name virtual function shall return a pointer to the string "iostream".

27.6 Stream buffers

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.

27.6.1 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.

— 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 \texttt{xend} here).

3 The following semantic constraints shall always apply for any set of three pointers for a sequence, using the pointer names given immediately above:

— If \texttt{xnext} is not a null pointer, then \texttt{xbeg} and \texttt{xend} shall also be non-null pointers into the same \texttt{charT} array, as described above; otherwise, \texttt{xbeg} and \texttt{xend} shall also be null.

— If \texttt{xnext} is not a null pointer and \texttt{xnext} < \texttt{xend} for an output sequence, then a write position is available. In this case, \texttt{*xnext} shall be assignable as the next element to write (to put, or to store a character value, into the sequence).

— If \texttt{xnext} is not a null pointer and \texttt{xbeg} < \texttt{xnext} for an input sequence, then a putback position is available. In this case, \texttt{xnext[-1]} shall have a defined value and is the next (preceding) element to store a character that is put back into the input sequence.

— If \texttt{xnext} is not a null pointer and \texttt{xnext} < \texttt{xend} for an input sequence, then a read position is available. In this case, \texttt{*xnext} shall have a defined value and is the next element to read (to get, or to obtain a character value, from the sequence).

\section{Class template \texttt{basic\_streambuf<charT, traits>}}

\begin{verbatim}
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_streambuf {
    public:

    // types:
    typedef charT char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits traits_type;
    virtual ~basic_streambuf();

    // 27.6.2.2.1 locales:
    locale pubimbue(const locale& loc);
    locale getloc() const;

    // 27.6.2.2.2 buffer and positioning:
    basic_streambuf<char_type, traits>*
      pubsetbuf(char_type* s, streamsize n);
    pos_type pubseekoff(off_type off, ios_base::seekdir way,
      ios_base::openmode which =
      ios_base::in | ios_base::out);
    pos_type pubseekpos(pos_type sp,
      ios_base::openmode which =
      ios_base::in | ios_base::out);
    int pubsync();

    // Get and put areas:
    // 27.6.2.2.3 Get area:
    streamsize in_avail();
    int_type snextc();
    int_type sbumpc();
  }
}
\end{verbatim}

\section*{§ 27.6.2}
int_type sgetc();
streamsize sgetn(char_type* s, streamsize n);

// 27.6.2.2.4 Putback:
int_type sputbackc(char_type c);
int_type sungetc();

// 27.6.2.2.5 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);

// 27.6.2.3.2 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);

// 27.6.2.3.3 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);

// 27.6.2.4 virtual functions:
// 27.6.2.4.1 Locales:
virtual void imbue(const locale& loc);

// 27.6.2.4.2 Buffer management and positioning:
virtual basic_streambuf<char_type, 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();

// 27.6.2.4.3 Get area:
virtual streamsize showmanyc();
virtual streamsize xsgetn(char_type* s, streamsize n);
virtual int_type underflow();
virtual int_type uflow();

// 27.6.2.4.4 Putback:
virtual int_type pbackfail(int_type c = traits::eof());

// 27.6.2.4.5 Put area:
The class template `basic_streambuf<CharT,traits>` serves as an abstract base class for deriving various stream buffers whose objects each control two character sequences:

- a character input sequence;
- a character output sequence.

### 27.6.2.1 basic_streambuf constructors

#### basic_streambuf();

**Effects:** Constructs an object of class `basic_streambuf<CharT,traits>` and initializes:

- all its pointer member objects to null pointers,
- the `getloc()` member to a copy of the global locale, `locale()`, at the time of construction.

**Remarks:** Once the `getloc()` member is initialized, results of calling locale member functions, and of members of facets so obtained, can safely be cached until the next time the member `imbue` is called.

#### basic_streambuf(const basic_streambuf& rhs);

**Effects:** Constructs a copy of `rhs`.

**Postconditions:**

- `eback()` == `rhs.eback()`
- `gptr()` == `rhs.gptr()`
- `egptr()` == `rhs.egptr()`
- `pbase()` == `rhs.pbase()`
- `pptr()` == `rhs.pptr()`
- `epptr()` == `rhs.epptr()`
- `getloc()` == `rhs.getloc()`

**Effects:** None.

### 27.6.2.2 basic_streambuf public member functions

#### 27.6.2.2.1 Locales

`locale pubimbue(const locale& loc);`

---

310) The default constructor is protected for class `basic_streambuf` to assure that only objects for classes derived from this class may be constructed.
Postcondition: \( \text{loc} == \text{getloc()} \).

Effects: Calls \( \text{imbue}(	ext{loc}) \).

Returns: Previous value of \( \text{getloc()} \).

locale getloc() const;

Returns: If \( \text{pubimbue}() \) has ever been called, then the last value of \( \text{loc} \) supplied, otherwise the current global locale, \( \text{locale()} \), in effect at the time of construction. If called after \( \text{pubimbue}() \) has been called but before \( \text{pubimbue} \) has returned (i.e., from within the call of \( \text{imbue}() \)) then it returns the previous value.

27.6.2.2.2 Buffer management and positioning

[streambuf.buffer]

basic_streambuf<\char_type,\traits>* pubsetbuf(\char_type* s, \streamsize n);

Returns: \( \text{setbuf}(s, n) \).

pos_type pubseekoff(off_type off, ios_base::seekdir way,
   ios_base::openmode which = ios_base::in | ios_base::out);

Returns: \( \text{seekoff}(off, \text{way}, \text{which}) \).

pos_type pubseekpos(pos_type sp,
   ios_base::openmode which = ios_base::in | ios_base::out);

Returns: \( \text{seekpos}(sp, \text{which}) \).

int pubsync();

Returns: \( \text{sync}() \).

27.6.2.2.3 Get area

[streambuf.pub.get]

streamsize in_avail();

Returns: If a read position is available, returns \( \text{egptr()} - \text{gptr()} \). Otherwise returns \( \text{showmanyc}() \) (27.6.2.4.3).

int_type snextc();

Effects: Calls \( \text{sbumpc}() \).

Returns: If that function returns \( \text{traits::eof()} \), returns \( \text{traits::eof()} \). Otherwise, returns \( \text{sgetc}() \).

int_type sbumpc();

Returns: If the input sequence read position is not available, returns \( \text{uflow}() \). Otherwise, returns \( \text{traits::to_int_type}(*\text{gptr}()) \) and increments the next pointer for the input sequence.

int_type sgetc();

Returns: If the input sequence read position is not available, returns \( \text{underflow}() \). Otherwise, returns \( \text{traits::to_int_type}(*\text{gptr}()) \).

streamsize sgetn(char_type* s, streamsize n);

Returns: \( \text{xsgtn}(s, n) \).
27.6.2.2.4 Putback

int_type sputbackc(char_type c);

Returns: If the input sequence putback position is not available, or if traits::eq(c,gptr()[-1]) is false, returns pbackfail(traits::to_int_type(c)). Otherwise, decrements the next pointer for the input sequence and returns traits::to_int_type(*gptr()).

int_type sungetc();

Returns: If the input sequence putback position is not available, returns pbackfail(). Otherwise, decrements the next pointer for the input sequence and returns traits::to_int_type(*gptr()).

27.6.2.2.5 Put area

int_type sputc(char_type c);

Returns: If the output sequence write position is not available, returns overflow(traits::to_int_type(c)). Otherwise, stores c at the next pointer for the output sequence, increments the pointer, and returns traits::to_int_type(c).

streamsize sputn(const char_type* s, streamsize n);

Returns: xsputn(s,n).

27.6.2.3 basic_streambuf protected member functions

27.6.2.3.1 Assignment

basic_streambuf& operator=(const basic_streambuf& rhs);

Effects: Assigns the data members of rhs to *this.

Postconditions:

- eback() == rhs.eback()
- gptr() == rhs.gptr()
- egptr() == rhs.egptr()
- pbase() == rhs.pbase()
- pptr() == rhs.pptr()
- eptr() == rhs.eptr()
- getloc() == rhs.getloc()

Returns: *this.

void swap(basic_streambuf& rhs);

Effects: Swaps the data members of rhs and *this.
27.6.2.3.2 Get area access

char_type* eback() const;
1
   Returns: The beginning pointer for the input sequence.

char_type* gptr() const;
2
   Returns: The next pointer for the input sequence.

char_type* egptr() const;
3
   Returns: The end pointer for the input sequence.

void gbump(int n);
4
   Effects: Adds n to the next pointer for the input sequence.

void setg(char_type* gbeg, char_type* gnext, char_type* gend);
5
   Postconditions: gbeg == eback(), gnext == gptr(), and gend == egptr().

27.6.2.3.3 Put area access

char_type* pbase() const;
1
   Returns: The beginning pointer for the output sequence.

char_type* pptr() const;
2
   Returns: The next pointer for the output sequence.

char_type* epptr() const;
3
   Returns: The end pointer for the output sequence.

void pbump(int n);
4
   Effects: Adds n to the next pointer for the output sequence.

void setp(char_type* pbeg, char_type* pend);
5
   Postconditions: pbeg == pbase(), pbeg == pptr(), and pend == epptr().

27.6.2.4 basic_streambuf virtual functions

27.6.2.4.1 Locales

void imbue(const locale&);
1
   Effects: Change any translations based on locale.

Remarks: Allows the derived class to be informed of changes in locale at the time they occur. Between
invocations of this function a class derived from streambuf can safely cache results of calls to locale
functions and to members of facets so obtained.

Default behavior: Does nothing.
27.6.2.4.2 Buffer management and positioning

basic_streambuf* setbuf(char_type* s, streamsize n);

Effects: Influences stream buffering in a way that is defined separately for each class derived from basic_streambuf in this Clause (27.8.1.4, 27.9.1.5).

Default behavior: Does nothing. Returns this.

pos_type seekoff(off_type off, ios_base::seekdir way,
    ios_base::openmode which
    = ios_base::in | ios_base::out);

Effects: Alters the stream positions within one or more of the controlled sequences in a way that is defined separately for each class derived from basic_streambuf in this Clause (27.8.1.4, 27.9.1.5).

Default behavior: Returns pos_type(off_type(-1)).

pos_type seekpos(pos_type sp,
    ios_base::openmode which
    = ios_base::in | ios_base::out);

Effects: Alters the stream positions within one or more of the controlled sequences in a way that is defined separately for each class derived from basic_streambuf in this Clause (27.8.1, 27.9.1.1).

Default behavior: Returns pos_type(off_type(-1)).

int sync();

Effects: Synchronizes the controlled sequences with the arrays. That is, if pbase() is non-null the characters between pbase() and pptr() are written to the controlled sequence. The pointers may then be reset as appropriate.

Returns: -1 on failure. What constitutes failure is determined by each derived class (27.9.1.5).

Default behavior: Returns zero.

27.6.2.4.3 Get area

streamsize showmanyc();

Returns: an estimate of the number of characters available in the sequence, or -1. If it returns a positive value, then successive calls to underflow() will not return traits::eof() until at least that number of characters have been extracted from the stream. If showmanyc() returns -1, then calls to underflow() or uflow() will fail.

Default behavior: Returns zero.

Remarks: Uses traits::eof().

streamsize xsgetn(char_type* s, streamsize n);

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().

311) The morphemes of showmanyc are “es-how-many-see”, not “show-manic”.
312) 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.”
Assigning stops when either \( n \) characters have been assigned or a call to `sbumpc()` would return `traits::eof()`.

**Returns:** The number of characters assigned.\(^{313}\)

**Remarks:** Uses `traits::eof()`.

```c
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:

1. If `gptr()` is non-NULL, then the `egptr() - gptr()` characters starting at `gptr()`, otherwise the empty sequence.
2. Some sequence (possibly empty) of characters read from the input sequence.

The **result character** is

1. If the pending sequence is non-empty, the first character of the sequence.
2. 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:

1. If `eback()` is null then empty,
2. Otherwise the `gptr() - eback()` characters beginning at `eback()`.

**Effects:** The function sets up the `gptr()` and `egptr()` satisfying one of:

1. 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
2. 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:

1. 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.
2. 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()`.

```c
int_type uflow();
```

---

\(^{313}\) Classes derived from `basic_streambuf` can provide more efficient ways to implement `xsgetn()` and `xsputn()` by overriding these definitions from the base class.
Requires: The constraints are the same as for underflow(), except that the result character shall be transferred from the pending sequence to the backup sequence, and the pending sequence shall not be empty before the transfer.

Default behavior: Calls underflow(). If underflow() returns traits::eof(), returns traits::eof(). Otherwise, returns the value of traits::to_int_type(*gptr()) and increment the value of the next pointer for the input sequence.

Returns: traits::eof() to indicate failure.

27.6.2.4.4 Putback

int_type pbackfail(int_type c = traits::eof());

Remarks: The public functions of basic_streambuf call this virtual function only when gptr() is null, gptr() == eback(), or traits::eq(traits::to_char_type(c),*gptr()[-1]) returns false. Other calls shall also satisfy that constraint.

The pending sequence is defined as for underflow(), with the modifications that

— If traits::eq_int_type(c,traits::eof()) returns true, then the input sequence is backed up one character before the pending sequence is determined.
— If traits::eq_int_type(c,traits::eof()) return false, then c is prepended. Whether the input sequence is backed up or modified in any other way is unspecified.

Postcondition: On return, the constraints of gptr(), eback(), and pptr() are the same as for underflow().

Returns: traits::eof() to indicate failure. Failure may occur because the input sequence could not be backed up, or if for some other reason the pointers could not be set consistent with the constraints. pbackfail() is called only when put back has really failed.

Returns some value other than traits::eof() to indicate success.

Default behavior: Returns traits::eof().

27.6.2.4.5 Put area

streamsize xsputn(const char_type* s, streamsize n);

Effects: Writes up to n characters to the output sequence as if by repeated calls to sputc(c). The characters written are obtained from successive elements of the array whose first element is designated by s. Writing stops when either n characters have been written or a call to sputc(c) would return traits::eof(). Is is unspecified whether the function calls overflow() when pptr() == epptr() becomes true or whether it achieves the same effects by other means.

Returns: The number of characters written.

int_type overflow(int_type c = traits::eof());

Effects: Consumes some initial subsequence of the characters of the pending sequence. The pending sequence is defined as the concatenation of

a) if pbase() is NULL then the empty sequence otherwise, pptr() - pbase() characters beginning at pbase().

§ 27.6.2.4.5
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().

27.7 Formatting and manipulators [iostream.format]

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>
  basic_istream<charT,traits>& ws(basic_istream<charT,traits>&& is);

template <class charT, class traits, class T>
  basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>&& is, T& x);
}

Header <ostream> synopsis

namespace std {
  template <class charT, class traits = char_traits<charT> >
    class basic_ostream;

314) That is, for each class derived from an instance of basic_streambuf in this Clause (27.8.1, 27.9.1.1), a specification of how consuming a character effects the associated output sequence is given. There is no requirement on a program-defined class.

315) 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);

template <class charT, class traits, class T>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>&& os, const T& x);

Header <iomanip> synopsis

namespace std {
    // types T1, T2, ... are unspecified implementation types
    T1 resetiosflags(ios_base::fmtflags mask);
    T2 setiosflags (ios_base::fmtflags mask);
    T3 setbase(int base);
    template<charT> T4 setfill(charT c);
    T5 setprecision(int n);
    T6 setw(int n);
    template <class moneyT> T7 get_money(moneyT& mon, bool intl = false);
    template <class moneyT> T8 put_money(const moneyT& mon, bool intl = false);
    template <class charT> T9 get_time(struct tm* tmb, const charT* fmt);
    template <class charT> T10 put_time(const struct tm* tmb, const charT* fmt);
}

27.7.1 Input streams

The header <istream> defines two types and a function signature that control input from a stream buffer along with a function template that extracts from stream rvalues.

27.7.1.1 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 (27.5.4)): 
        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;

        // 27.7.1.1.1 Constructor/destructor: 
        explicit basic_istream(basic_streambuf<charT,traits>* sb);
        virtual ~basic_istream();

        // 27.7.1.1.3 Prefix/suffix: 
        class sentry;
    }

§ 27.7.1.1
// 27.7.1.2 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>>(
    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);

// 27.7.1.3 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();

int sync();

pos_type tellg();

basic_istream<charT,traits>& seekg(pos_type);

basic_istream<charT,traits>& seekg(off_type, ios_base::seekdir);

protected:

§ 27.7.1.1
basic_istream(basic_istream&& rhs);

    // 27.7.1.1.2 Assign/swap:
    basic_istream& operator=(basic_istream&& rhs);
    void swap(basic_istream rhs);
};

// 27.7.1.2.3 character extraction templates:
template<class charT, class traits>
    basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>&,
        charT&);
template<class traits>
    basic_istream<char, traits>& operator>>(basic_istream<char, traits>&,
        unsigned char&);
template<class traits>
    basic_istream<char, traits>& operator>>(basic_istream<char, traits>&,
        signed char&);
template<class charT, class traits>
    basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>&,
        charT*);
template<class traits>
    basic_istream<char, traits>& operator>>(basic_istream<char, traits>&,
        unsigned char*);
template<class traits>
    basic_istream<char, traits>& operator>>(basic_istream<char, traits>&,
        signed char*);
}

1 The class basic_istream defines a number of member function signatures that assist in reading and interpreting input from sequences controlled by a stream buffer.

2 Two groups of member function signatures share common properties: the formatted input functions (or extractors) and the unformatted input functions. Both groups of input functions are described as if they obtain (or extract) input characters by calling rdbuf()->sbumpc() or rdbuf()->sgetc(). They may use other public members of istream.

3 If rdbuf()->sbumpc() or rdbuf()->sgetc() returns traits::eof(), then the input function, except as explicitly noted otherwise, completes its actions and does setstate(eofbit), which may throw ios_base::failure (27.5.4.3), before returning.

4 If one of these called functions throws an exception, then unless explicitly noted otherwise, the input function sets badbit in error state. If badbit is on in exceptions(), the input function rethrows the exception without completing its actions, otherwise it does not throw anything and proceeds as if the called function had returned a failure indication.

27.7.1.1.1 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) (27.5.4.1).

Postcondition: gcount() == 0
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().

27.7.1.1.2 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().

27.7.1.1.3 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
    public:
        explicit sentry(basic_istream< charT, traits >& is, bool noskipws = false);
        sentry();
        explicit operator bool() const { return ok_; }
        sentry(const sentry&) = delete;
        sentry& operator=(const sentry&) = delete;
    };
}

The class sentry defines a class that is responsible for doing exception safe prefix and suffix operations.

explicit sentry(basic_istream< charT, traits >& is, bool noskipws = false);

Effects: If is.good() is false, calls is.setstate(failbit). Otherwise, prepares for formatted or unformattted 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.316 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).

316) 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.
Remarks: The constructor `explicit sentry(basic_istream<charT,traits>& is, bool noskipws = false)` uses the currently imbued locale in `is`, to determine whether the next input character is whitespace or not.

To decide if the character `c` is a whitespace character, the constructor performs as if it executes the following code fragment:

```cpp
const ctype<charT>& ctype = use_facet<ctype<charT> >(is.getloc());
if (ctype.is(ctype.space, c) != 0)
    // c is a whitespace character.
```

If, after any preparation is completed, `is.good()` is `true`, `ok_ != false` otherwise, `ok_ == false`.

During preparation, the constructor may call `setstate(failbit)` (which may throw `ios_base::failure (27.5.4.3)`)

```
~sentry();
```

Effects: None.

```cpp
explicit operator bool() const;
```

Effects: Returns `ok_`.

### 27.7.1.2 Formatted input functions

#### 27.7.1.2.1 Common requirements

Each formatted input function begins execution by constructing an object of class `sentry` with the `noskipws` (second) argument `false`. If the `sentry` object returns `true`, when converted to a value of type `bool`, the function endeavors to obtain the requested input. If an exception is thrown during input then `ios::badbit` is turned on in `*this`'s error state. If `(exceptions()&badbit) != 0` then the exception is rethrown. In any case, the formatted input function destroys the `sentry` object. If no exception has been thrown, it returns `*this`.

#### 27.7.1.2.2 Arithmetic Extractors

```cpp
operator>>(unsigned short& val);
operator>>(unsigned int& val);
operator>>(long& val);
operator>>(unsigned long& val);
operator>>(long long& val);
operator>>(float& val);
operator>>(double& val);
operator>>(long double& val);
operator>>(bool& val);
operator>>(void*& val);
```

As in the case of the inserters, these extractors depend on the locale’s `num_get<>` (22.4.2.1) object to perform parsing the input stream data. These extractors behave as formatted input functions (as described in 27.7.1.2.1). After a `sentry` object is constructed, the conversion occurs as if performed by the following code fragment:

---

317) The `sentry` constructor and destructor can also perform additional implementation-dependent operations.

318) This is done without causing an `ios::failure` to be thrown.
typedef num_get< charT, istreambuf_iterator<charT,traits> > numget;
    iostate err = iostate::goodbit;
    use_facet< numget >(loc).get(*this, 0, *this, err, val);
    setstate(err);

In the above fragment, loc stands for the private member of the basic_ios class. [Note: The first argument provides an object of the istreambuf_iterator class which is an iterator pointed to an input stream. It bypasses istreams and uses streambufs directly. — end note] Class locale relies on this type as its interface to istream, so that it does not need to depend directly on istream.

    operator>>(short& val);

2 The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

    typedef num_get< charT, istreambuf_iterator<charT,traits> > numget;
    iostate err = ios_base::goodbit;
    long lval;
    use_facet<numget>(loc).get(*this, 0, *this, err, lval);
    if (lval < numeric_limits<short>::min()) {
        err |= ios_base::failbit;
        val = numeric_limits<short>::min();
    } else if (numeric_limits<short>::max() < lval) {
        err |= ios_base::failbit;
        val = numeric_limits<short>::max();
    } else
        val = static_cast<short>(lval);
    setstate(err);

    operator>>(int& val);

3 The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

    typedef num_get< charT, istreambuf_iterator<charT,traits> > numget;
    iostate err = ios_base::goodbit;
    long lval;
    use_facet<numget>(loc).get(*this, 0, *this, err, lval);
    if (lval < numeric_limits<int>::min()) {
        err |= ios_base::failbit;
        val = numeric_limits<int>::min();
    } else if (numeric_limits<int>::max() < lval) {
        err |= ios_base::failbit;
        val = numeric_limits<int>::max();
    } else
        val = static_cast<int>(lval);
    setstate(err);

### 27.7.1.2.3 basic_istream::operator>>

basic_istream<charT,traits>& operator>>(
    basic_istream<charT,traits>& (*pf)(basic_istream<charT,traits>&&))

1   **Effects:** None. This extractor does not behave as a formatted input function (as described in 27.7.1.2.1.)
Returns: `pf(*this)`.

```cpp
basic_istream<charT,traits>& operator>>(
    basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&&);
```

**Effects:** Calls `pf(*this)`. This extractor does not behave as a formatted input function (as described in 27.7.1.2.1).

Returns: `*this`.

```cpp
basic_istream<charT,traits>& operator>>(
    (ios_base& (*pf)(ios_base&));
```

**Effects:** Calls `pf(*this)`.

This extractor does not behave as a formatted input function (as described in 27.7.1.2.1).

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 27.7.1.2.1) 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` (27.5.4.3).

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);
```

---

319) See, for example, the function signature `ws(basic_istream&)` (27.7.1.4).
320) See, for example, the function signature `dec(ios_base&)` (27.5.5.3).
template<class traits>
    basic_istream<char,traits>& operator>>(basic_istream<char,traits>& in,
        signed char& c);

    Effects: Behaves like a formatted input member (as described in 27.7.2.1) of in. After a sentry object is constructed a character is extracted from in, if one is available, and stored in c. Otherwise, the function calls in.setstate(failbit).

    Returns: in.

basic_istream<charT,traits>& operator>>(
    basic_streambuf<charT,traits>* sb);

    Effects: Behaves as an unformatted input function (as described in 27.7.3, paragraph 1). If sb is null, calls setstate(failbit), which may throw ios_base::failure (27.5.4.3). After a sentry object is constructed, extracts characters from *this and inserts them in the output sequence controlled by sb. Characters are extracted and inserted until any of the following occurs:

    — end-of-file occurs on the input sequence;
    — inserting in the output sequence fails (in which case the character to be inserted is not extracted);
    — an exception occurs (in which case the exception is caught).

    If the function inserts no characters, it calls setstate(failbit), which may throw ios_base::failure (27.5.4.3). If it inserted no characters because it caught an exception thrown while extracting characters from *this and failbit is on in exceptions() (27.5.4.3), then the caught exception is rethrown.

    Returns: *this.

27.7.1.3 Unformatted input functions

Each unformatted input function begins execution by constructing an object of class sentry with the default argument noskipws (second) argument true. If the sentry object returns true, when converted to a value of type bool, the function endeavors to obtain the requested input. Otherwise, if the sentry constructor exits by throwing an exception or if the sentry object returns false, when converted to a value of type bool, the function returns without attempting to obtain any input. In either case the number of extracted characters is set to 0; unformatted input functions taking a character array of non-zero size as an argument shall also store a null character (using charT()) in the first location of the array. If an exception is thrown during input then ios::badbit is turned on321 in *this's error state. (Exceptions thrown from basic_ios<>::clear() are not caught or rethrown.) If (exceptions()&badbit) != 0 then the exception is rethrown. It also counts the number of characters extracted. If no exception has been thrown it ends by storing the count in a member object and returning the value specified. In any event the sentry object is destroyed before leaving the unformatted input function.

streamsize gcount() const;

    Effects: None. This member function does not behave as an unformatted input function (as described in 27.7.1.3, paragraph 1).

    Returns: The number of characters extracted by the last unformatted input member function called for the object.

int_type get();

321) This is done without causing an ios::failure to be thrown.
Effects: Behaves as an unformatted input function (as described in 27.7.1.3, paragraph 1). After constructing a sentry object, extracts a character \( c \), if one is available. Otherwise, the function calls `setstate(failbit)`, which may throw `ios_base::failure` (27.5.4.3).

Returns: \( c \) if available, otherwise `traits::eof()`.

```cpp
docstr_c::get(char_type& c);
```

Effects: Behaves as an unformatted input function (as described in 27.7.1.3, 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` (27.5.4.3)).

Returns: `*this`.

```cpp
docstr_c::get(char_type* s, streamsize n,
              char_type delim);
```

Effects: Behaves as an unformatted input function (as described in 27.7.1.3, 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` (27.5.4.3)). 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`.

```cpp
docstr_c::get(char_type* s, streamsize n)
```

Effects: Calls `get(s,n,widen('\n'))`

Returns: Value returned by the call.

```cpp
docstr_c::get(basic_streambuf<char_type,traits>& sb,
              char_type delim);
```

Effects: Behaves as an unformatted input function (as described in 27.7.1.3, 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` (27.5.4.3).

Returns: `*this`.

322) Note that this function is not overloaded on types `signed char` and `unsigned char`.

323) Note that this function is not overloaded on types `signed char` and `unsigned char`.

§ 27.7.1.3
basic_istream<charT,traits>& get(basic_streambuf<char_type,traits>& sb);

Effects: Calls get(sb, widen(‘\n’))

Returns: Value returned by the call.

basic_istream<charT,traits>& getline(char_type* s, streamsize n,
  char_type delim);

Effects: Behaves as an unformatted input function (as described in §27.7.1.3, 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.324 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);325
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.326 If the function extracts no characters, it calls setstate(failbit) (which may throw ios_base::failure(27.5.4.3)).327 In any case, if n is greater than zero, it then stores a null character (using charT()) into the next successive location of the array.

Returns: *this.

Example:

```cpp
#include <iostream>

int main() {
    using namespace std;
    const int line_buffer_size = 100;

    char buffer[line_buffer_size];
    int line_number = 0;
    while (cin.getline(buffer, line_buffer_size, ‘\n’) || cin.gcount()) {
        int count = cin.gcount();
        if (cin.eof())
            cout << "Partial final line";  // cin.fail() is false
        else if (cin.fail()) {
            cout << "Partial long line";
            cin.clear(cin.rdstate() & ~ios_base::failbit);
        } else {
            count--;
            // Don’t include newline in count
            cout << "Line " << ++line_number;
        }
        cout << "(" << count << " chars): " << buffer << endl;
    }
}
```

324) Note that this function is not overloaded on types signed char and unsigned char.
325) Since the final input character is “extracted,” it is counted in the gcount(), even though it is not stored.
326) This allows an input line which exactly fills the buffer, without setting failbit. This is different behavior than the historical AT&T implementation.
327) This implies an empty input line will not cause failbit to be set.

§ 27.7.1.3

1017
basic_istream<charT,traits>& getline(char_type* s, streamsize n);

24 Returns: getline(s,n,widen('
'))

baseline_istream<charT,traits>&
    ignore(streamsize n = 1, int_type delim = traits::eof());

25 Effects: Behaves as an unformatted input function (as described in 27.7.1.3, 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() (18.3.1), 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 (27.5.4.3));
   — traits::eq_int_type(traits::to_int_type(c), delim) for the next available input character c (in which case c is extracted).

26 Remarks: The last condition will never occur if traits::eq_int_type(delim, traits::eof()).

Returns: *this.

int_type peek();

28 Effects: Behaves as an unformatted input function (as described in 27.7.1.3, paragraph 1). After constructing a sentry object, reads but does not extract the current input character.

29 Returns: traits::eof() if good() is false. Otherwise, returns rdbuf()->sgetc().

baseline_istream<charT,traits>& read(char_type* s, streamsize n);

30 Effects: Behaves as an unformatted input function (as described in 27.7.1.3, paragraph 1). After constructing a sentry object, if !good() calls setstate(failbit) which may throw an exception, and return. Otherwise extracts characters and stores them into successive locations of an array whose first element is designated by s. Characters are extracted and stored until either of the following occurs:
   — n characters are stored;
   — end-of-file occurs on the input sequence (in which case the function calls setstate(failbit|eofbit), which may throw ios_base::failure (27.5.4.3)).

31 Returns: *this.

streamsize readsome(char_type* s, streamsize n);

32 Effects: Behaves as an unformatted input function (as described in 27.7.1.3, 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 (27.5.4.3)), and extracts no characters;
   — If rdbuf()->in_avail() == 0, extracts no characters
   — If rdbuf()->in_avail() > 0, extracts min(rdbuf()->in_avail(),n)).

328) Note that this function is not overloaded on types signed char and unsigned char.

§ 27.7.1.3
Returns: The number of characters extracted.

basic_istream<

Effects: Behaves as an unformatted input function (as described in §27.7.1.3, 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 (27.5.4.3)). [ 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 §27.7.1.3, 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 rd buf() ->sungetc(). If rdbuf() is null, or if sungetc() returns traits::eof(), calls setstate(badbit) (which may throw ios_base::failure (27.5.4.3)). [ 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 §27.7.1.3, 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 (27.5.4.3), and returns -1. Otherwise, returns zero.

pos_type tellg();

Effects: Behaves as an unformatted input function (as described in §27.7.1.3, paragraph 1), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to gcount().

Returns: After constructing a sentry object, if fail() != false, returns pos_type(-1) to indicate failure. Otherwise, returns rdbuf()->pubseekoff(0, cur, in).

basic_istream<

Effects: Behaves as an unformatted input function (as described in §27.7.1.3, paragraph 1), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to gcount(). After constructing a sentry object, if fail() != true, executes rdbuf()->pubseekpos(pos, ios_base::in). In case of failure, the function calls setstate(failbit) (which may throw ios_base::failure).

Returns: *this.

basic_istream<

Effects: Behaves as an unformatted input function (as described in §27.7.1.3, 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).

Returns: *this.

27.7.1.4 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 27.7.1.3, paragraph 1), except
that it does not count the number of characters extracted and does not affect the value returned by
subsequent calls to is.gcount(). After constructing a sentry object extracts characters as long as the next
available character c is whitespace or until there are no more characters in the sequence. Whitespace
characters are distinguished with the same criterion as used by sentry::sentry (27.7.1.1.3). If ws
stops extracting characters because there are no more available it sets eofbit, but not failbit.

Returns: is.

27.7.1.5 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);
        virtual ~basic_iostream();

    protected:
        basic_iostream(basic_iostream&& rhs);

        // assign/swap
        basic_iostream& operator=(basic_iostream&& rhs);
        void swap(basic_iostream& rhs);
    };
}

1 The class basic_iostream inherits a number of functions that allow reading input and writing output to
sequences controlled by a stream buffer.

27.7.1.5.1 basic_iostream constructors

explicit basic_iostream(basic_streambuf<charT,traits>* sb);

§ 27.7.1.5.1
**Effects:** Constructs an object of class `basic_iostream`, assigning initial values to the base classes by calling `basic_istream<charT,traits>(sb)` (27.7.1.1) and `basic_ostream<charT,traits>(sb)` (27.7.2.1). 

**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)`.
// 27.7.2.2 Constructor/destructor:
explicit basic_ostream(basic_streambuf<char_type,traits>* sb);
virtual ~basic_ostream();

// 27.7.2.4 Prefix/suffix:
class sentry;

// 27.7.2.6 Formatted output:
basic_ostream<charT,traits>& operator<<(*
    basic_ostream<charT,traits>& (*pf)(basic_ostream<charT,traits>&&));
basic_ostream<charT,traits>& operator<<(*
    basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&&));
basic_ostream<charT,traits>& operator<<(*
    ios_base& (*pf)(ios_base&));

basic_ostream<charT,traits>& operator<<(bool n);
basic_ostream<charT,traits>& operator<<(short n);
basic_ostream<charT,traits>& operator<<(unsigned short n);
basic_ostream<charT,traits>& operator<<(int n);
basic_ostream<charT,traits>& operator<<(unsigned int n);
basic_ostream<charT,traits>& operator<<(long n);
basic_ostream<charT,traits>& operator<<(unsigned long n);
basic_ostream<charT,traits>& operator<<(long long n);
basic_ostream<charT,traits>& operator<<(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);

// 27.7.2.7 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();

// 27.7.2.5 seeks:
pos_type tellp();
basic_ostream<charT,traits>& seekp(pos_type);
basic_ostream<charT,traits>& seekp(off_type, ios_base::seekdir);

protected:
basic_ostream(basic_ostream&& rhs);

// 27.7.2.3 Assign/swap
basic_ostream& operator=(basic_ostream&& rhs);
void swap(basic_ostream& rhs);

};

// 27.7.2.6.4 character inserters
template<class charT, class traits>
basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&, charT);

template<class charT, class traits>
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.

27.7.2.2 `basic_ostream` constructors

```cpp
explicit basic_ostream(basic_streambuf<charT,traits>* sb);
```

**Effects:** Constructs an object of class `basic_ostream`, assigning initial values to the base class by calling `basic_ios<charT,traits>::init(sb)` (27.5.4.1).

**Postcondition:** `rdbuf() == sb`.

```cpp
virtual ~basic_ostream();
```
Effects: Destroys an object of class basic_ostream.

Remarks: Does not perform any operations on rdbuf().

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.

27.7.2.3 Class basic_ostream assign and swap

basic_ostream& operator=(basic_ostream&& rhs);

Effects: swap(rhs).

Returns: *this.

void swap(basic_ostream& rhs);

Effects: Calls basic_ios<CharT, traits>::swap(rhs).

27.7.2.4 Class basic_ostream::sentry

namespace std {
    template <class CharT, class Traits = char_traits<CharT> >
    class basic_ostream<CharT, Traits>::sentry {
        // 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().\(^{329}\)

If, after any preparation is completed, os.good() is true, ok_ == true otherwise, ok_ == false. During preparation, the constructor may call setstate(failbit) (which may throw ios_base::failure (27.5.4.3))\(^{330}\) "sentry();

If (os.flags() & ios_base::unitbuf) && !uncaught_exception() && os.good()) is true, calls os.rdbuf()->pubsync(). If that function returns -1, sets badbit in os.rdstate() without propagating an exception.

Throws: Nothing.

\(^{329}\) The call os.tie()->flush() does not necessarily occur if the function can determine that no synchronization is necessary.

\(^{330}\) The sentry constructor and destructor can also perform additional implementation-dependent operations.
explicit operator bool() const;

Effects: Returns ok_.

27.7.2.5 basic_ostream seek members

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).

Returns: *this.

27.7.2.6 Formatted output functions

27.7.2.6.1 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_base::failbit), which might throw an exception. If an exception is thrown during output, then ios::badbit is turned on in *this's error state. If (exceptions()&badbit) != 0 then the exception is rethrown. Whether or not an exception is thrown, the sentry object is destroyed before leaving the formatted output function. If no exception is thrown, the result of the formatted output function is *this.

The descriptions of the individual formatted output operations describe how they perform output and do not mention the sentry object.

27.7.2.6.2 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<<(float val);
operator<<(double val);
operator<<(long double val);
operator<<(const void* val);

§ 27.7.2.6.2
Effects: The classes `num_get<>` and `num_put<>` handle locale-dependent numeric formatting and parsing. These inserter functions use the imbeded `locale` value to perform numeric formatting. When `val` is of type `bool, long, unsigned long, long long, unsigned long long, double, long double, or const void*`, the formatting conversion occurs as if it performed the following code fragment:

```cpp
bool failed = use_facet<
    num_put<charT,ostreambuf_iterator<charT,traits> >
>(getloc()).put(*this, *this, fill(), val).failed();
```

When `val` is of type `short` the formatting conversion occurs as if it performed the following code fragment:

```cpp
ios_base::fmtflags baseflags = ios_base::flags() & ios_base::basefield;
bool failed = use_facet<
    num_put<charT,ostreambuf_iterator<charT,traits> >
>(getloc()).put(*this, *this, fill(),
              baseflags == ios_base::oct || baseflags == ios_base::hex
    ? static_cast<long>(static_cast<unsigned short>(val))
    : static_cast<long>(val)).failed();
```

When `val` is of type `int` the formatting conversion occurs as if it performed the following code fragment:

```cpp
ios_base::fmtflags baseflags = ios_base::flags() & ios_base::basefield;
bool failed = use_facet<
    num_put<charT,ostreambuf_iterator<charT,traits> >
>(getloc()).put(*this, *this, fill(),
              baseflags == ios_base::oct || baseflags == ios_base::hex
    ? static_cast<long>(static_cast<unsigned int>(val))
    : static_cast<long>(val)).failed();
```

When `val` is of type `unsigned short` or `unsigned int` the formatting conversion occurs as if it performed the following code fragment:

```cpp
bool failed = use_facet<
    num_put<charT,ostreambuf_iterator<charT,traits> >
>(getloc()).put(*this, *this, fill(),
              static_cast<unsigned long>(val)).failed();
```

When `val` is of type `float` the formatting conversion occurs as if it performed the following code fragment:

```cpp
bool failed = use_facet<
    num_put<charT,ostreambuf_iterator<charT,traits> >
>(getloc()).put(*this, *this, fill(),
              static_cast<double>(val)).failed();
```

The first argument provides an object of the `ostreambuf_iterator<>` class which is an iterator for class `basic_ostream<>`. It bypasses streams and uses streambufs directly. Class `locale` relies on these types as its interface to iostreams, since for flexibility it has been abstracted away from direct dependence on `ostream`. The second parameter is a reference to the base subobject of type `ios_base`. It provides formatting specifications such as field width, and a locale from which to obtain other facets.

If `failed` is `true` then does `setstate(badbit)`, which may throw an exception, and returns.

Returns: `*this`. 

§ 27.7.2.6.2
### 27.7.2.6.3 basic_ostream::operator<<

basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& (*pf)(basic_ostream<charT,traits>&))

1. **Effects:** None. Does not behave as a formatted output function (as described in 27.7.2.6.1).
2. **Returns:** pf(*this).

basic_ostream<charT,traits>& operator<<(basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&))

3. **Effects:** Calls pf(*this). This inserter does not behave as a formatted output function (as described in 27.7.2.6.1).
4. **Returns:** *this.

basic_ostream<charT,traits>& operator<<(ios_base& (*pf)(ios_base&))

5. **Effects:** Calls pf(*this). This inserter does not behave as a formatted output function (as described in 27.7.2.6.1).
6. **Returns:** *this.

basic_ostream<charT,traits>& operator<<(basic_streambuf<charT,traits>* sb);

7. **Effects:** Behaves as an unformatted output function (as described in 27.7.2.7, paragraph 1). After the sentry object is constructed, if sb is null calls setstate(badbit) (which may throw ios_base::failure).
8. 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.
9. If the function inserts no characters, it calls setstate(failbit) (which may throw ios_base::failure (27.5.4.3)). 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.
10. **Returns:** *this.

### 27.7.2.6.4 Character inserter function templates

```cpp
template<class charT, class traits>
basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& out, charT c);
```

```cpp
template<class charT, class traits>
basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& out, char c);
```

// specialization

---

332) See, for example, the function signature endl(basic_ostream&) (27.7.2.8).
333) See, for example, the function signature dec(ios_base&) (27.5.5.3).
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
      char c);

  // signed and unsigned
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
      signed char c);

template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
      unsigned char c);

Effects: Behaves like a formatted inserter (as described in 27.7.2.6.1) of out. After a sentry object is constructed it inserts characters. In case c has type char and the character type of the stream is not char, then the character to be inserted is out.widen(c); otherwise the character is c. Padding is determined as described in 22.4.2.2.2. `width(0)` is called. The insertion character and any required padding are inserted into out.

Returns: out.

template<class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& out,
      const charT* s);

template<class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& out,
      const char* s);

template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
      const char* s);

template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
      const signed char* s);

template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
      const unsigned char* s);

Requires: s shall not be a null pointer.

Effects: Behaves like a formatted inserter (as described in 27.7.2.6.1) 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 22.4.2.2.2. The n characters starting at s are widened using `out.widen` (27.5.4.2). The widened characters and any required padding are inserted into out. Calls `width(0)`.

Returns: out.
27.7.2.7 Unformatted output functions

1 Each unformatted output function begins execution by constructing an object of class `sentry`. If this object returns `true`, while converting to a value of type `bool`, the function endeavors to generate the requested output. If an exception is thrown during output, then `ios::badbit` is turned on\(^\text{334}\) 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.

```cpp
basic_ostream<charT,traits>& put(char_type c);
```

2 **Effects:** Behaves as an unformatted output function (as described in 27.7.2.7, paragraph 1). After constructing a sentry object, inserts the character `c`, if possible.\(^\text{335}\) Otherwise, calls `setstate(badbit)` (which may throw `ios_base::failure` (27.5.4.3)).

3 **Returns:** `*this`.

```cpp
basic_ostream& write(const char_type* s, streamsize n);
```

4 **Effects:** Behaves as an unformatted output function (as described in 27.7.2.7, paragraph 1). After constructing a sentry object, obtains characters to insert from successive locations of an array whose first element is designated by `s`.\(^\text{336}\) Characters are inserted until either of the following occurs:

- `n` characters are inserted;
- inserting in the output sequence fails (in which case the function calls `setstate(badbit)` , which may throw `ios_base::failure` (27.5.4.3)).

5 **Returns:** `*this`.

```cpp
basic_ostream& flush();
```

6 **Effects:** Behaves as an unformatted output function (as described in 27.7.2.6.1, 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` (27.5.4.3)). Otherwise, if the sentry object returns `false`, does nothing.

7 **Returns:** `*this`.

27.7.2.8 Standard basic_ostream manipulators

```cpp
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`.

```cpp
namespace std {
  template <class charT, class traits>
}
```

§ 27.7.2.8
basic_ostream<charT,traits>& ends(basic_ostream<charT,traits>&& os);
}

Effects: Inserts a null character into the output sequence: calls os.put(charT()).

Returns: os.

namespace std {
  template <class charT, class traits>
  basic_ostream<charT,traits>& flush(basic_ostream<charT,traits>& os);
}

Effects: Calls os.flush().

Returns: os.

27.7.2.9 Rvalue stream insertion

[ostream.rvalue]

template <class charT, class traits, class T>
basic_ostream<charT,traits>&
  operator<<(basic_ostream<charT,traits>&& os, const T& x);

Effects: os << x

Returns: os

27.7.3 Standard manipulators

[std.manip]

The header <iomanip> defines several functions that support extractors and inserters that alter information maintained by class ios_base and its derived classes.

unspecified resetiosflags(ios_base::fmtflags mask);

Returns: An object of unspecified type such that if out is an object of type basic_ostream<charT,traits> then the expression out <<resetiosflags(mask) behaves as if it called f(out, mask), or if in is an object of type basic_istream<charT, traits> then the expression in >>resetiosflags(mask) behaves as if it called f(in, mask), where the function f is defined as:

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:

337) 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).

§ 27.7.3
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`.

```cpp
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`.

```cpp
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`.

```cpp
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`.

§ 27.7.3
unspecified setw(int n);

7      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:

void f(ios_base& str, int n) {
    // set width
    str.width(n);
}

The expression out <<setw(n) shall have type basic_ostream<\charT, traits>& and value out. The expression in >>setw(n) shall have type basic_istream<\charT, traits>& and value in.

27.7.4 Extended Manipulators [ext.manip]

1 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.

template <class moneyT> unspecified get_money(moneyT& mon, bool intl = false);

2 Requires: The type moneyT shall be either long double or a specialization of the basic_string template (Clause 21).

3 Effects: The expression in >> get_money(mon, intl) described below behaves as a formatted input function (27.7.1.2.1).

4 Returns: An object of unspecified type such that if in is an object of type basic_istream<\charT, traits> then the expression in >>get_money(mon, intl) behaves as if it called f(in, mon, intl), where the function f is defined as:

template <class charT, class traits, class moneyT>
void f(basic_ios<charT, traits>& str, moneyT& mon, bool intl) {
    typedef istreambuf_iterator<charT, traits> Iter;
    typedef money_get<charT, Iter> MoneyGet;
    ios_base::iostate err = ios_base::goodbit;
    const MoneyGet &mg = use_facet<MoneyGet>(str.getloc());
    mg.get(Iter(str.rdbuf()), Iter(), intl, str, err, mon);
    if (ios_base::goodbit != err)
        str.setstate(err);
}

The expression in >>get_money(mon, intl) shall have type basic_istream<\charT, traits>& and value in.

template <class moneyT> unspecified put_money(const moneyT& mon, bool intl = false);

5 Requires: The type moneyT shall be either long double or a specialization of the basic_string template (Clause 21).

6 Returns: An object of unspecified type such that if out is an object of type basic_ostream<\charT, traits> then the expression out <<put_money(mon, intl) behaves as a formatted input function that calls f(out, mon, intl), where the function f is defined as:

§ 27.7.4
template <class charT, class traits, class moneyT>
void f(basic_ios<charT, traits>& str, const moneyT& mon, bool intl) {
    typedef ostreambuf_iterator<charT, traits> Iter;
    typedef money_put<charT, Iter> MoneyPut;
    const MoneyPut& mp = use_facet<MoneyPut>(str.getloc());
    const Iter end = mp.put(Iter(str.rdbuf()), intl, str, str.fill(), mon);
    if (end.failed())
        str.setstate(ios::badbit);
}

The expression out <<put_money(mon, intl) shall have type basic_ostream<charT, traits>& and value out.

template <class charT> unspecified get_time(struct tm* tmb, const charT* fmt);

Requires: The argument tmb shall be a valid pointer to an object of type struct tm, and the argument fmt shall be a valid pointer to an array of objects of type charT with char_traits<charT>::length(fmt) elements.

Returns: An object of unspecified type such that if in is an object of type basic_istream<charT, traits>, then the expression in >>get_time(tmb, fmt) behaves as if it called f(in, tmb, fmt), where the function f is defined as:

    template <class charT, class traits>
    void f(basic_istream<charT, traits>& str, struct tm* tmb, const charT* fmt) {
        typedef istreambuf_iterator<charT, traits> Iter;
        typedef time_get<charT, Iter> TimeGet;
        ios_base::iostate err = ios_base::goodbit;
        const TimeGet& tg = use_facet<TimeGet>(str.getloc());
        tg.get(Iter(str.rdbuf())), Iter(), str, err, tmb,
        fmt, fmt + traits::length(fmt));
        if (err != ios_base::goodbit)
            str.setstate(err);
    }

The expression in >>get_time(tmb, fmt) shall have type basic_istream<charT, traits>& and value in.

template <class charT> unspecified put_time(const struct tm* tmb, const charT* fmt);

Requires: The argument tmb shall be a valid pointer to an object of type struct tm, and the argument fmt shall be a valid pointer to an array of objects of type charT with char_traits<charT>::length(fmt) elements.

Returns: An object of unspecified type such that if out is an object of type basic_ostream<charT, traits>, then the expression out <<put_time(tmb, fmt) behaves as if it called f(out, tmb, fmt), where the function f is defined as:

    template <class charT, class traits>
    void f(basic_istream<charT, traits>& str, const struct tm* tmb, const charT* fmt) {
        typedef ostreambuf_iterator<charT, traits> Iter;

§ 27.7.4 1033
typedef time_put<charT, Iter> TimePut;

const TimePut& tp = use_facet<TimePut>(str.getloc());
const Iter end = tp.put(Iter(str.rdbuf()), str, str.fill(), tmb, 
    fmt, fmt + traits::length(fmt));

if (end.failed())
    str.setstate(ios_base::badbit);
}

The expression `out << put_time(tmb, fmt)` shall have type `basic_istream<charT, traits>&` and value `out`.

### 27.8 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 21.3.

#### Header `<sstream>` synopsis

```cpp
namespace std {
    template <class charT, class traits = char_traits<charT>,
              class Allocator = allocator<charT> >
        class basic_stringbuf;
    typedef basic_stringbuf<char> stringbuf;
typedef basic_stringbuf<wchar_t> wstringbuf;

template <class charT, class traits = char_traits<charT>,
              class Allocator = allocator<charT> >
        class basic_istringstream;
    typedef basic_istringstream<char> istringstream;
    typedef basic_istringstream<wchar_t> wistringstream;

template <class charT, class traits = char_traits<charT>,
              class Allocator = allocator<charT> >
        class basic_ostringstream;
    typedef basic_ostringstream<char> ostringstream;
    typedef basic_ostringstream<wchar_t> wostringstream;

template <class charT, class traits = char_traits<charT>,
              class Allocator = allocator<charT> >
        class basic_stringstream;
    typedef basic_stringstream<char> stringstream;
    typedef basic_stringstream<wchar_t> wstringstream;
}
```

### 27.8.1 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;

// 27.8.1.1 Constructors:
explicit basic_stringbuf(ios_base::openmode which = ios_base::in | ios_base::out);
explicit basic_stringbuf(const basic_string<charT,traits,Allocator>& str, ios_base::openmode which = ios_base::in | ios_base::out);
basic_stringbuf(basic_stringbuf&& rhs);

// 27.8.1.2 Assign and swap:
basic_stringbuf& operator=(basic_stringbuf&& rhs);
void swap(basic_stringbuf & rhs);

// 27.8.1.3 Get and set:
basic_string<charT,traits,Allocator> str() const;
void str(const basic_string<charT,traits,Allocator>& s);

protected:
// 27.8.1.4 Overridden virtual functions:
virtual int_type underflow();
virtual int_type pbackfail(int_type c = traits::eof());
virtual int_type overflow (int_type c = traits::eof());
virtual basic_streambuf<charT,traits>* setbuf(charT*, streamsize);

virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode which = ios_base::in | ios_base::out);
virtual pos_type seekpos(pos_type sp, ios_base::openmode which = ios_base::in | ios_base::out);

private:
ios_base::openmode mode; // exposition only
};

template <class charT, class traits, class Allocator> void swap(basic_stringbuf<charT, traits, Allocator>& x, basic_stringbuf<charT, traits, Allocator>& y);

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.
27.8.1.1 basic_stringbuf constructors

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() (27.6.2.1), 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() (27.6.2.1), and initializing mode with which. Then calls str(s).

basic_stringbuf(basic_stringbuf&& rhs);

Effects: Move constructs from the rvalue rhs. It is implementation-defined whether the sequence pointers in *this (eback(), gptr(), egptr(), pbase(), pptr(), epptr()) obtain the values which rhs had. Whether they do or not, *this and rhs reference separate buffers (if any at all) after the construction. The openmode, locale and any other state of rhs is also copied.

Postconditions: Let rhs_p refer to the state of rhs just prior to this construction and let rhs_a refer to the state of rhs just after this construction.

- str() == rhs_p.str()
- gptr() - eback() == rhs_p.gptr() - rhs_p.eback()
- egptr() - eback() == rhs_p.egptr() - rhs_p.eback()
- pptr() - pbase() == rhs_p.pptr() - rhs_p.pbase()
- epptr() - pbase() == rhs_p.epptr() - rhs_p.pbase()
- if (eback()) eback() != rhs_a.eback()
- if (gptr()) gptr() != rhs_a.gptr()
- if (egptr()) egptr() != rhs_a.egptr()
- if (pbase()) pbase() != rhs_a.pbase()
- if (pptr()) pptr() != rhs_a.pptr()
- if (epptr()) epptr() != rhs_a.epptr()

27.8.1.2 Assign and swap

basic_stringbuf& operator=(basic_stringbuf&& rhs);

Effects: After the move assignment *this has the observable state it would have had if it had been move constructed from rhs (see 27.8.1.1).

Returns: *this.

void swap(basic_stringbuf& rhs);

Effects: Exchanges the state of *this and rhs.
template <class charT, class traits, class Allocator>
void swap(basic_stringbuf<charT, traits, Allocator>& x,
        basic_stringbuf<charT, traits, Allocator>& y);

Effects: x.swap(y).

### 27.8.1.3 Member functions

- **basic_string<
    charT, traits, Allocator> str() const**

  Returns: A basic_string object whose content is equal to the basic_stringbuf underlying character sequence. If the basic_stringbuf was created only in input mode, the resultant basic_string contains the character sequence in the range [eback(), egptr()). If the basic_stringbuf was created with which & ios_base::out being true then the resultant basic_string contains the character sequence in the range [pbase(), high_mark), where high_mark represents the position one past the highest initialized character in the buffer. Characters can be initialized by writing to the stream, by constructing the basic_stringbuf with a basic_string, or by calling the str(basic_string) member function. In the case of calling the str(basic_string) member function, all characters initialized prior to the call are now considered uninitialized (except for those characters re-initialized by the new basic_string). Otherwise the basic_stringbuf has been created in neither input nor output mode and a zero length basic_string is returned.

- **void str(const basic_string<
    charT, traits, Allocator>& s);**

  Effects: Copies the content of s into the basic_stringbuf underlying character sequence and initializes the input and output sequences according to mode.

  Postconditions: If mode & ios_base::out is true, pbase() points to the first underlying character and pptr() >= 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.

### 27.8.1.4 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.

§ 27.8.1.4
— If \( \text{traits::eq_int_type(c,traits::eof())} \) returns \text{true} and if the input sequence has a put-back position available, assigns \( \text{gptr()} - 1 \) to \( \text{gptr()} \).

    Returns: \text{traits::not_eof(c)}.

3

Returns: \text{traits::eof()} to indicate failure.

4

Remarks: If the function can succeed in more than one of these ways, it is unspecified which way is chosen.

\[
\text{int_type overflow(int_type c = traits::eof());}
\]

5

Effects: Appends the character designated by \( c \) to the output sequence, if possible, in one of two ways:

    — If \( \text{traits::eq_int_type(c,traits::eof())} \) returns \text{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 \( \text{sputc(c)} \).

    Signals success by returning \( c \).

    — If \( \text{traits::eq_int_type(c,traits::eof())} \) returns \text{true}, there is no character to append.

    Signals success by returning a value other than \text{traits::eof()}.

6

Remarks: The function can alter the number of write positions available as a result of any call.

7

Returns: \text{traits::eof()} to indicate failure.

8

The function can make a write position available only if \( (\text{mode} \& \text{ios_base::out}) \neq 0 \). To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements to hold the current array object (if any), plus at least one additional write position. If \( (\text{mode} \& \text{ios_base::in}) \neq 0 \), the function alters the read end pointer \( \text{egptr()} \) to point just past the new write position.

\[
\text{pos_type seekoff(off_type off, ios_base::seekdir way,}
    \text{ios_base::openmode which}
    = \text{ios_base::in | ios_base::out);}
\]

9

Effects: Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 127.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (\text{which} &amp; \text{ios_base::in}) == \text{ios_base::in} )</td>
<td>positions the input sequence</td>
</tr>
<tr>
<td>( (\text{which} &amp; \text{ios_base::out}) == \text{ios_base::out} )</td>
<td>positions the output sequence</td>
</tr>
</tbody>
</table>
| \( (\text{which} \& (\text{ios_base::in | ios_base::out}) == (\text{ios_base::in |}
| and way == either \text{ios_base::beg or} \text{ios_base::end}          | positions both the input and the output sequences |
| Otherwise                                                                 | the positioning operation fails. |
For a sequence to be positioned, if its next pointer (either \texttt{gptr()} or \texttt{pptr()}) is a null pointer and the new offset \texttt{newoff} is nonzero, the positioning operation fails. Otherwise, the function determines \texttt{newoff} as indicated in Table 128.

<table>
<thead>
<tr>
<th>Condition</th>
<th>newoff Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{way == ios_base::beg}</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{way == ios_base::cur}</td>
<td>the next pointer minus the beginning pointer (xnext - xbeg).</td>
</tr>
<tr>
<td>\texttt{way == ios_base::end}</td>
<td>the high mark pointer minus the beginning pointer (high_mark - xbeg).</td>
</tr>
</tbody>
</table>

If \((\texttt{newoff} + \texttt{off}) < 0\), or if \texttt{newoff} + \texttt{off} refers to an uninitialized character (as defined in 27.8.1.3 paragraph 1), the positioning operation fails. Otherwise, the function assigns \texttt{xbeg} + \texttt{newoff} + \texttt{off} to the next pointer \texttt{xnext}.

\texttt{Returns: pos\_type(newoff)}, constructed from the resultant offset \texttt{newoff} (of type \texttt{off\_type}), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is \texttt{pos\_type(off\_type(-1))}.

\texttt{pos\_type seekpos(pos\_type sp, ios\_base::openmode which = ios\_base::in | ios\_base::out);}

\textit{Effects:} Equivalent to \texttt{seekoff(off\_type(sp), ios\_base::beg, which)}.

\texttt{Returns: sp} to indicate success, or \texttt{pos\_type(off\_type(-1))} to indicate failure.

\texttt{basic\_streambuf<char\_T, traits>\* setbuf(char\_T\* s, stream\_size n);}  

\textit{Effects:} implementation-defined, except that \texttt{setbuf(0,0)} has no effect.

\texttt{Returns: this.}

### 27.8.2 Class template basic\_istringstream

namespace std {
    template <class char\_T, class traits = char\_traits<char\_T>, class Allocator = allocator<char\_T> >
    class basic\_istringstream : public basic\_istream<char\_T, traits> {
    public:
        typedef char\_T char\_type;
        typedef typename traits::int\_type int\_type;
        typedef typename traits::pos\_type pos\_type;
        typedef typename traits::off\_type off\_type;
        typedef traits traits\_type;
        typedef Allocator allocator\_type;

        // 27.8.2.1 Constructors:
        explicit basic\_istringstream(ios\_base::openmode which = ios\_base::in);
        explicit basic\_istringstream(  
            const basic\_string<char\_T, traits, Allocator>& str,  
            ios\_base::openmode which = ios\_base::in);
        basic\_istringstream(basic\_istringstream\& rhs);
    }
}
// 27.8.2.2 Assign and swap:
basic_istringstream& operator=(basic_istringstream&& rhs);
void swap(basic_istringstream& rhs);

// 27.8.2.3 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.

27.8.2.1 basic_istringstream constructors

explicit basic_istringstream(ios_base::openmode which = ios_base::in);
Effects: Constructs an object of class basic_istringstream<charT, traits>, initializing the base class with basic_istream(&sb) and initializing sb with basic_stringbuf<charT, traits, Allocator>(which | ios_base::in)) (27.8.1.1).

explicit basic_istringstream(
    const basic_string<charT,traits,allocator>& str,
    ios_base::openmode which = ios_base::in);
Effects: Constructs an object of class basic_istringstream<charT, traits>, initializing the base class with basic_istream(&sb) and initializing sb with basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::in)) (27.8.1.1).

basic_istringstream(basic_istringstream&& rhs);
Effects: Move constructs from the rvalue rhs. This is accomplished by move constructing the base class, and the contained basic_stringbuf. Next basic_istream<charT,traits>::set_rdbuf(&sb) is called to install the contained basic_stringbuf.

27.8.2.2 Assign and swap

basic_istringstream& operator=(basic_istringstream&& rhs);
Effects: Move assigns the base and members of *this from the base and corresponding members of rhs.
Returns: *this.
void swap(basic_istringstream& rhs);

\textbf{Effects:} Exchanges the state of *this and rhs by calling \texttt{basic_istream<charT,traits>::swap(rhs)} and \texttt{sb.swap(rhs.sb)}.

\begin{verbatim}
template <class charT, class traits, class Allocator>
void swap(basic_istringstream<charT, traits, Allocator>& x, 
          basic_istringstream<charT, traits, Allocator>& y);
\end{verbatim}

\textbf{Effects:} x.swap(y).

\section*{27.8.2.3 Member functions}

\begin{verbatim}
basic_stringbuf<charT,traits,Allocator>* rdbuf() const;
\end{verbatim}

\textbf{Returns:} \texttt{const_cast<basic_stringbuf<charT,traits,Allocator>*&>(&sb)}.

\begin{verbatim}
basic_string<charT,traits,Allocator> str() const;
\end{verbatim}

\textbf{Returns:} \texttt{rdbuf()->str()}.  

\begin{verbatim}
void str(const basic_string<charT,traits,Allocator>& s);
\end{verbatim}

\textbf{Effects:} Calls \texttt{rdbuf()->str(s)}.

\section*{27.8.3 Class template basic_ostringstream}

\begin{verbatim}
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;

        // 27.8.3.1 Constructors/destructor:
        explicit basic_ostringstream(ios_base::openmode which = ios_base::out);
        explicit basic_ostringstream(
            const basic_string<charT,traits,Allocator>& str, 
            ios_base::openmode which = ios_base::out);
        basic_ostringstream(basic_ostringstream&& rhs);

        // 27.8.3.2 Assign/swap:
        basic_ostringstream& operator=(basic_ostringstream&& rhs);
        void swap(basic_ostringstream& rhs);

        // 27.8.3.3 Members:
        basic_stringbuf<charT,traits,Allocator>* rdbuf() const;

        basic_string<charT,traits,Allocator> str() const;
    }
}
\end{verbatim}
The class `basic_ostreamstream<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.

#### 27.8.3.1 `basic_ostreamstream` constructors

```cpp
template <class charT, class traits, class Allocator>
void swap(basic_ostreamstream<charT, traits, Allocator>& x,
          basic_ostreamstream<charT, traits, Allocator>& y);
```

1. The class `basic_ostreamstream<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.

#### 27.8.3.2 Assign and swap

```cpp
basic_ostreamstream& operator=(basic_ostreamstream&& rhs);
```

1. **Effects**: Move assigns the base and members of *this from the base and corresponding members of rhs.

```cpp
void swap(basic_ostreamstream& rhs);
```

2. **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_ostreamstream<charT, traits, Allocator>& x,
          basic_ostreamstream<charT, traits, Allocator>& y);
```
Effects: x.swap(y).

27.8.3.3 Member functions

basic_stringbuf<charT,traits,Allocator>* rdbuf() const;

Returns: const_cast<basic_stringbuf<charT,traits,Allocator>*>(&sb).

basic_string<charT,traits,Allocator> str() const;

Returns: rdbuf()->str().

void str(const basic_string<charT,traits,Allocator>& s);

Effects: Calls rdbuf()->str(s).

27.8.4 Class template basic_stringstream

namespace std {
    template <class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT> >
    class basic_stringstream :
        public basic_iostream<charT,traits> {
        public:

            // types:
            typedef charT char_type;
            typedef typename traits::int_type int_type;
            typedef typename traits::pos_type pos_type;
            typedef typename traits::off_type off_type;
            typedef traits traits_type;
            typedef Allocator allocator_type;

            // constructors/destructor
            explicit basic_stringstream(
                ios_base::openmode which = ios_base::out|ios_base::in);
            explicit basic_stringstream(
                const basic_string<charT,traits,Allocator>& str,
                ios_base::openmode which = ios_base::out|ios_base::in);
            basic_stringstream(basic_stringstream&& rhs);

            // 27.8.5.1 Assign/swap:
            basic_stringstream& operator=(basic_stringstream&& rhs);
            void swap(basic_stringstream& rhs);

            // Members:
            basic_stringbuf<charT,traits,Allocator>* rdbuf() const;
            basic_string<charT,traits,Allocator> str() const;
            void str(const basic_string<charT,traits,Allocator>& s);

            private:
                basic_stringbuf<charT, traits> sb; // exposition only
            };

    template <class charT, class traits, class Allocator>

§ 27.8.4
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.

### 27.8.5 `basic_stringstream` constructors

**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`.

### 27.8.5.1 Assign and swap

**basic_stringstream& operator=(basic_stringstream&& rhs);**

*Effects:* Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

*Returns:* `*this`.

**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);**

*Effects:* `x.swap(y)`.
27.8.6 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;

Returns: rdbuf()->str().

void str(const basic_string<charT,traits,Allocator>& str);

Effects: Calls rdbuf()->str(str).

27.9 File-based streams

27.9.1 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

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> (27.9.2).

[Note: The class template basic_filebuf treats a file as a source or sink of bytes. In an environment that uses a large character set, the file typically holds multibyte character sequences and the basic_filebuf object converts those multibyte sequences into wide character sequences. — end note]
typedef charT char_type;
typedef typename traits::int_type int_type;
typedef typename traits::pos_type pos_type;
typedef typename traits::off_type off_type;
typedef traits traits_type;

// 27.9.1.2 Constructors/destructor:
basic_filebuf();
basic_filebuf(basic_filebuf&& rhs);
virtual ~basic_filebuf();

// 27.9.1.3 Assign/swap:
basic_filebuf& operator=(basic_filebuf&& rhs);
void swap(basic_filebuf& rhs);

// 27.9.1.4 Members:
bool is_open() const;
basic_filebuf<charT,traits>* open(const char* s,
   ios_base::openmode mode);
basic_filebuf<charT,traits>* open(const string& s,
   ios_base::openmode mode);
basic_filebuf<charT,traits>* close();

protected:
// 27.9.1.5 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);
}

1 The class basic_filebuf<charT,traits> associates both the input sequence and the output sequence with a file.

2 The restrictions on reading and writing a sequence controlled by an object of class basic_filebuf<charT,traits> are the same as for reading and writing with the Standard C library FILEs.

3 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.

4 An instance of `basic_filebuf` behaves as described in 27.9.1.1 provided `traits::pos_type` is `fpos<traits::state_type>`. Otherwise the behavior is undefined.

5 In order to support file I/O and multibyte/wide character conversion, conversions are performed using members of a facet, referred to as a `codecvt` in following sections, obtained as if by

```cpp
const codecvt<charT,char,typename traits::state_type>& a_codecvt =
    use_facet<codecvt<charT,char,typename traits::state_type>>(getloc());
```

### 27.9.1.2 `basic_filebuf` constructors

#### [filebuf.cons]

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>basic_filebuf()</code></td>
<td><strong>Effects:</strong> Constructs an object of class <code>basic_filebuf&lt;charT,traits&gt;</code>, initializing the base class with <code>basic_streambuf&lt;charT,traits&gt;()</code> (27.6.2.1).  <strong>Postcondition:</strong> <code>is_open() == false</code>.</td>
</tr>
</tbody>
</table>
| `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 17.6.4.11).

### 27.9.1.3 Assign and swap

```cpp
basic_filebuf& operator=(basic_filebuf&& rhs);
```

**Effects:** Calls `this->close()` then move assigns from `rhs`. After the move assignment `*this` has the observable state it would have had if it had been move constructed from `rhs` (see 27.9.1.2).

**Returns:** `*this`.

```cpp
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);
```

**Effects:** `x.swap(y)`.

### 27.9.1.4 Member functions

```cpp
bool is_open() const;
```

**Returns:** `true` if a previous call to `open` succeeded (returned a non-null value) and there has been no intervening call to close.

```cpp
basic_filebuf<charT, traits>* open(const char* s,
                                    ios_base::openmode mode);
```

**Effects:** If `is_open() != false`, returns a null pointer. Otherwise, initializes the `filebuf` as required. It then opens a file, if possible, whose name is the NTBS `s` (as if by calling `std::fopen(s, modstr)`). The NTBS `modstr` is determined from `mode & ~ios_base::ate` as indicated in Table 129. If `mode` is not some combination of flags shown in the table then the open fails.

3 If the open operation succeeds and `(mode & ios_base::ate) != 0`, positions the file to the end (as if by calling `std::fseek(file, 0, SEEK_END)`).  \(^{338}\)

4 If the repositioning operation fails, calls `close()` and returns a null pointer to indicate failure.

**Returns:** `this` if successful, a null pointer otherwise.

```cpp
basic_filebuf<charT, traits>* open(const string& s,
                                    ios_base::openmode mode);
```

**Returns:** `open(s.c_str(), mode);`

```cpp
basic_filebuf<charT, traits>* close();
```

---

\(^{338}\) The macro `SEEK_END` is defined, and the function signatures `fopen(const char*, const char*)` and `fseek(FILE*, long, int)` are declared, in `<cstdio>` (27.9.2).
Table 129 — File open modes

<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></td>
</tr>
<tr>
<td>+</td>
<td>“w”</td>
</tr>
<tr>
<td>+</td>
<td>“a”</td>
</tr>
<tr>
<td>+</td>
<td>“a”</td>
</tr>
<tr>
<td>+</td>
<td>“w”</td>
</tr>
<tr>
<td>+</td>
<td>“r”</td>
</tr>
<tr>
<td>+</td>
<td>“r+”</td>
</tr>
<tr>
<td>+</td>
<td>“w+”</td>
</tr>
<tr>
<td>+</td>
<td>“a+”</td>
</tr>
<tr>
<td>+</td>
<td>“a+”</td>
</tr>
<tr>
<td>+</td>
<td>“wb”</td>
</tr>
<tr>
<td>+</td>
<td>“ab”</td>
</tr>
<tr>
<td>+</td>
<td>“ab”</td>
</tr>
<tr>
<td>+</td>
<td>“rb”</td>
</tr>
<tr>
<td>+</td>
<td>“r+b”</td>
</tr>
<tr>
<td>+</td>
<td>“w+b”</td>
</tr>
<tr>
<td>+</td>
<td>“a+b”</td>
</tr>
<tr>
<td>+</td>
<td>“a+b”</td>
</tr>
</tbody>
</table>

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 std::fclose(file)). If any of the calls made by the function, including std::fclose, fails, close fails by returning a null pointer. If one of these calls throws an exception, the exception is caught and rethrown after closing the file.

Returns: this on success, a null pointer otherwise.

Postcondition: is_open() == false.

27.9.1.5 Overridden virtual functions

streamsize showmanyc();

Effects: Behaves the same as basic_streambuf::showmanyc() (27.6.2.4).

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 by doing

339) The function signature fclose(FILE*) is declared in <cstdio> (27.9.2).
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.
— 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 coverting 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 130.

<table>
<thead>
<tr>
<th>way Value</th>
<th>stdio Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic_ios::beg</td>
<td>SEEK_SET</td>
</tr>
<tr>
<td>basic_ios::cur</td>
<td>SEEK_CUR</td>
</tr>
<tr>
<td>basic_ios::end</td>
<td>SEEK_END</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.

\textit{Returns:} sp on success. Otherwise returns pos_type(off_type(-1)).

\textbf{int sync();}

\textit{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.

\textbf{void imbue(const locale& loc);}

\textit{Requires:} If the file is not positioned at its beginning and the encoding of the current locale as determined by a\_codecvt.encoding() is state-dependent (\ref{22.4.1.4.2}) then that facet is the same as the corresponding facet of loc.

\textit{Effects:} Causes characters inserted or extracted after this call to be converted according to loc until another call of imbue.

\textit{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.

\section*{27.9.1.6 Class template basic_ifstream}  

\begin{verbatim}
namespace std {
    template <class charT, class traits = char_traits<charT> >
    class basic_ifstream : public basic_istream<charT,traits> {
public:
    typedef charT            char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits           traits_type;

    // 27.9.1.7 Constructors:
    basic_ifstream();
    explicit basic_ifstream(const char* s,  
        ios_base::openmode mode = ios_base::in);
    explicit basic_ifstream(const string& s,  
        ios_base::openmode mode = ios_base::in);
    basic_ifstream(basic_ifstream&& rhs);

    // 27.9.1.8 Assign/swap:
    basic_ifstream& operator=(basic_ifstream&& rhs);
    void swap(basic_ifstream& rhs);

    // 27.9.1.9 Members:

\end{verbatim}
The class `basic_ifstream<charT, traits>` supports reading from named files. It uses a `basic_filebuf<charT, traits>` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

— sb, the `filebuf` object.

### 27.9.1.7 basic_ifstream constructors [ifstream.cons]

```cpp
basic_ifstream();
1  Effects: Constructs an object of class `basic_ifstream<charT, traits>`, initializing the base class with `basic_istream(&sb)` and initializing sb with `basic_filebuf<charT, traits>()` (27.7.1.1.1, 27.9.1.2).

explicit basic_ifstream(const char* s,
ios_base::openmode mode = ios_base::in);
2  Effects: Constructs an object of class `basic_ifstream`, initializing the base class with `basic_istream(&sb)` and initializing sb with `basic_filebuf<charT, traits>()` (27.7.1.1.1, 27.9.1.2), then calls `rdbuf()->open(s, mode | ios_base::in)` (27.7.1.1.1, 27.9.1.2). If that function returns a null pointer, calls `setstate(failbit)`.

explicit basic_ifstream(const string& s,
ios_base::openmode mode = ios_base::in);
3  Effects: the same as `basic_ifstream(s.c_str(), mode)`.

basic_ifstream(basic_ifstream&& rhs);
4  Effects: Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_filebuf`. Next `basic_istream<charT, traits>::set_rdbuf(&sb)` is called to install the contained `basic_filebuf`.
```

### 27.9.1.8 Assign and swap [ifstream.assign]

```cpp
basic_ifstream& operator=(basic_ifstream&& rhs);
1  Effects: Move assigns the base and members of `*this` from the base and corresponding members of rhs.
2  Returns: `*this`.
```
void swap(basic_ifstream& rhs);

Effects: Exchanges the state of *this and rhs by calling basic_istream<charT,traits>::swap(rhs)
and sb.swap(rhs.sb).

template <class charT, class traits>
void swap(basic_ifstream<charT, traits>& x,
basic_ifstream<charT, traits>& y);

Effects: x.swap(y).

27.9.1.9 Member functions

basic_filebuf<charT,traits>* rdbuf() const;

Returns: const_cast<basic_filebuf<charT,traits>*>(&sb).

bool is_open() const;

Returns: rdbuf()->is_open().

void open(const char* s, ios_base::openmode mode = ios_base::in);

Effects: Calls rdbuf()->open(s, mode | ios_base::in). If that function does not return a null
pointer calls clear(), otherwise calls setstate(failbit) (which may throw ios_base::failure (27.5.4.3)).

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 (27.5.4.3)).

27.9.1.10 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;

// 27.9.1.11 Constructors:

basic_ofstream();

explicit basic_ofstream(const char* s,
    ios_base::openmode mode = ios_base::out);

explicit basic_ofstream(const string& s,
    ios_base::openmode mode = ios_base::out);

basic_ofstream(basic_ofstream&& rhs);

// 27.9.1.12 Assign/swap:

§ 27.9.1.10
basic_ofstream& operator=(basic_ofstream&& rhs);
void swap(basic_ofstream& rhs);

// 27.9.1.13 Members:
basic_filebuf<charT,traits>* rdbuf() const;

bool is_open() const;
void open(const char* s, ios_base::openmode mode = ios_base::out);
void open(const string& s, ios_base::openmode mode = ios_base::out);
void close();
private:
    basic_filebuf<charT,traits> sb; // exposition only
};

template <class charT, class traits>
void swap(basic_ofstream<charT, traits>& x,
          basic_ofstream<charT, traits>& y);

1 The class basic_ofstream<charT, traits> supports writing to named files. It uses a basic_filebuf<charT,
traits> object to control the associated sequence. For the sake of exposition, the maintained data is pre-
sented here as:
    — sb, the filebuf object.

27.9.1.11 basic_ofstream constructors

basic_ofstream();

Effects: Constructs an object of class basic_ofstream<charT,traits>, initializing the base class with
basic_ostream(&sb) and initializing sb with basic_filebuf<charT,traits>() (27.7.2.2, 27.9.1.2).

explicit basic_ofstream(const char* s,
    ios_base::openmode mode = ios_base::out);

Effects: Constructs an object of class basic_ofstream<charT,traits>, initializing the base class with
basic_ostream(&sb) and initializing sb with basic_filebuf<charT,traits>() (27.7.2.2, 27.9.1.2),
then calls rdbuf()->open(s, mode|ios_base::out). If that function returns a null pointer, calls
setstate(failbit).

explicit basic_ofstream(const string& s,
    ios_base::openmode mode = ios_base::out);

Effects: the same as basic_ofstream(s.c_str(), mode);

basic_ofstream(basic_ofstream&& rhs);

Effects: Move constructs from the rvalue rhs. This is accomplished by move constructing the base
class, and the contained basic_filebuf. Next basic_ostream<charT,traits>::set_rdbuf(&sb) is
called to install the contained basic_filebuf.

27.9.1.12 Assign and swap

basic_ofstream& operator=(basic_ofstream&& rhs);

§ 27.9.1.12 1055
Effects: Move assigns the base and members of \*this from the base and corresponding members of rhs.

Returns: \*this.

```cpp
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).

```cpp
template <class charT, class traits>
void swap(basic_ofstream<charT, traits>& x,
          basic_ofstream<charT, traits>& y);
```

Effects: x.swap(y).

### 27.9.1.13 Member functions

[ofstream.members]

```cpp
basic_filebuf<charT,traits>* rdbuf() const;
```

Returns: const_cast<basic_filebuf<charT,traits>&>(&sb).

```cpp
bool is_open() const;
```

Returns: rdbuf()->is_open().

```cpp
void open(const char* s, ios_base::openmode mode = ios_base::out);
```

Effects: Calls rdbuf()->open(s, mode | ios_base::out). If that function does not return a null pointer calls clear(), otherwise calls setstate(failbit) (which may throw ios_base::failure (27.5.4.3)).

```cpp
void close();
```

Effects: Calls rdbuf()->close() and, if that function fails (returns a null pointer), calls setstate(failbit) (which may throw ios_base::failure (27.5.4.3)).

```cpp
void open(const string& s, ios_base::openmode mode = ios_base::out);
```

Effects: calls open(s.c_str(), mode);

### 27.9.1.14 Class template basic_fstream

[fstream]

```cpp
namespace std {
    template <class charT, class traits=char_traits<charT> >
    class basic_fstream :
        public basic_iostream<charT,traits> {

    public:
        typedef charT char_type;
        typedef typename traits::int_type int_type;
        typedef typename traits::pos_type pos_type;
        typedef typename traits::off_type off_type;
        typedef traits traits_type;

        // constructors/destructor
        basic_fstream();
    }
}
```
explicit basic_fstream(const char* s,
    ios_base::openmode mode = ios_base::in|ios_base::out);
explicit basic_fstream(const string& s,
    ios_base::openmode mode = ios_base::in|ios_base::out);
basic_fstream(basic_fstream&& rhs);

// 27.9.1.16 Assignswap:
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>
void swap(basic_fstream<charT, traits>& x,
    basic_fstream<charT, traits>& y);

1 The class template basic_fstream<charT,traits> supports reading and writing from named files. It uses
a basic_filebuf<charT,traits> object to control the associated sequences. For the sake of exposition,
the maintained data is presented here as:
— sb, the basic_filebuf object.

27.9.1.15 basic_fstream constructors [fstream.cons]
basic_fstream();

1 Effects: Constructs an object of class basic_fstream<charT,traits>, initializing the base class with
basic_iosstream(&sb) and initializing sb with basic_filebuf<charT,traits>().
explicit basic_fstream(const char* s,
    ios_base::openmode mode = ios_base::in|ios_base::out);

2 Effects: Constructs an object of class basic_fstream<charT, traits>, initializing the base class
with basic_iosstream(&sb) and initializing sb with basic_filebuf<charT, traits>(). Then calls
rdbuf()->open(s, mode). If that function returns a null pointer, calls setstate(failbit).

explicit basic_fstream(const string& s,
    ios_base::openmode mode = ios_base::in|ios_base::out);

3 Effects: the same as basic_fstream(s.c_str(), mode);
basic_fstream(basic_fstream&& rhs);

§ 27.9.1.15
Effects: Move constructs from the rvalue rhs. This is accomplished by move constructing the base class, and the contained basic_filebuf. Next basic_istream<charT,traits>::set_rdbuf(&sb) is called to install the contained basic_filebuf.

27.9.1.16 Assign and swap

basic_fstream& operator=(basic_fstream&& rhs);

Effects: Move assigns the base and members of *this from the base and corresponding members of rhs.
Returns: *this.

void swap(basic_fstream& rhs);

Effects: Exchanges the state of *this and rhs by calling basic_iostream<charT,traits>::swap(rhs) and sb.swap(rhs.sb).

template <class charT, class traits>
void swap(basic_fstream<charT, traits>& x, basic_fstream<charT, traits>& y);

Effects: x.swap(y).

27.9.1.17 Member functions

basic_filebuf<charT,traits>* rdbuf() const;
Returns: const_cast<basic_filebuf<charT,traits>*>(&sb).

bool is_open() const;
Returns: rdbuf()->is_open().

void open(const char* s, ios_base::openmode mode = ios_base::in|ios_base::out);
Effects: Calls rdbuf()->open(s,mode). If that function does not return a null pointer calls clear(), otherwise calls setstate(failbit), (which may throw ios_base::failure) (27.5.4.3).

void open(const string& s, ios_base::openmode mode = ios_base::in|ios_base::out);

Effects: calls open(s.c_str(), mode);

void close();
Effects: Calls rdbuf()->close() and, if that function returns returns a null pointer, calls setstate(failbit) (27.5.4.3) (which may throw ios_base::failure).

27.9.2 C Library files

Table 131 describes header <cstdio>.
Calls to the function tmpnam with an argument of NULL may introduce a data race (17.6.4.8) with other calls to tmpnam with an argument of NULL.
Table 131 — 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>BUFSIZE</td>
<td>FOPEN_MAX</td>
</tr>
<tr>
<td>EOF</td>
<td>L_tmpnam</td>
</tr>
<tr>
<td>FILENAME_MAX</td>
<td>NULL <code>&lt;cstdio&gt;</code></td>
</tr>
<tr>
<td>Types:</td>
<td>FILE fpos_t size_t <code>&lt;cstdio&gt;</code></td>
</tr>
<tr>
<td>Functions:</td>
<td></td>
</tr>
<tr>
<td>clearerr</td>
<td>fopen fsetpos putc setbuf vprintf</td>
</tr>
<tr>
<td>fclose</td>
<td>fprintf ftell putchar setvbuf vscanf</td>
</tr>
<tr>
<td>feof</td>
<td>fputc fwrite puts snprintf vsprintf</td>
</tr>
<tr>
<td>ferror</td>
<td>fputs getc rename sprintf vsprintf</td>
</tr>
<tr>
<td>fflush</td>
<td>freopen getchar remove tmpfile vsscanf</td>
</tr>
<tr>
<td>fgetpos</td>
<td>fscanf perror scanf ungetc</td>
</tr>
<tr>
<td>fget</td>
<td>fseek printf sscanf vsprintf</td>
</tr>
</tbody>
</table>

See also: ISO C 7.9, Amendment 1 4.6.2.

Table 132 describes header `<cinttypes>`. [Note: The macros defined by `<cinttypes>` are provided unconditionally. In particular, the symbol `__STDC_FORMAT_MACROS`, mentioned in footnote 182 of the C standard, plays no role in C++. — end note]

Table 132 — 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 wcstoimax</td>
</tr>
<tr>
<td>div</td>
<td>imaxdiv strtoimax wcstoimax</td>
</tr>
</tbody>
</table>
28 Regular expressions library [re]

28.1 General [re.general]

1 This Clause describes components that C++ programs may use to perform operations involving regular expression matching and searching.

2 The following subclauses describe a basic regular expression class template and its traits that can handle char-like template arguments, two specializations of this template class that handle sequences of char and wchar_t, a class template that holds the result of a regular expression match, a series of algorithms that allow a character sequence to be operated upon by a regular expression, and two iterator types for enumerating regular expression matches, as described in Table 133.

Table 133 — Regular expressions library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.2</td>
<td>Definitions</td>
</tr>
<tr>
<td>28.3</td>
<td>Requirements</td>
</tr>
<tr>
<td>28.5</td>
<td>Constants</td>
</tr>
<tr>
<td>28.6</td>
<td>Exception type</td>
</tr>
<tr>
<td>28.7</td>
<td>Traits</td>
</tr>
<tr>
<td>28.8</td>
<td>Regular expression template &lt;regex&gt;</td>
</tr>
<tr>
<td>28.9</td>
<td>Submatches</td>
</tr>
<tr>
<td>28.10</td>
<td>Match results</td>
</tr>
<tr>
<td>28.11</td>
<td>Algorithms</td>
</tr>
<tr>
<td>28.12</td>
<td>Iterators</td>
</tr>
<tr>
<td>28.13</td>
<td>Grammar</td>
</tr>
</tbody>
</table>

28.2 Definitions [re.def]

1 The following definitions shall apply to this Clause:

28.2.1 collating element [defns.regex.collating.element]

a sequence of one or more characters within the current locale that collate as if they were a single character.

28.2.2 finite state machine [defns.regex.finite.state.machine]

an unspecified data structure that is used to represent a regular expression, and which permits efficient matches against the regular expression to be obtained.

28.2.3 format specifier [defns.regex.formatspecifier]

a sequence of one or more characters that is to be replaced with some part of a regular expression match.
28.2.4 matched
a sequence of zero or more characters is matched by a regular expression when the characters in the sequence correspond to a sequence of characters defined by the pattern.

28.2.5 primary equivalence class
a set of one or more characters which share the same primary sort key: that is the sort key weighting that depends only upon character shape, and not accentation, case, or locale specific tailorings.

28.2.6 regular expression
a pattern that selects specific strings from a set of character strings.

28.2.7 sub-expression
a subset of a regular expression that has been marked by parenthesis.

28.3 Requirements

1 This subclause defines requirements on classes representing regular expression traits. [Note: The class template 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 134 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 134 — Regular expression traits class requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::char_type</td>
<td>charT</td>
<td>The character container type used in the implementation of class template basic_regex.</td>
</tr>
<tr>
<td>X::string_type</td>
<td>std::basic_string&lt;CharT&gt;</td>
<td>A copy constructible type of X::char_type</td>
</tr>
<tr>
<td>X::locale_type</td>
<td>A copy constructible type of X::locale_type</td>
<td>A type that represents the locale used by the traits class.</td>
</tr>
<tr>
<td>X::char_class_type</td>
<td>A bitmask type (17.5.2.1.3).</td>
<td>A bitmask type representing a particular character classification.</td>
</tr>
<tr>
<td>X::length(p)</td>
<td>std::size_t</td>
<td>Yields the smallest i such that p[i] == 0. Complexity is linear in i.</td>
</tr>
</tbody>
</table>

§ 28.3
Table 134 — Regular expression traits class requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>v.translate(c)</td>
<td>X::char_type</td>
<td>Returns a character such that for any character d that is to be considered equivalent to c then v.translate(c) == v.translate(d).</td>
</tr>
<tr>
<td>v.translate_nocase(c)</td>
<td>X::char_type</td>
<td>For all characters C that are to be considered equivalent to c when comparisons are to be performed without regard to case, then v.translate_nocase(c) == v.translate_nocase(C).</td>
</tr>
<tr>
<td>v.transform(F1, F2)</td>
<td>X::string_type</td>
<td>Returns a sort key for the character sequence designated by the iterator range [F1,F2) such that if the character sequence [G1,G2) sorts before the character sequence [H1,H2) then v.transform(G1, G2) &lt; v.transform(H1, H2).</td>
</tr>
<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>
</tbody>
</table>
Table 134 — Regular expression traits class requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>v.value(c, I)</td>
<td>int</td>
<td>Returns the value represented by the digit c in base I if the character c is a valid digit in base I; otherwise returns -1. [Note: the value of I will only be 8, 10, or 16. — end note]</td>
</tr>
<tr>
<td>u.imbue(loc)</td>
<td>X::locale_type</td>
<td>Imbues u with the locale loc and returns the previous locale used by u if any.</td>
</tr>
<tr>
<td>v.getloc()</td>
<td>X::locale_type</td>
<td>Returns the current locale used by v, if any.</td>
</tr>
</tbody>
</table>

5 [Note: Class template regex_traits satisfies the requirements for a regular expression traits class when it is specialized for char or wchar_t. This Class template is described in the header <regex>, and is described in Clause 28.7. — end note]

28.4 Header <regex> synopsis

```cpp
namespace std {
    #include <initializer_list>
    
    // 28.5, regex constants:
    namespace regex_constants {
        enum error_type;
    } // namespace regex_constants
    
    // 28.6, class regex_error:
    class regex_error;
    
    // 28.7, class template regex_traits:
    template <class charT> struct regex_traits;
    
    // 28.8, class template basic_regex:
    template <class charT, class traits = regex_traits<charT>> class basic_regex;
    typedef basic_regex<char> regex;
    typedef basic_regex<wchar_t> wregex;
    
    // 28.8.6, basic_regex swap:
    template <class charT, class traits>
    void swap(basic_regex<charT, traits>& e1, basic_regex<charT, traits>& e2);
    
    // 28.9, class template sub_match:
    template <class BidirectionalIterator>
    class sub_match;
    typedef sub_match<const char*> csub_match;
    typedef sub_match<const wchar_t*> wcsub_match;
    typedef sub_match<string::const_iterator> ssub_match;
    typedef sub_match<wstring::const_iterator> wssub_match;
    
    // 28.9.2, sub_match non-member operators:
    template <class BiIter>
```
bool operator==(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

template <class BiIter>
bool operator!=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

template <class BiIter>
bool operator<(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

template <class BiIter>
bool operator<=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

template <class BiIter>
bool operator>=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

template <class BiIter>
bool operator>(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

template <class BiIter, class ST, class SA>
bool operator==(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

template <class BiIter, class ST, class SA>
bool operator!=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

template <class BiIter, class ST, class SA>
bool operator<(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

template <class BiIter, class ST, class SA>
bool operator>(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

template <class BiIter, class ST, class SA>
bool operator>=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

template <class BiIter, class ST, class SA>
bool operator<=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

template <class BiIter, class ST, class SA>
bool operator==(const sub_match<BiIter>& lhs, const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template <class BiIter, class ST, class SA>
bool operator!=(const sub_match<BiIter>& lhs, const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

§ 28.4 1064
bool operator<(
    const sub_match<BiIter>& lhs,
    const basic_string<
        typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template <class BiIter, class ST, class SA>
bool operator<(
    const sub_match<BiIter>& lhs,
    const basic_string<
        typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template <class BiIter, class ST, class SA>
bool operator<(
    const sub_match<BiIter>& lhs,
    const basic_string<
        typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template <class BiIter, class ST, class SA>
bool operator<(
    const sub_match<BiIter>& lhs,
    const basic_string<
        typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template <class BiIter>
bool operator==(typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter>
bool operator==(typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter>
bool operator==(typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter>
bool operator==(typename iterator_traits<BiIter>::value_type const* lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter, class ST, class SA>
bool operator==(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const* rhs);

template <class BiIter, class ST, class SA>
bool operator==(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const* rhs);

§ 28.4 1065
bool operator<=(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const* rhs);

template <class BiIter>
bool operator==(typename iterator_traits<BiIter>::value_type const& lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter>
bool operator!=(typename iterator_traits<BiIter>::value_type const& lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter>
bool operator<(typename iterator_traits<BiIter>::value_type const& lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter>
bool operator>(typename iterator_traits<BiIter>::value_type const& lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter>
bool operator>=(typename iterator_traits<BiIter>::value_type const& lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter>
bool operator<=(typename iterator_traits<BiIter>::value_type const& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

template <class BiIter>
bool operator==(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

template <class BiIter>
bool operator!=(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

template <class BiIter>
bool operator<(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

template <class BiIter>
bool operator>(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

template <class BiIter>
bool operator>=(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

template <class BiIter>
bool operator<=(const sub_match<BiIter>& lhs,
    typename iterator_traits<BiIter>::value_type const& rhs);

template <class BiIter>
bool operator==(const sub_match<BiIter>& lhs,
    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);

// 28.10, class template match_results:
template <class BidirectionalIterator,       
class Allocator = allocator<sub_match<BidirectionalIterator> > > 
class match_results;

typedef match_results<const char*> cmatch;
typedef match_results<const wchar_t*> wcmatch;
typedef match_results<string::const_iterator> smatch;
typedef match_results<wstring::const_iterator> wsmatch;
// match_results comparisons
template <class BidirectionalIterator, class Allocator>
bool operator==(const match_results<BidirectionalIterator, Allocator>& m1,
                const match_results<BidirectionalIterator, Allocator>& m2);

template <class BidirectionalIterator, class Allocator>
bool operator!=(const match_results<BidirectionalIterator, Allocator>& m1,
                const match_results<BidirectionalIterator, Allocator>& m2);

// 28.10.6, match_results swap:
template <class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1,
          match_results<BidirectionalIterator, Allocator>& m2);

// 28.11.2, function template regex_match:
template <class BidirectionalIterator, class Allocator,
class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 match_results<BidirectionalIterator, Allocator>& m,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags =
                 regex_constants::match_default);

template <class BidirectionalIterator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags =
                 regex_constants::match_default);

template <class charT, class Allocator, class traits>
bool regex_match(const charT* str, match_results<const charT*, Allocator>& m,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags =
                 regex_constants::match_default);

template <class ST, class SA, class Allocator, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
                 match_results<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,
const basic_regex<charT, traits>& e,
regex_constants::match_flag_type flags =
regex_constants::match_default);

template <class ST, class SA, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
const basic_regex<charT, traits>& e,
regex_constants::match_flag_type flags =
regex_constants::match_default);

template <class ST, class SA, class Allocator, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
match_results<
typename basic_string<charT, ST, SA>::const_iterator,
Allocator>& m,
const basic_regex<charT, traits>& e,
regex_constants::match_flag_type flags =
regex_constants::match_default>;

// 28.11.4, function template regex_replace:

template <class OutputIterator, class BidirectionalIterator,
class traits, class charT, class ST, class SA>
OutputIterator
regex_replace(OutputIterator out,
BidirectionalIterator first, BidirectionalIterator last,
const basic_regex<charT, traits>& e,
const basic_string<charT, ST, SA>& fmt,
regex_constants::match_flag_type flags =
regex_constants::match_default);

template <class OutputIterator, class BidirectionalIterator,
class traits, class charT>
OutputIterator
regex_replace(OutputIterator out,
BidirectionalIterator first, BidirectionalIterator last,
const basic_regex<charT, traits>& e,
const charT* fmt,
regex_constants::match_flag_type flags =
regex_constants::match_default);

template <class traits, class charT, class ST, class SA,
class FST, class FSA>
basic_string<charT, ST, SA>
```cpp
regex_replace(const basic_string<charT, ST, SA>& s,
              const basic_regex<charT, traits>& e,
              const basic_string<charT, FST, FSA>& fmt,
              regex_constants::match_flag_type flags =
              regex_constants::match_default);
```

```cpp
template <class traits, class charT, class ST, class SA>
basic_string<charT, ST, SA>
regex_replace(const basic_string<charT, ST, SA>& s,
              const basic_regex<charT, traits>& e,
              const charT* fmt,
              regex_constants::match_flag_type flags =
              regex_constants::match_default);
```

```cpp
template <class traits, class charT>
basic_string<charT>
regex_replace(const charT* s,
              const basic_regex<charT, traits>& e,
              const basic_string<charT, ST, SA>& fmt,
              regex_constants::match_flag_type flags =
              regex_constants::match_default);
```

```cpp
template <class traits, class charT>
basic_string<charT>
regex_replace(const charT* s,
              const basic_regex<charT, traits>& e,
              const charT* fmt,
              regex_constants::match_flag_type flags =
              regex_constants::match_default);
```

```cpp
// 28.5 1069
1 The namespace std::regex_constants holds symbolic constants used by the regular expression library.
```

**28.5 Namespace std::regex_constants**

The namespace std::regex_constants holds symbolic constants used by the regular expression library.
This namespace provides three types, \texttt{syntax\_option\_type}, \texttt{match\_flag\_type}, and \texttt{error\_type}, along with several constants of these types.

### 28.5.1 Bitmask Type \texttt{syntax\_option\_type}

```cpp
class std { 
namespace regex\_constants { 
    enum syntax\_option\_type { 
        icase = \texttt{unspecified},
        nosubs = \texttt{unspecified},
        optimize = \texttt{unspecified},
        collate = \texttt{unspecified},
        ECMAScript = \texttt{unspecified},
        basic = \texttt{unspecified},
        extended = \texttt{unspecified},
        awk = \texttt{unspecified},
        grep = \texttt{unspecified},
        egrep = \texttt{unspecified},
    };
    constexpr syntax\_option\_type\ operator\~(syntax\_option\_type f);
    constexpr syntax\_option\_type\ operator&(syntax\_option\_type lhs, syntax\_option\_type rhs);
    constexpr syntax\_option\_type\ operator|(syntax\_option\_type lhs, syntax\_option\_type rhs);
}
```

1. The type \texttt{syntax\_option\_type} is an implementation-defined bitmask type (17.5.2.1.3). Setting its elements has the effects listed in table 135. A valid value of type \texttt{syntax\_option\_type} shall have exactly one of the elements ECMAScript, basic, extended, awk, grep, egrep, set.

```cpp
class std { 
namespace regex\_constants { 
    enum syntax\_option\_type { 
        icase = \texttt{unspecified},
        nosubs = \texttt{unspecified},
        optimize = \texttt{unspecified},
        collate = \texttt{unspecified},
        ECMAScript = \texttt{unspecified},
        basic = \texttt{unspecified},
        extended = \texttt{unspecified},
        awk = \texttt{unspecified},
        grep = \texttt{unspecified},
        egrep = \texttt{unspecified},
    };
    constexpr syntax\_option\_type\ operator\~(syntax\_option\_type f);
    constexpr syntax\_option\_type\ operator&(syntax\_option\_type lhs, syntax\_option\_type rhs);
    constexpr syntax\_option\_type\ operator|(syntax\_option\_type lhs, syntax\_option\_type rhs);
}
```

2. \texttt{constexpr syntax\_option\_type\ operator\~(syntax\_option\_type f);}  
   
   \texttt{Returns: syntax\_option\_type( f).}

3. \texttt{constexpr syntax\_option\_type\ operator&(syntax\_option\_type lhs, syntax\_option\_type rhs);}  
   
   \texttt{Returns: syntax\_option\_type(int(lhs) \& int(rhs)).}

4. \texttt{constexpr syntax\_option\_type\ operator|(syntax\_option\_type lhs, syntax\_option\_type rhs);}  
   
   \texttt{Returns: syntax\_option\_type(int(lhs) | int(rhs)).}

### 28.5.2 Bitmask Type \texttt{regex\_constants::match\_flag\_type}

```cpp
class std { 
namespace regex\_constants { 
    enum match\_flag\_type { 
        match\_default = 0,
        match\_not\_bol = \texttt{unspecified},
        match\_not\_eol = \texttt{unspecified},
        match\_not\_bow = \texttt{unspecified},
        match\_not\_eow = \texttt{unspecified},
        match\_any = \texttt{unspecified},
        match\_not\_null = \texttt{unspecified},
        match\_continuous = \texttt{unspecified},
        match\_prev\_avail = \texttt{unspecified},
    };
```

§ 28.5.2
### Table 135 — syntax_option_type effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Effect(s) if set</th>
</tr>
</thead>
<tbody>
<tr>
<td>icase</td>
<td>Specifies that matching of regular expressions against a character container</td>
</tr>
<tr>
<td></td>
<td>sequence shall be performed without regard to case.</td>
</tr>
<tr>
<td>nosubs</td>
<td>Specifies that when a regular expression is matched against a character container</td>
</tr>
<tr>
<td></td>
<td>sequence, no sub-expression matches shall be stored in the supplied match_results</td>
</tr>
<tr>
<td>optimize</td>
<td>Specifies that the regular expression engine should pay more attention to the</td>
</tr>
<tr>
<td></td>
<td>speed with which regular expressions are matched, and less to the speed with</td>
</tr>
<tr>
<td></td>
<td>which regular expression objects are constructed. Otherwise it has no detectable</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>that used by basic regular expressions in POSIX, Base Definitions and Headers,</td>
</tr>
<tr>
<td></td>
<td>Section 9, Regular Expressions.</td>
</tr>
<tr>
<td>extended</td>
<td>Specifies that the grammar recognized by the regular expression engine shall be</td>
</tr>
<tr>
<td></td>
<td>that used by extended regular expressions in POSIX, Base Definitions and Headers,</td>
</tr>
<tr>
<td></td>
<td>Section 9, Regular Expressions.</td>
</tr>
<tr>
<td>awk</td>
<td>Specifies that the grammar recognized by the regular expression engine shall be</td>
</tr>
<tr>
<td></td>
<td>that used by the utility awk in POSIX.</td>
</tr>
<tr>
<td>grep</td>
<td>Specifies that the grammar recognized by the regular expression engine shall be</td>
</tr>
<tr>
<td></td>
<td>that used by the utility grep in POSIX.</td>
</tr>
<tr>
<td>egrep</td>
<td>Specifies that the grammar recognized by the regular expression engine shall be</td>
</tr>
<tr>
<td></td>
<td>that used by the utility grep when given the -E option in POSIX.</td>
</tr>
</tbody>
</table>

```cpp
format_default = 0,
format_sed = unspecified,
format_no_copy = unspecified,
format_first_only = unspecified,
};

constexpr match_flag_type operator~(match_flag_type f);
constexpr match_flag_type operator&(match_flag_type lhs, match_flag_type rhs);
constexpr match_flag_type operator|(match_flag_type lhs, match_flag_type rhs);
};
```

1 The type `regex_constants::match_flag_type` is an implementation-defined bitmask type (17.5.2.1.3). 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 136 for any bitmask elements set.
Table 136 — `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>$</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 <code>match_not_bol</code> and <code>match_not_bow</code> 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 <code>replace</code> function in ECMA-262, part 15.4.11 <code>String.prototype.replace</code>. 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 <code>sed</code> 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>

```cpp
constexpr match_flag_type operator~(match_flag_type f);
// Returns: match_flag_type( f).
constexpr match_flag_type operator&(match_flag_type lhs, match_flag_type rhs);
// Returns: match_flag_type(int(lhs) & int(rhs)).
constexpr match_flag_type operator|(match_flag_type lhs, match_flag_type rhs);
// Returns: match_flag_type(int(lhs) | int(rhs)).
```

### 28.5.3 Implementation-defined `error_type` [re.err]

```cpp
namespace std {
    namespace regex_constants {

```
enum error_type {
    error_collate = unspecified,
    error_ctype = unspecified,
    error_escape = unspecified,
    error_backref = unspecified,
    error_brack = unspecified,
    error_paren = unspecified,
    error_brace = unspecified,
    error_badbrace = unspecified,
    error_range = unspecified,
    error_space = unspecified,
    error_badrepeat = unspecified,
    error_complexity = unspecified,
    error_stack = unspecified,
};

constexpr error_type operator~(error_type f);
constexpr error_type operator&(error_type lhs, error_type rhs);
constexpr error_type operator|(error_type lhs, error_type rhs);
}

1 The type error_type is an implementation-defined enumeration type (17.5.2.1.2). Values of type error_type represent the error conditions described in table 137:

Table 137 — 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>

constexpr error_type operator~(error_type f);

2 Returns: error_type( f).

constexpr error_type operator&(error_type lhs, error_type rhs);
Returns: `error_type(int(lhs) & int(rhs))`.

```cpp
constexpr error_type operator|(error_type lhs, error_type rhs);
```

Returns: `error_type(int(lhs) | int(rhs))`.

### 28.6 Class `regex_error`

```cpp
class regex_error : public std::runtime_error {
  public:
    explicit regex_error(regex_constants::error_type ecode);
    regex_constants::error_type code() const;
};
```

The class `regex_error` defines the type of objects thrown as exceptions to report errors from the regular expression library.

```cpp
regex_error(regex_constants::error_type ecode);
```

Effects: Constructs an object of class `regex_error`.

Postcondition: `ecode == code()`.

```cpp
regex_constants::error_type code() const;
```

Returns: The error code that was passed to the constructor.

### 28.7 Class template `regex_traits`

```cpp
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);
  };
```

§ 28.7 1074
locale_type getloc() const;
};

The specializations `regex_traits<char>` and `regex_traits<wchar_t>` shall be valid and shall satisfy the requirements for a regular expression traits class (28.3).

typedef bitmask_type char_class_type;

The type `char_class_type` is used to represent a character classification and is capable of holding an implementation specific set returned by `lookup_classname`.

static std::size_t length(const char_type* p);

`Returns:` `char_traits<charT>::length(p);`

charT translate(charT c) const;

`Returns:` `(c).`

charT translate_nocase(charT c) const;

`Returns:` `use_facet<ctype<charT>>(getloc()).tolower(c).`

template <class ForwardIterator>
string_type transform(ForwardIterator first, ForwardIterator last) const;

`Effects:`

string_type str(first, last);
return use_facet<collate<charT>>(getloc()).transform(&*str.begin(), &*str.begin() + str.length());

template <class ForwardIterator>
string_type transform_primary(ForwardIterator first, ForwardIterator last) const;

`Effects:` if typeid(use_facet<collate<charT>> == typeid(collate_byname<charT>) and the form of the sort key returned by `collate_byname<charT>::transform(first, last)` is known and can be converted into a primary sort key then returns that key, otherwise returns an empty string.

template <class ForwardIterator>
string_type lookup_collatename(ForwardIterator first, ForwardIterator last) const;

`Returns:` a sequence of one or more characters that represents the collating element consisting of the character sequence designated by the iterator range `[first,last)`. Returns an empty string if the character sequence is not a valid collating element.

template <class ForwardIterator>
char_class_type lookup_classname(
    ForwardIterator first, ForwardIterator last, bool icase = false) const;

`Returns:` an unspecified value that represents the character classification named by the character sequence designated by the iterator range `[first,last)`. If the parameter `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.\[340\] The value returned shall

\[340\] For example, if the parameter `icase` is true then `[:lower:]` is the same as `[:alpha:]`.\]
be independent of the case of the characters in the character sequence. If the name is not recognized then returns a value that compares equal to 0.

Remarks: For \texttt{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 \texttt{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.

\begin{verbatim}
bool isctype(charT c, char_class_type f) const;
\end{verbatim}

Effects: Determines if the character \texttt{c} is a member of the character classification represented by \texttt{f}.

Returns: Converts \texttt{f} into a value \texttt{m} of type \texttt{std::ctype_base::mask} in an unspecified manner, and returns \texttt{true} if \texttt{use_facet<ctype<charT> >>(getloc()).is(c, m)} is \texttt{true}. Otherwise returns \texttt{true} if \texttt{f} bitwise or'ed with the result of calling \texttt{lookup_classname} with an iterator pair that designates the character sequence "w" is not equal to 0 and \texttt{c == '_'}, or if \texttt{f} bitwise or'ed with the result of calling \texttt{lookup_classname} with an iterator pair that designates the character sequence "blank" is not equal to 0 and \texttt{c} is one of an implementation-defined subset of the characters for which \texttt{isspace(c, getloc())} returns \texttt{true}, otherwise returns \texttt{false}.

\begin{verbatim}
int value(charT ch, int radix) const;
\end{verbatim}

Requires: The value of \texttt{radix} shall be 8, 10, or 16.

Returns: the value represented by the digit \texttt{ch} in base \texttt{radix} if the character \texttt{ch} is a valid digit in base \texttt{radix}; otherwise returns \texttt{-1}.

\begin{verbatim}
locale_type imbue(locale_type loc);
\end{verbatim}

Effects: Imbues \texttt{this} with a copy of the locale \texttt{loc}. \texttt{Note:} calling \texttt{imbue} with a different locale than the one currently in use invalidates all cached data held by \texttt{*this}. \texttt{— end note}\]

Returns: if no locale has been previously imbued then a copy of the global locale in effect at the time of construction of \texttt{*this}, otherwise a copy of the last argument passed to \texttt{imbue}.

Postcondition: \texttt{getloc() == loc}.

\begin{verbatim}
locale_type getloc()const;
\end{verbatim}

Returns: if no locale has been imbued then a copy of the global locale in effect at the time of construction of \texttt{*this}, otherwise a copy of the last argument passed to \texttt{imbue}.

\section{Class template basic_regex}

1 For a char-like type \texttt{charT}, specializations of class template \texttt{basic_regex} represent regular expressions constructed from character sequences of \texttt{charT} characters. In the rest of 28.8, \texttt{charT} denotes a given char-like type. Storage for a regular expression is allocated and freed as necessary by the member functions of class \texttt{basic_regex}.

2 Objects of type specialization of \texttt{basic_regex} are responsible for converting the sequence of \texttt{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. \texttt{Note:} implementations will typically declare some function templates as friends of \texttt{basic_regex} to achieve this \texttt{— end note}\]

3 The functions described in this Clause report errors by throwing exceptions of type \texttt{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 constexpr regex_constants::syntax_option_type icase = regex_constants::icase;
        static constexpr regex_constants::syntax_option_type nosubs = regex_constants::nosubs;
        static constexpr regex_constants::syntax_option_type optimize = regex_constants::optimize;
        static constexpr regex_constants::syntax_option_type collate = regex_constants::collate;
        static constexpr regex_constants::syntax_option_type ECMAScript = regex_constants::ECMAScript;
        static constexpr regex_constants::syntax_option_type basic = regex_constants::basic;
        static constexpr regex_constants::syntax_option_type extended = regex_constants::extended;
        static constexpr regex_constants::syntax_option_type awk = regex_constants::awk;
        static constexpr regex_constants::syntax_option_type grep = regex_constants::grep;
        static constexpr regex_constants::syntax_option_type egrep = regex_constants::egrep;

        // 28.8.2, construct/copy/destroy:
        basic_regex();
        explicit basic_regex(const charT* p,
                              flag_type f = regex_constants::ECMAScript);
        basic_regex(const charT* p, size_t len, flag_type f);
        basic_regex(const basic_regex&);
        basic_regex(basic_regex&&);
        template <class ST, class SA>
        explicit basic_regex(const basic_string<charT, ST, SA>& p,
                              flag_type f = regex_constants::ECMAScript);
        template <class ForwardIterator>
        basic_regex(ForwardIterator first, ForwardIterator last,
                    flag_type f = regex_constants::ECMAScript);
        basic_regex(initializer_list<charT>);
        basic_regex(initializer_list<charT> il);
        ~basic_regex();

        basic_regex& operator=(const basic_regex&);
        basic_regex& operator=(basic_regex&&);
        basic_regex& operator=(const charT* ptr);
        basic_regex& operator=(initializer_list<charT> il);
    };
}
basic_regex& operator=(const basic_string<charT, ST, SA>& p);

// 28.8.3, assign:
  basic_regex& assign(const basic_regex& that);
basic_regex& assign(basic_regex&& that);
basic_regex& assign(const charT* ptr,
    flag_type f = regex_constants::ECMAScript);
basic_regex& assign(const charT* p, size_t len, flag_type f);
template <class string_traits, class A>
  basic_regex& assign(const basic_string<charT, string_traits, A>& s,
    flag_type f = regex_constants::ECMAScript);
template <class InputIterator>
  basic_regex& assign(InputIterator first, InputIterator last,
    flag_type f = regex_constants::ECMAScript);
basic_regex& assign(initializer_list<charT>,
    flag_type = regex_constants::ECMAScript);

// 28.8.4, const operations:
  unsigned mark_count() const;
  flag_type flags() const;

// 28.8.5, locale:
  locale_type imbue(locale_type loc);
  locale_type getloc() const;

// 28.8.6, swap:
void swap(basic_regex&);
};

28.8.1 basic_regex constants

static constexpr regex_constants::syntax_option_type
  icase = regex_constants::icase;
static constexpr regex_constants::syntax_option_type
  nosubs = regex_constants::nosubs;
static constexpr regex_constants::syntax_option_type
  optimize = regex_constants::optimize;
static constexpr regex_constants::syntax_option_type
  collate = regex_constants::collate;
static constexpr regex_constants::syntax_option_type
  ECMAScript = regex_constants::ECMAScript;
static constexpr regex_constants::syntax_option_type
  basic = regex_constants::basic;
static constexpr regex_constants::syntax_option_type
  extended = regex_constants::extended;
static constexpr regex_constants::syntax_option_type
  awk = regex_constants::awk;
static constexpr regex_constants::syntax_option_type
  grep = regex_constants::grep;
static constexpr regex_constants::syntax_option_type
  egrep = regex_constants::egrep;

1 The static constant members are provided as synonyms for the constants declared in namespace regex_
constants.

### 28.8.2 basic_regex constructors

```cpp
basic_regex();
```  
**Effects:** Constructs an object of class `basic_regex` that does not match any character sequence.

```cpp
basic_regex(const charT* p, flag_type f = regex_constants::ECMAScript);
```

**Requires:** `p` shall not be a null pointer.

**Throws:** `regex_error` if `p` is not a valid regular expression.

**Effects:** Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the array of `charT` of length `char_traits<charT>::length(p)` whose first element is designated by `p`, and interpreted according to the flags `f`.

**Postconditions:** `flags()` returns `f`. `mark_count()` returns the number of marked sub-expressions within the expression.

```cpp
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.

```cpp
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.

```cpp
basic_regex(basic_regex&& e);
```

**Effects:** Move constructs an object of class `basic_regex` from `e`.

**Postconditions:** `flags()` and `mark_count()` return the values that `e.flags()` and `e.mark_count()`, respectively, had before construction. `e` is in a valid state with unspecified value.

**Throws:** nothing.

```cpp
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 con-
structed from the regular expression contained in the sequence of characters \([first, last)\), and in-
terpreted according to the flags specified in \(f\).

Postconditions: flags() returns \(f\). mark_count() returns the number of marked sub-expressions
within the expression.

basic_regex(initializer_list<charT> il,
            flag_type f = regex_constants::ECMAScript);

Effects: Same as basic_regex(il.begin(), il.end(), \(f\)).

28.8.3 basic_regex assign          [re.regex.assign]

basic_regex& operator=(const basic_regex& e);

Effects: returns assign(e).

basic_regex& operator=(basic_regex&& e);

Effects: returns assign(std::move(e)).

basic_regex& operator=(const charT* ptr);

Requires: ptr shall not be a null pointer.

Effects: returns assign(ptr).

basic_regex& operator=(initializer_list<charT> il);

Effects: returns assign(il.begin(), il.end()).

template <class ST, class SA>
    basic_regex& operator=(const basic_string<charT, ST, SA>& p);

Effects: returns assign(p).

basic_regex& assign(const basic_regex& that);

Effects: copies that into *this and returns *this.

Postconditions: flags() and mark_count() return that.flags() and that.mark_count(), respec-
tively.

basic_regex& assign(basic_regex&& that);

Effects: move assigns from that into *this and returns *this.

Postconditions: flags() and mark_count() return the values that that.flags() and that.mark_count()
, respectively, had before assignment. that is in a valid state with unspecified value.

Throws: nothing.
basic_regex& assign(const charT* ptr, flag_type f = regex_constants::ECMAScript);

Returns: assign(string_type(ptr), f).

basic_regex& assign(const charT* ptr, size_t len,
flag_type f = regex_constants::ECMAScript);

Returns: assign(string_type(ptr, len), f).

template <class string_traits, class A>
basic_regex& assign(const basic_string<charT, string_traits, A>& s,
flag_type f = regex_constants::ECMAScript);

Throws: regex_error if s is not a valid regular expression.

Returns: *this.

Effects: Assigns the regular expression contained in the string s, interpreted according the flags specified in f. If an exception is thrown, *this is unchanged.

Postconditions: If no exception is thrown, flags() returns f and mark_count() returns the number of marked sub-expressions within the expression.

template <class InputIterator>
basic_regex& assign(InputIterator first, InputIterator last,
flag_type f = regex_constants::ECMAScript);

Requires: The type InputIterator shall satisfy the requirements for an Input Iterator (24.2.3).

Returns: assign(string_type(first, last), f).

basic_regex& assign(initializer_list<charT> il,
flag_type f = regex_constants::ECMAScript);

Effects: Same as assign(il.begin(), il.end(), f).

Returns: *this.

28.8.4 basic_regex constant operations

unsigned mark_count() const;

Effects: Returns the number of marked sub-expressions within the regular expression.

flag_type flags() const;

Effects: Returns a copy of the regular expression syntax flags that were passed to the object’s constructor or to the last call to assign.

28.8.5 basic_regex locale

locale_type imbue(locale_type loc);

Effects: Returns the result of traits_inst.imbue(loc) where traits_inst is a (default initialized) instance of the template type argument traits stored within the object. After a call to imbue the basic_regex object does not match any character sequence.
locale_type getloc() const;

Effects: Returns the result of traits_inst.getloc() where traits_inst is a (default initialized) instance of the template parameter traits stored within the object.

28.8.6 basic_regex swap

void swap(basic_regex& e);

Effects: Swaps the contents of the two regular expressions.

Postcondition: *this contains the regular expression that was in e, e contains the regular expression that was in *this.

Complexity: constant time.

28.8.7 basic_regex non-member functions

28.8.7.1 basic_regex non-member swap

template <class charT, class traits>
void swap(basic_regex<charT, traits>& lhs, basic_regex<charT, traits>& rhs);

Effects: Calls lhs.swap(rhs).

28.9 Class template sub_match

Class template sub_match denotes the sequence of characters matched by a particular marked sub-expression.

namespace std {
    template <class BidirectionalIterator>
    class sub_match : public std::pair<BidirectionalIterator, BidirectionalIterator> {
        public:
            typedef typename iterator_traits<BidirectionalIterator>::
                value_type value_type;
            typedef typename iterator_traits<BidirectionalIterator>::
                difference_type difference_type;
            typedef BidirectionalIterator iterator;
            typedef basic_string<value_type> string_type;

            bool matched;

            difference_type length() const;
            operator string_type() const;
            string_type str() const;

            int compare(const sub_match& s) const;
            int compare(const string_type& s) const;
            int compare(const value_type* s) const;
    }
};

28.9.1 sub_match members

difference_type length() const;

§ 28.9.1
operator string_type() const;

Returns: matched ? string_type(first, second) : string_type().

string_type str() const;

Returns: matched ? string_type(first, second) : string_type().

int compare(const sub_match& s) const;

Returns: str().compare(s.str()).

int compare(const string_type& s) const;

Returns: str().compare(s).

int compare(const value_type* s) const;

Returns: str().compare(s).

### 28.9.2 sub_match non-member operators

template <class BiIter>
bool operator==(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs.compare(rhs) == 0.

template <class BiIter>
bool operator!=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs.compare(rhs) != 0.

template <class BiIter>
bool operator<(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs.compare(rhs) < 0.

template <class BiIter>
bool operator<=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs.compare(rhs) <= 0.

template <class BiIter>
bool operator>(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs.compare(rhs) > 0.

template <class BiIter, class ST, class SA>
bool operator==(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);
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().

8.

template <class BiIter, class ST, class SA>
bool operator<(const sub_match<BiIter>& lhs,
    const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

Returns: lhs < rhs.str().

9.

template <class BiIter, class ST, class SA>
bool operator>(const sub_match<BiIter>& lhs,
    const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

Returns: lhs > rhs.str().

10.

template <class BiIter, class ST, class SA>
bool operator>=(const sub_match<BiIter>& lhs,
    const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

Returns: lhs >= rhs.str().

11.

template <class BiIter, class ST, class SA>
bool operator<=(const sub_match<BiIter>& lhs,
    const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

Returns: lhs <= rhs.str().

12.

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.

13.

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.

§ 28.9.2
template <class BiIter, class ST, class SA>
bool operator<(const sub_match<BiIter>& lhs,
const basic_string<
    typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

Returns: lhs.str() < rhs.

template <class BiIter, class ST, class SA>
bool operator>(const sub_match<BiIter>& lhs,
const basic_string<
    typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

Returns: lhs.str() > rhs.

template <class BiIter, class ST, class SA>
bool operator>=(const sub_match<BiIter>& lhs,
const basic_string<
    typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

Returns: lhs.str() >= rhs.

template <class BiIter, class ST, class SA>
bool operator<=(const sub_match<BiIter>& lhs,
const basic_string<
    typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

Returns: lhs.str() <= rhs.

template <class BiIter>
bool operator==(typename iterator_traits<BiIter>::value_type const* lhs,
const sub_match<BiIter>& rhs);

Returns: lhs == rhs.str().

template <class BiIter>
bool operator!=(typename iterator_traits<BiIter>::value_type const* lhs,
const sub_match<BiIter>& rhs);

Returns: lhs != rhs.str().

template <class BiIter>
bool operator<(typename iterator_traits<BiIter>::value_type const* lhs,
const sub_match<BiIter>& rhs);

Returns: lhs < rhs.str().

template <class BiIter>
bool operator>(typename iterator_traits<BiIter>::value_type const* lhs,
const sub_match<BiIter>& rhs);

Returns: lhs > rhs.str().

template <class BiIter>
bool operator>=(typename iterator_traits<BiIter>::value_type const* lhs,
const sub_match<BiIter>& rhs);

Returns: lhs >= rhs.str().
template <class BiIter>
bool operator<=(typename iterator_traits<BiIter>::value_type const* lhs,
               const sub_match<BiIter>& rhs);

    Returns: lhs <= rhs.str().

template <class BiIter>
bool operator==(const sub_match<BiIter>& lhs,
               typename iterator_traits<BiIter>::value_type const* rhs);

    Returns: lhs.str() == rhs.

template <class BiIter>
bool operator!=(const sub_match<BiIter>& lhs,
               typename iterator_traits<BiIter>::value_type const* rhs);

    Returns: lhs.str() != rhs.

template <class BiIter>
bool operator<(const sub_match<BiIter>& lhs,
               typename iterator_traits<BiIter>::value_type const* rhs);

    Returns: lhs.str() < rhs.

template <class BiIter>
bool operator>(const sub_match<BiIter>& lhs,
               typename iterator_traits<BiIter>::value_type const* rhs);

    Returns: lhs.str() > rhs.

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<Bi Iter>::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);

§ 28.9.2 1086
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& rhs);

Returns: lhs.str() < basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs).

template <class BiIter>
bool operator!=(const sub_match<BiIter>& lhs,
const sub_match<BiIter>& rhs);

Returns: lhs.str() != basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs).

template <class BiIter>
bool operator<=(const sub_match<BiIter>& rhs);

Returns: lhs.str() <= basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs).

template <class BiIter>
bool operator>=(const sub_match<BiIter>& rhs);

Returns: lhs.str() >= basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs).

§ 28.9.2
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

1 Class template `match_results` denotes a collection of character sequences representing the result of a regular expression match. Storage for the collection is allocated and freed as necessary by the member functions of class template `match_results`.

2 The class template `match_results` shall satisfy the requirements of an allocator-aware container and of a sequence container, as specified in 23.2.3, except that only operations defined for const-qualified sequence containers are supported.

3 The `sub_match` object stored at index 0 represents sub-expression 0, i.e., the whole match. In this case the `sub_match` member `matched` is always true. The `sub_match` object stored at index `n` denotes what matched the marked sub-expression `n` within the matched expression. If the sub-expression `n` participated in a regular expression match then the `sub_match` member `matched` evaluates to true, and members `first` and `second` denote the range of characters `[first,second)` which formed that match. Otherwise `matched` is false, and members `first` and `second` point to the end of the sequence that was searched. [Note: The `sub_match` objects representing different sub-expressions that did not participate in a regular expression match need not be distinct. — end note]

```cpp
namespace std {
    template <class BidirectionalIterator,
        class Allocator = allocator<sub_match<BidirectionalIterator> >
    class match_results {
    public:
        typedef sub_match<BidirectionalIterator> value_type;
        typedef const value_type& const_reference;
        typedef const_reference reference;
        typedef implementation-defined const_iterator;
        typedef const_iterator iterator;
        typedef typename iterator_traits<BidirectionalIterator>::difference_type difference_type;
        typedef typename allocator_traits<Allocator>::size_type size_type;
        typedef Allocator allocator_type;
        typedef typename iterator_traits<BidirectionalIterator>::value_type char_type;
        typedef basic_string<char_type> string_type;

        // 28.10.1, construct/copy/destroy:
        explicit match_results(const Allocator& a = Allocator());
        match_results(const match_results& m);
        match_results(match_results&& m);
        match_results& operator=(const match_results& m);
        match_results& operator=(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;

cost_reference prefix() const;
cost_reference suffix() const;
cost_iterator begin() const;
cost_iterator end() const;
cost_iterator cbegin() const;
cost_iterator cend() const;

// 28.10.4, format:
template <class OutputIter>
OutputIter format(OutputIter out,
    const char_type* fmt_first, const char_type* fmt_last,
    regex_constants::match_flag_type flags =
    regex_constants::format_default) const;
template <class OutputIter, class ST, class SA>
OutputIter format(OutputIter out,
    const basic_string<char_type, ST, SA>& fmt,
    regex_constants::match_flag_type flags =
    regex_constants::format_default) const;
template <class ST, class SA>
basic_string<char_type, ST, SA> format(const basic_string<char_type, ST, SA>& fmt,
    regex_constants::match_flag_type flags =
    regex_constants::format_default) const;

string_type format(const char_type* fmt,
    regex_constants::match_flag_type flags =
    regex_constants::format_default) const;

// 28.10.5, allocator:
allocator_type get_allocator() const;

// 28.10.6, swap:
void swap(match_results& that);
};
}

28.10.1 match_results constructors [re.results.const]

1 In all match_results constructors, a copy of the Allocator argument shall be used for any memory allocation performed by the constructor or member functions during the lifetime of the object.

match_results(const Allocator& a = Allocator());

2 Effects: Constructs an object of class match_results.

Postconditions: size() returns 0. str() returns basic_string<char_type>().
match_results(const match_results& m);

**Effects:** Constructs an object of class `match_results`, as a copy of m.

match_results(match_results&& m);

**Effects:** Move-constructs an object of class `match_results` from m satisfying the same postconditions as Table 138. Additionally, the stored `Allocator` value is move constructed from `m.get_allocator()`. After the initialization of `*this`, sets m to an unspecified but valid state.

**Throws:** Nothing if the allocator's move constructor throws nothing.

match_results& operator=(const match_results& m);

**Effects:** Assigns m to `*this`. The postconditions of this function are indicated in Table 138.

match_results& operator=(match_results&& m);

**Effects:** Move-assigns m to `*this`. The postconditions of this function are indicated in Table 138. After the assignment, m is in a valid but unspecified state.

**Throws:** Nothing.

### Table 138 — `match_results` assignment operator effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>size()</td>
<td>m.size()</td>
</tr>
<tr>
<td>str(n)</td>
<td><code>m.str(n)</code> for all integers <code>n &lt; m.size()</code></td>
</tr>
<tr>
<td>prefix()</td>
<td>m.prefix()</td>
</tr>
<tr>
<td>suffix()</td>
<td>m.suffix()</td>
</tr>
<tr>
<td>(<code>*this</code>)[n]</td>
<td><code>m[n]</code> for all integers <code>n &lt; m.size()</code></td>
</tr>
<tr>
<td>length(n)</td>
<td>m.length(n) for all integers <code>n &lt; m.size()</code></td>
</tr>
<tr>
<td>position(n)</td>
<td>m.position(n) for all integers <code>n &lt; m.size()</code></td>
</tr>
</tbody>
</table>

#### 28.10.2 `match_results` size

`size_type size() const;`

**Returns:** One plus the number of marked sub-expressions in the regular expression that was matched if `*this` represents the result of a successful match. Otherwise returns 0.  

[Note: The state of a `match_results` object can be modified only by passing that object to `regex_match` or `regex_search`. Sections 28.11.2 and 28.11.3 specify the effects of those algorithms on their `match_results` arguments. — end note]

`size_type max_size() const;`

**Returns:** The maximum number of `sub_match` elements that can be stored in `*this`.

`bool empty() const;`

**Returns:** `size() == 0`.  

§ 28.10.2
28.10.3 \texttt{match\_results} element access \[re.results.acc\]

\begin{verbatim}
difference_type length(size_type sub = 0) const;
    \textit{Returns}: \texttt{(*this)[sub].length()}.

difference_type position(size_type sub = 0) const;
    \textit{Returns}: The distance from the start of the target sequence to \texttt{(*this)[sub].first}.

string_type str(size_type sub = 0) const;
    \textit{Returns}: \texttt{string_type\(((*this)[sub])\)}.

const_reference operator[]\(\)(size_type n) const;
    \textit{Returns}: A reference to the \texttt{sub\_match} object representing the character sequence that matched marked sub-expression \textit{n}. If \textit{n} \(\leq\) \textit{0} then returns a reference to a \texttt{sub\_match} object representing the character sequence that matched the whole regular expression. If \textit{n} \(\geq\) \texttt{size()} then returns a \texttt{sub\_match} object representing an unmatched sub-expression.

const_reference prefix() const;
    \textit{Returns}: A reference to the \texttt{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;
    \textit{Returns}: A reference to the \texttt{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;
    \textit{Returns}: A starting iterator that enumerates over all the sub-expressions stored in \texttt{*this}.

const_iterator end() const;
const_iterator cend() const;
    \textit{Returns}: A terminating iterator that enumerates over all the sub-expressions stored in \texttt{*this}.
\end{verbatim}

28.10.4 \texttt{match\_results} formatting \[re.results.form\]

\begin{verbatim}
template <class OutputIter>
    OutputIter format(OutputIter out,
        const char_type* fmt_first, const char_type* fmt_last,
        regex_constants::match_flag_type flags =
            regex_constants::format_default) const;
\end{verbatim}

\begin{verbatim}
    Requires: OutputIter shall satisfy the requirements for an Output Iterator (24.2.4).
    Effects: Copies the character sequence \([\texttt{fmt\_first},\texttt{fmt\_last}]\) to OutputIter \texttt{out}. Replaces each format specifier or escape sequence in the copied range with either the character(s) it represents or the sequence of characters within \texttt{*this} to which it refers. The bitmasks specified in \texttt{flags} determine which format specifiers and escape sequences are recognized.
    Returns: \texttt{out}.
\end{verbatim}
template <class OutputIter, class ST, class SA>
    OutputIter format(OutputIter out,
        const basic_string<char_type, ST, SA>& fmt,
        regex_constants::match_flag_type flags =
            regex_constants::format_default) const;

    Effects: Equivalent to return format(out, fmt.data(), fmt.data() + fmt.size(), flags).

template <class ST, class SA>
    basic_string<char_type, ST, SA>
        format(const basic_string<char_type, ST, SA>& fmt,
            regex_constants::match_flag_type flags =
                regex_constants::format_default) const;

    Effects: Constructs an empty string result of type basic_string<char_type, ST, SA> and calls
    format(back_inserter(result), fmt, flags).

    Returns: result.

string_type
    format(const char_type* fmt,
            regex_constants::match_flag_type flags =
                regex_constants::format_default) const;

    Effects: Constructs an empty string result of type string_type and calls
    format(back_inserter(result), fmt, fmt + char_traits<char_type>::length(fmt), flags).

    Returns: result.

28.10.5 match_results allocator

allocator_type get_allocator() const;

    Returns: a copy of the Allocator that was passed to the object’s constructor or, if that allocator has
    been replaced, a copy of the most recent replacement.

28.10.6 match_results swap

void swap(match_results& that);

    Effects: Swaps the contents of the two sequences.

    Postcondition: *this contains the sequence of matched sub-expressions that were in that, that con-
    tains the sequence of matched sub-expressions that were in *this.

    Complexity: constant time.

template <class BidirectionalIterator, class Allocator>
    void swap(match_results<BidirectionalIterator, Allocator>& m1,
              match_results<BidirectionalIterator, Allocator>& m2);

    Effects: m1.swap(m2).

28.10.7 match-results non-member functions
template <class BidirectionalIterator, class Allocator>
bool operator==(const match_results<BidirectionalIterator, Allocator>& m1,
                const match_results<BidirectionalIterator, Allocator>& m2);

Returns: true 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]

Requires: The type BidirectionalIterator shall satisfy the requirements of a Bidirectional Iterator (24.2.6).

Effects: Determines whether there is a match between the regular expression e, and all of the character sequence [first,last). The parameter flags is used to control how the expression is matched against the character sequence. Returns true if such a match exists, false otherwise.

Postconditions: 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 139.

Table 139 — Effects of regex_match algorithm

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>m.size()</td>
<td>1 + e.mark_count()</td>
</tr>
<tr>
<td>m.empty()</td>
<td>false</td>
</tr>
<tr>
<td>m.prefix().first</td>
<td>first</td>
</tr>
<tr>
<td>m.prefix().second</td>
<td>first</td>
</tr>
<tr>
<td>m.prefix().matched</td>
<td>false</td>
</tr>
<tr>
<td>m.suffix().first</td>
<td>last</td>
</tr>
<tr>
<td>m.suffix().second</td>
<td>last</td>
</tr>
<tr>
<td>m.suffix().matched</td>
<td>false</td>
</tr>
<tr>
<td>m[0].first</td>
<td>first</td>
</tr>
<tr>
<td>m[0].second</td>
<td>last</td>
</tr>
<tr>
<td>m[0].matched</td>
<td>true if a full match was found.</td>
</tr>
</tbody>
</table>
Table 139 — Effects of regex_match algorithm (continued)

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>m[n].first</td>
<td>For all integers n &lt; m.size(), the start of the sequence that matched sub-expression n. Alternatively, if sub-expression n did not participate in the match, then last.</td>
</tr>
<tr>
<td>m[n].second</td>
<td>For all integers n &lt; m.size(), the end of the sequence that matched sub-expression n. Alternatively, if sub-expression n did not participate in the match, then last.</td>
</tr>
<tr>
<td>m[n].matched</td>
<td>For all integers n &lt; m.size(), true if sub-expression n participated in the match, false otherwise.</td>
</tr>
</tbody>
</table>

template <class BidirectionalIterator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
const basic_regex<charT, traits>& e,
regex_constants::match_flag_type flags =
regex_constants::match_default);

4 Effects: Behaves “as if” by constructing an instance of match_results<BidirectionalIterator> what, and then returning the result of regex_match(first, last, what, e, flags).

template <class charT, class Allocator, class traits>
bool regex_match(const charT* str,
match_results<const charT*, Allocator>& m,
const basic_regex<charT, traits>& e,
regex_constants::match_flag_type flags =
regex_constants::match_default);

5 Returns: regex_match(str, str + char_traits<charT>::length(str), m, e, flags).

template <class ST, class SA, class Allocator, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
match_results<
typename basic_string<charT, ST, SA>::const_iterator,
Allocator>& m,
const basic_regex<charT, traits>& e,
regex_constants::match_flag_type flags =
regex_constants::match_default);

6 Returns: regex_match(s.begin(), s.end(), m, e, flags).

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);

7 Returns: regex_match(str, str + char_traits<charT>::length(str), e, flags)

template <class ST, class SA, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
const basic_regex<charT, traits>& e,
regex_constants::match_flag_type flags =
regex_constants::match_default);
Returns: `regex_match(s.begin(), s.end(), e, flags)`.

### 28.11.3 `regex_search` [re.alg.search]

```cpp
template <class BidirectionalIterator, class Allocator, class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last, 
                 match_results<BidirectionalIterator, Allocator>& m, 
                 const basic_regex<charT, traits>& e, 
                 regex_constants::match_flag_type flags = 
                 regex_constants::match_default);
```

**Requires:** Type `BidirectionalIterator` shall satisfy the requirements of a Bidirectional Iterator (24.1.4).

**Effects:** Determines whether there is some sub-sequence within `[first,last)` that matches the regular expression `e`. The parameter `flags` is used to control how the expression is matched against the character sequence. Returns `true` if such a sequence exists, `false` otherwise.

**Postconditions:** 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 140.

<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 <code>false</code> otherwise.</td>
</tr>
<tr>
<td>m[n].first</td>
<td>For all integers <code>n &lt; m.size()</code>, the start of the sequence that matched sub-expression <code>n</code>. Alternatively, if sub-expression <code>n</code> did not participate in the match, then <code>last</code>.</td>
</tr>
<tr>
<td>m[n].second</td>
<td>For all integers <code>n &lt; m.size()</code>, the end of the sequence that matched sub-expression <code>n</code>. Alternatively, if sub-expression <code>n</code> did not participate in the match, then <code>last</code>.</td>
</tr>
<tr>
<td>m[n].matched</td>
<td>For all integers <code>n &lt; m.size()</code>, <code>true</code> if sub-expression <code>n</code> participated in the match, <code>false</code> otherwise.</td>
</tr>
</tbody>
</table>

```cpp
template <class charT, class Allocator, class traits>
bool regex_search(const charT* str, match_results<const charT*, Allocator>& m, 
                  const basic_regex<charT, traits>& e, 
                  regex_constants::match_flag_type flags = 
                  regex_constants::match_default);
```

§ 28.11.3
 regex_constants::match_default);

4 Returns: The result of regex_search(str, str + char_traits<charT>::length(str), m, e, flags).

template <class ST, class SA, class Allocator, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
    match_results<
        typename basic_string<charT, ST, SA>::const_iterator,
        Allocator>& m,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags =
    regex_constants::match_default);

5 Returns: The result of 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);

6 Effects: Behaves “as if” by constructing an object what of type match_results<BidirectionalIterator>
and then returning the result of regex_search(first, last, what, e, flags).

template <class charT, class traits>
bool regex_search(const charT* str,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags =
    regex_constants::match_default);

7 Returns: regex_search(str, str + char_traits<charT>::length(str), e, flags)

template <class ST, class SA, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags =
    regex_constants::match_default);

8 Returns: regex_search(s.begin(), s.end(), e, flags).

28.11.4 regex_replace

[re.alg.replace]

template <class OutputIterator, class BidirectionalIterator,
    class traits, class charT, class ST, class SA>
OutputIterator
regex_replace(OutputIterator out,
    BidirectionalIterator first, BidirectionalIterator last,
    const basic_regex<charT, traits>& e,
    const basic_string<charT, ST, SA>& fmt,
    regex_constants::match_flag_type flags =
    regex_constants::match_default);

template <class OutputIterator, class BidirectionalIterator,
    class traits, class charT>
OutputIterator
regex_replace(OutputIterator out,
    BidirectionalIterator first, BidirectionalIterator last,
const basic_regex<charT, traits>& e,
const charT* fmt,
regex_constants::match_flag_type flags =
regex_constants::match_default);

Effects: Constructs a regex_iterator object i as if by
regex_iterator<BidirectionalIterator,
charT, traits> i(first, last, e, flags), and uses i to enumerate through all of the matches
m of type match_results<BidirectionalIterator> that occur within the sequence [first,last).
If no such matches are found and !(flags & regex_constants::format_no_copy) then calls
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) for the first form of the function and m.format(out, fmt, fmt + char_traits<charT>::length(fmt), flags) for the second.

Finally, if such a match is found and !(flags & regex_constants::format_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, class ST, class SA, class FST, class FSA>
basic_string<charT, ST, SA>
regex_replace(const basic_string<charT, ST, SA>& s,
const basic_regex<charT, traits>& e,
const basic_string<charT, FST, FSA>& fmt,
regex_constants::match_flag_type flags =
regex_constants::match_default);

template <class traits, class charT, class ST, class SA>
basic_string<charT, ST, SA>
regex_replace(const basic_string<charT, ST, SA>& s,
const basic_regex<charT, traits>& e,
const charT* fmt,
regex_constants::match_flag_type flags =
regex_constants::match_default);

Effects: Constructs an empty string result of type basic_string<charT, ST, SA> and calls
regex_replace(back_inserter(result), s.begin(), s.end(), e, fmt, flags).

Returns: result.

template <class traits, class charT, class ST, class SA>
basic_string<charT>
regex_replace(const charT* s,
const basic_regex<charT, traits>& e,
const basic_string<charT, ST, SA>& fmt,
regex_constants::match_flag_type flags =
regex_constants::match_default);

template <class traits, class charT>
basic_string<charT>
regex_replace(const charT* s,
const basic_regex<charT, traits>& e,
const charT* fmt,
regex_constants::match_flag_type flags =
regex_constants::match_default);
Effects: Constructs an empty string `result` of type `basic_string<charT>` and calls `regex_replace(back_inserter(result), s, s + char_traits<charT>::length(s), e, fmt, flags)`.

Returns: `result`.

### 28.12 Regular expression Iterators

#### 28.12.1 Class template `regex_iterator`

The class template `regex_iterator` is an iterator adaptor. It represents a new view of an existing iterator sequence by enumerating all the occurrences of a regular expression within that sequence. A `regex_iterator` uses `regex_search` to find successive regular expression matches within the sequence from which it was constructed. After the iterator is constructed, and every time `operator++` is used, the iterator finds and stores a value of `match_results<BidirectionalIterator>`. If the end of the sequence is reached (`regex_search` returns `false`), the iterator becomes equal to the end-of-sequence iterator value. The default constructor constructs an end-of-sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end-of-sequence iterator is not defined. For any other iterator value a `const match_results<BidirectionalIterator>&` is returned. The result of `operator->` on an end-of-sequence iterator is not defined. For any other iterator value a `const match_results<BidirectionalIterator>**` is returned. It is impossible to store things into `regex_iterator`s.

Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.

```cpp
namespace std {
    template <class BidirectionalIterator,
             class charT = typename iterator_traits<BidirectionalIterator>::value_type,
             class traits = regex_traits<charT> >
    class regex_iterator {
        public:
            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,
                           const 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&);
            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;
    }
}
```
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

```cpp
regex_iterator();
```

**Effects:** Constructs an end-of-sequence iterator.

```cpp
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

```cpp
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`.

```cpp
bool operator!=(const regex_iterator& right) const;
```

**Returns:** `!(*this == right)`.

### 28.12.1.3 regex_iterator dereference

```cpp
const value_type& operator*() const;
```

**Returns:** `match`.

```cpp
const value_type* operator->() const;
```

**Returns:** `&match`.

### 28.12.1.4 regex_iterator increment

```cpp
regex_iterator& operator++();
```

**Effects:** Constructs a local variable `start` of type `BidirectionalIterator` and initializes it with the value of `match[0].second`.

§ 28.12.1.4 1099
If the iterator holds a zero-length match and \texttt{start} == \texttt{end} the operator sets \texttt{*this} to the end-of-sequence iterator and returns \texttt{*this}.

Otherwise, if the iterator holds a zero-length match the operator calls \texttt{regex_search(start, end, match, *pregex, flags | regex_constants::match_not_null | regex_constants::match_continuous)}. If the call returns \texttt{true} the operator returns \texttt{*this}. Otherwise the operator increments \texttt{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 \texttt{flags} to \texttt{flags | regex_constants::match_prev_avail} and calls \texttt{regex_search(start, end, match, *pregex, flags)}. If the call returns \texttt{false} the iterator sets \texttt{*this} to the end-of-sequence iterator. The iterator then returns \texttt{*this}.

In all cases in which the call to \texttt{regex_search} returns \texttt{true}, \texttt{match.prefix().first} shall be equal to the previous value of \texttt{match[0].second}, and for each index \texttt{i} in the half-open range \texttt{[0, match.size())} for which \texttt{match[i].matched} is true, \texttt{match[i].position()} shall return \texttt{distance(begin, match[i].first)}.

[\textit{Note:} this means that \texttt{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 \texttt{regex_search}. — end note]

It is unspecified how the implementation makes these adjustments.

[\textit{Note:} this means that a compiler may call an implementation-specific search function, in which case a user-defined specialization of \texttt{regex_search} will not be called. — end note]

\begin{verbatim}
regex_iterator operator++(int);
\end{verbatim}

\begin{verbatim}
Effects:

regex_iterator tmp = *this;
++(*this);
return tmp;
\end{verbatim}

\section*{28.12.2 Class template \texttt{regex_token_iterator}}

The class template \texttt{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 \texttt{sub_match} class template instance that represents what matched a particular sub-expression within the regular expression.

When class \texttt{regex_token_iterator} is used to enumerate a single sub-expression with index \texttt{-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 \texttt{regex_iterator\langle BidirectionalIterator\rangle position} and sets the internal count \texttt{N} to zero. It also maintains a sequence \texttt{subs} which contains a list of the sub-expressions which will be enumerated. Every time \texttt{operator++} is used the count \texttt{N} is incremented; if \texttt{N} exceeds or equals \texttt{subs.size()}, then the iterator increments member \texttt{position} and sets count \texttt{N} to zero.

If the end of sequence is reached (\texttt{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 \texttt{-1}, in which case the iterator enumerates one last sub-expression that contains all the characters from the end of the last regular expression match to the end of the input sequence being enumerated, provided that this would not be an empty sub-expression.
The default constructor constructs an end-of-sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end-of-sequence iterator is not defined. For any other iterator value a `const sub_match<BidirectionalIterator>&` is returned. The result of `operator->` on an end-of-sequence iterator is not defined. For any other iterator value a `const sub_match<BidirectionalIterator>*` is returned.

It is impossible to store things into `regex_token_iterator`s. Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.

```cpp
namespace std {
  template <class BidirectionalIterator,
           class charT = typename iterator_traits<BidirectionalIterator>::value_type,
           class traits = regex_traits<charT> >
  class regex_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);
      regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                           const regex_type& re,
                           initializer_list<int> submatches,
                           regex_constants::match_flag_type m =
                           regex_constants::match_default);
      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);
      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;
  }
```
A suffix iterator is a `regex_token_iterator` object that points to a final sequence of characters at the end of the target sequence. In a suffix iterator the member `result` holds a pointer to the data member `suffix`, the value of the member `suffix.match` is `true`, `suffix.first` points to the beginning of the final sequence, and `suffix.second` points to the end of the final sequence.

[Note: for a suffix iterator, data member `suffix.first` is the same as the end of the last match found, and `suffix.second` is the same as the end of the target sequence — end note]

The current match is `(*position).prefix()` if `subs[N] == -1`, or `(*position)[subs[N]]` for any other value of `subs[N].`

### 28.12.2.1 regex_token_iterator constructors

#### Effects: Constructs the end-of-sequence iterator.

```cpp
regex_token_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
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b, const regex_type& re,
initializer_list<int> submatches,
regex_constants::match_flag_type m = regex_constants::match_default);
```

```cpp
template <std::size_t N>
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
const regex_type& re,
const int (&submatches)[N],
regex_constants::match_flag_type m = regex_constants::match_default);
```

#### Requires: Each of the initialization values of `submatches` shall be `>= -1`.

#### Effects: The first constructor initializes the member `subs` to hold the single value `submatch`. The second constructor initializes the member `subs` to hold a copy of the argument `submatches`. The third and fourth constructors initialize the member `subs` to hold a copy of the sequence of integer values pointed
to by the iterator range \([\text{submatches.begin()}, \text{submatches.end()}]\) and \([&\text{submatches}, &\text{submatches} + N]\), respectively.

Each constructor then sets \(N\) to 0, and \(\text{position}\) to \(\text{position}\_\text{iterator}(a, b, \text{re}, m)\). If \(\text{position}\) is not an end-of-sequence iterator the constructor sets \(\text{result}\) to the address of the current match. Otherwise if any of the values stored in \(\text{subs}\) is equal to -1 the constructor sets \(*\text{this}\) to a suffix iterator that points to the range \([a, b)\), otherwise the constructor sets \(*\text{this}\) to an end-of-sequence iterator.

### 28.12.2.2 regex_token_iterator comparisons

```plaintext
bool operator==(const regex_token_iterator& right) const;
```

**Returns:** true if \(*\text{this}\) and \(\text{right}\) are both end-of-sequence iterators, or if \(*\text{this}\) and \(\text{right}\) are both suffix iterators and \(\text{suffix} == \text{right.suffix}\); otherwise returns false if \(*\text{this}\) or \(\text{right}\) is an end-of-sequence iterator or a suffix iterator. Otherwise returns true if \(\text{position} == \text{right}\_\text{position}\), \(N == \text{right}\_N\), and \(\text{subs} == \text{right}\_\text{subs}\). Otherwise returns false.

```plaintext
bool operator!=(const regex_token_iterator& right) const;
```

**Returns:** \(!(*\text{this} == \text{right})\).

### 28.12.2.3 regex_token_iterator dereference

```plaintext
const value_type& operator*() const;
```

**Returns:** \(*\text{result}\).

```plaintext
const value_type* operator->() const;
```

**Returns:** \(\text{result}\).

### 28.12.2.4 regex_token_iterator increment

```plaintext
regex_token_iterator& operator++();
```

**Effects:** Constructs a local variable \(\text{prev}\) of type \(\text{position}\_\text{iterator}\), initialized with the value of \(\text{position}\).

1. If \(*\text{this}\) is a suffix iterator, sets \(*\text{this}\) to an end-of-sequence iterator.

2. Otherwise, if \(N + 1 < \text{subs}\_\text{size}\), increments \(N\) and sets \(\text{result}\) to the address of the current match.

3. Otherwise, sets \(N\) to 0 and increments \(\text{position}\). If \(\text{position}\) is not an end-of-sequence iterator the operator sets \(\text{result}\) to the address of the current match.

4. Otherwise, if any of the values stored in \(\text{subs}\) is equal to -1 and \(\text{prev}\_\text{suffix}\_\text{length}\) is not 0 the operator sets \(*\text{this}\) to a suffix iterator that points to the range \([\text{prev}\_\text{suffix}\_\text{first}, \text{prev}\_\text{suffix}\_\text{second})\).

5. Otherwise, sets \(*\text{this}\) to an end-of-sequence iterator.

**Returns:** \(*\text{this}\)

```plaintext
regex_token_iterator& operator++(int);
```

§ 28.12.2.4
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.

The following productions within the ECMAScript grammar are modified as follows:

```
ClassAtom ::
- ClassAtomNoDash
ClassAtomExClass
ClassAtomCollatingElement
ClassAtomEquivalence
```

The following new productions are then added:

```
ClassAtomExClass ::
[ : ClassName : ]
ClassAtomCollatingElement ::
[ . ClassName . ]
ClassAtomEquivalence ::
[ = ClassName = ]
```

```
ClassName ::
ClassNameCharacter
ClassNameCharacter ClassName
```

```
ClassNameCharacter ::
SourceCharacter but not one of "." "=" "::
```

The productions `ClassAtomExClass`, `ClassAtomCollatingElement` and `ClassAtomEquivalence` provide functionality equivalent to that of the same features in regular expressions in POSIX.

The regular expression grammar may be modified by any `regex_constants::syntax_option_type` flags specified when constructing an object of type specialization of `basic_regex` according to the rules in table 135.

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:]]
```
A `ClassName` production when used in a `ClassAtomCollatingElement` production is not valid if the value returned by `traits_inst.lookup_collatename` for that name is an empty string.

The results from multiple calls to `traits_inst.lookup_classname` can be bitwise OR’ed together and subsequently passed to `traits_inst.isctype`.

A `ClassName` production when used in a `ClassAtomEquivalence` production is not valid if the value returned by `traits_inst.lookup_collatename` for that name is an empty string or if the value returned by `traits_inst.transform_primary` for the result of the call to `traits_inst.lookup_collatename` is an empty string.

When the sequence of characters being transformed to a finite state machine contains an invalid class name the translator shall throw an exception object of type `regex_error`.

If the `CV` of a `UnicodeEscapeSequence` is greater than the largest value that can be held in an object of type `charT` the translator shall throw an exception object of type `regex_error`. [Note: this means that values of the form "uxxxx" that do not fit in a character are invalid. — end note]

Where the regular expression grammar requires the conversion of a sequence of characters to an integral value, this is accomplished by calling `traits_inst.value`.

The behavior of the internal finite state machine representation when used to match a sequence of characters is as described in ECMA-262. The behavior is modified according to any match_flag_type flags 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 `(flags() & regex_constants::icase)` the two characters are equal if `traits_inst.translate_nocase(c) == traits_inst.translate_nocase(d)`;
  2. otherwise, if `flags() & regex_constants::collate` the two characters are equal if `traits_inst.translate(c) == traits_inst.translate(d)`;
  3. otherwise, the two characters are equal if `c == d`.

- During matching of a regular expression finite state machine against a sequence of characters, comparison of a collating element range `c1-c2` against a character `c` is conducted as follows: if `flags() & regex_constants::collate` is false then the character `c` is matched if `c1 <= c && c <= c2`, otherwise `c` is matched in accordance with the following algorithm:

```cpp
string_type str1 = string_type(1,
  flags() & icase ?
  traits_inst.translate_nocase(c1) : traits_inst.translate(c1);
string_type str2 = string_type(1,
  flags() & icase ?
  traits_inst.translate_nocase(c2) : traits_inst.translate(c2);
string_type str = string_type(1,
```
flags() & icase ?
    traits_inst.translate_nocase(c) : traits_inst.translate(c);
return traits_inst.transform(str1.begin(), str1.end())
<= traits_inst.transform(str.begin(), str.end())
&& traits_inst.transform(str.begin(), str.end())
<= traits_inst.transform(str2.begin(), str2.end());

— During matching of a regular expression finite state machine against a sequence of characters, testing
whether a collating element is a member of a primary equivalence class is conducted by first converting
the collating element and the equivalence class to sort keys using traits::transform_primary, and
then comparing the sort keys for equality.

— During matching of a regular expression finite state machine against a sequence of characters, a char-
acter c is a member of a character class designated by an iterator range [first,last) if traits_
inst.isctype(c, traits_inst.lookup_classname(first, last, flags() & icase)) is true.
29 Atomic operations library

29.1 General

This Clause describes components for fine-grained atomic access. This access is provided via operations on atomic objects.\(^{341}\)

The following subclauses describe atomics requirements and components for types and operations, as summarized below.

Table 141 — Atomics library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.3</td>
<td>Order and Consistency</td>
</tr>
<tr>
<td>29.4</td>
<td>Lock-free Property</td>
</tr>
<tr>
<td>29.5</td>
<td>Atomic Types</td>
</tr>
<tr>
<td></td>
<td>&lt;atomic&gt;</td>
</tr>
<tr>
<td>29.6</td>
<td>Operations on Atomic Types</td>
</tr>
<tr>
<td>29.7</td>
<td>Flag Type and Operations</td>
</tr>
</tbody>
</table>

29.2 Header <atomic> synopsis

namespace std {
    // 29.3, order and consistency
    enum memory_order;
    template <class T>
        T kill_dependency(T y);

    // 29.4, lock-free property
    #define ATOMIC_INTEGRAL_LOCK_FREE unspecified
    #define ATOMIC_ADDRESS_LOCK_FREE unspecified

    // 29.7, flag type and operations
    struct atomic_flag;
    bool atomic_flag_test_and_set(volatile atomic_flag*);
    bool atomic_flag_test_and_set(atomic_flag*);
    bool atomic_flag_test_and_set_explicit(volatile atomic_flag*, memory_order);
    bool atomic_flag_test_and_set_explicit(atomic_flag*, memory_order);
    void atomic_flag_clear(volatile atomic_flag*);
    void atomic_flag_clear_explicit(volatile atomic_flag*, memory_order);
    void atomic_flag_clear_explicit(atomic_flag*, memory_order);

    #define ATOMIC_VAR_INIT(value) see below

    // 29.5.1, integral types
    struct atomic_bool;
    bool atomic_is_lock_free(const volatile atomic_bool*);\(^{341}\)

Atomic objects are neither active nor radioactive.

\(^{341}\) Atomic objects are neither active nor radioactive.
bool atomic_is_lock_free(const atomic_bool*);
void atomic_store(volatile atomic_bool*, bool);
void atomic_store(atomic_bool*, bool);
void atomic_store_explicit(volatile atomic_bool*, bool, memory_order);
void atomic_store_explicit(atomic_bool*, bool, memory_order);
bool atomic_load(const volatile atomic_bool*);
bool atomic_load(const atomic_bool*);
bool atomic_load_explicit(const volatile atomic_bool*, memory_order);
bool atomic_load_explicit(const atomic_bool*, memory_order);
bool atomic_exchange(volatile atomic_bool*, bool);
bool atomic_exchange(atomic_bool*, bool);
bool atomic_exchange_explicit(volatile atomic_bool*, bool, memory_order);
bool atomic_exchange_explicit(atomic_bool*, bool, memory_order);
bool atomic_compare_exchange_weak(volatile atomic_bool*, bool*, bool);
bool atomic_compare_exchange_weak(atomic_bool*, bool*, bool);
bool atomic_compare_exchange_strong(volatile atomic_bool*, bool*, bool);
bool atomic_compare_exchange_strong(atomic_bool*, bool*, bool);
bool atomic_compare_exchange_weak_explicit(volatile atomic_bool*, bool*, bool, memory_order, memory_order);
bool atomic_compare_exchange_weak_explicit(atomic_bool*, bool*, bool, memory_order, memory_order);
bool atomic_compare_exchange_strong_explicit(volatile atomic_bool*, bool*, bool, memory_order, memory_order);
bool atomic_compare_exchange_strong_explicit(atomic_bool*, bool*, bool, memory_order, memory_order);

// For each of the integral types:
struct atomic_itype;
bool atomic_is_lock_free(const volatile atomic_itype*);
bool atomic_is_lock_free(const atomic_itype*);
void atomic_store(volatile atomic_itype*, integral);
void atomic_store(atomic_itype*, integral);
void atomic_store_explicit(volatile atomic_itype*, integral, memory_order);
void atomic_store_explicit(atomic_itype*, integral, memory_order);
integral atomic_load(const volatile atomic_itype*);
integral atomic_load(const atomic_itype*);
integral atomic_load_explicit(const volatile atomic_itype*, memory_order);
integral atomic_load_explicit(const atomic_itype*, memory_order);
integral atomic_exchange(volatile atomic_itype*, integral);
integral atomic_exchange(atomic_itype*, integral);
integral atomic_exchange_explicit(volatile atomic_itype*, integral, memory_order);
integral atomic_exchange_explicit(atomic_itype*, integral, memory_order);
bool atomic_compare_exchange_weak(volatile atomic_itype*, integral*, integral);
bool atomic_compare_exchange_weak(atomic_itype*, integral*, integral);
bool atomic_compare_exchange_strong(atomic_itype*, integral*, integral);
bool atomic_compare_exchange_strong(atomic_itype*, integral*, integral);
bool atomic_compare_exchange_weak_explicit(volatile atomic_itype*, integral*, integral*,
integral, memory_order, memory_order);
bool atomic_compare_exchange_weak_explicit(atomic_itype*, integral*, integral*,
integral, memory_order, memory_order);
bool atomic_compare_exchange_strong_explicit(volatile atomic_itype*, integral*, integral*,
integral, memory_order, memory_order);
bool atomic_compare_exchange_strong_explicit(atomic_itype*, integral*, integral*,
integral, memory_order, memory_order);
bool atomic_compare_exchange_strong_explicit(atomic_itype*, integral*,
    integral, memory_order, memory_order);
integral atomic_fetch_add(atomic_itype*, integral);
integral atomic_fetch_add_explicit(volatile atomic_itype*, integral,
    memory_order);
integral atomic_fetch_add_explicit(atomic_itype*, integral,
    memory_order);
integral atomic_fetch_sub(volatile atomic_itype*, integral);
integral atomic_fetch_sub(atomic_itype*, integral);
integral atomic_fetch_sub_explicit(volatile atomic_itype*, integral,
    memory_order);
integral atomic_fetch_sub_explicit(atomic_itype*, integral,
    memory_order);
integral atomic_fetch_and(volatile atomic_itype*, integral);
integral atomic_fetch_and(atomic_itype*, integral);
integral atomic_fetch_and_explicit(volatile atomic_itype*, integral,
    memory_order);
integral atomic_fetch_and_explicit(atomic_itype*, integral,
    memory_order);
integral atomic_fetch_or(volatile atomic_itype*, integral);
integral atomic_fetch_or(atomic_itype*, integral);
integral atomic_fetch_or_explicit(volatile atomic_itype*, integral,
    memory_order);
integral atomic_fetch_or_explicit(atomic_itype*, integral,
    memory_order);
integral atomic_fetch_xor(volatile atomic_itype*, integral);
integral atomic_fetch_xor(atomic_itype*, integral);
integral atomic_fetch_xor_explicit(volatile atomic_itype*, integral,
    memory_order);
integral atomic_fetch_xor_explicit(atomic_itype*, integral,
    memory_order);

// 29.5.2, address types
struct atomic_address;
bool atomic_is_lock_free(const volatile atomic_address*);
bool atomic_is_lock_free(const atomic_address*);
void atomic_store(volatile atomic_address*, void*);
void atomic_store(atomic_address*, void*);
void atomic_store_explicit(volatile atomic_address*, void*, memory_order);
void atomic_store_explicit(atomic_address*, void*, memory_order);
void* atomic_load(const volatile atomic_address*);
void* atomic_load(const atomic_address*);
void* atomic_load_explicit(const volatile atomic_address*, memory_order);
void* atomic_load_explicit(const atomic_address*, memory_order);
void* atomic_exchange(volatile atomic_address*, void*);
void* atomic_exchange(const atomic_address*, void*);
void* atomic_exchange_explicit(volatile atomic_address*, void*, memory_order);
void* atomic_exchange_explicit(const atomic_address*, void*, memory_order);
bool atomic_compare_exchange_weak(volatile atomic_address*, void**, void*);
bool atomic_compare_exchange_weak(atomic_address*, void**, void*);
bool atomic_compare_exchange_strong(volatile atomic_address*, void**, void*);
bool atomic_compare_exchange_strong(atomic_address*, void**, void*);
bool atomic_compare_exchange_weak_explicit(volatile atomic_address*, void**, void*,
    memory_order, memory_order);
bool atomic_compare_exchange_weak_explicit(atomic_address*, void**, void*,
    memory_order, memory_order);
memory_order, memory_order);
bool atomic_compare_exchange_strong_explicit(volatile atomic_address*, void**, void*,
memory_order, memory_order);
bool atomic_compare_exchange_strong_explicit(atomic_address*, void**, void*,
memory_order, memory_order);
void* atomic_fetch_add(volatile atomic_address*, ptrdiff_t);
void* atomic_fetch_add(atomic_address*, ptrdiff_t);
void* atomic_fetch_add_explicit(volatile atomic_address*, ptrdiff_t,
memory_order);
void* atomic_fetch_add_explicit(atomic_address*, ptrdiff_t,
memory_order);
void* atomic_fetch_sub(volatile atomic_address*, ptrdiff_t);
void* atomic_fetch_sub(atomic_address*, ptrdiff_t);
void* atomic_fetch_sub_explicit(volatile atomic_address*, ptrdiff_t,
memory_order);
void* atomic_fetch_sub_explicit(atomic_address*, ptrdiff_t,
memory_order);

// 29.5.3, generic types
template<class T> struct atomic;
template<class T> struct atomic<T*>;
template<> struct atomic<integral>;

// 29.8, fences
void atomic_thread_fence(memory_order);
void atomic_signal_fence(memory_order);

29.3 Order and Consistency

namespace std {

typedef enum memory_order {
    memory_order_relaxed, memory_order_consume, memory_order_acquire,
    memory_order_release, memory_order_acq_rel, memory_order_seq_cst
} memory_order;
}

The enumeration memory_order specifies the detailed regular (non-atomic) memory synchronization order as defined in 1.10 and may provide for operation ordering. Its enumerated values and their meanings are as follows:

— memory_order_relaxed: no operation orders memory.
— memory_order_release, memory_order_acq_rel, and memory_order_seq_cst: a store operation performs a release operation on the affected memory location.
— memory_order_consume: a load operation performs a consume operation on the affected memory location.
— memory_order_acquire, memory_order_acq_rel, and memory_order_seq_cst: a load operation performs an acquire operation on the affected memory location.

[Note: Atomic operations specifying memory_order_relaxed are relaxed with respect to memory ordering. Implementations must still guarantee that any given atomic access to a particular atomic object be indivisible with respect to all other atomic accesses to that object. — end note]
There shall be a single total order $S$ on all \texttt{memory\_order\_seq\_cst} operations, consistent with the “happens before” order and modification orders for all affected locations, such that each \texttt{memory\_order\_seq\_cst} operation that loads a value observes either the last preceding modification according to this order $S$, or the result of an operation that is not \texttt{memory\_order\_seq\_cst}. [\textit{Note: Although it is not explicitly required that $S$ include locks, it can always be extended to an order that does include lock and unlock operations, since the ordering between those is already included in the “happens before” ordering. — end note}]

For an atomic operation $B$ that reads the value of an atomic object $M$, if there is a \texttt{memory\_order\_seq\_cst} fence $X$ sequenced before $B$, then $B$ observes either the last \texttt{memory\_order\_seq\_cst} modification of $M$ preceding $X$ in the total order $S$ or a later modification of $M$ in its modification order.

For atomic operations $A$ and $B$ on an atomic object $M$, where $A$ modifies $M$ and $B$ takes its value, if there is a \texttt{memory\_order\_seq\_cst} fence $X$ such that $A$ is sequenced before $X$ and $B$ follows $X$ in $S$, then $B$ observes either the effects of $A$ or a later modification of $M$ in its modification order.

For atomic operations $A$ and $B$ on an atomic object $M$, where $A$ modifies $M$ and $B$ takes its value, if there are \texttt{memory\_order\_seq\_cst} fences $X$ and $Y$ such that $A$ is sequenced before $X$, $Y$ is sequenced before $B$, and $X$ precedes $Y$ in $S$, then $B$ observes either the effects of $A$ or a later modification of $M$ in its modification order.

For atomic operations $A$ and $B$ on an atomic object $M$, if there are \texttt{memory\_order\_seq\_cst} fences $X$ and $Y$ such that $A$ is sequenced before $X$, $Y$ is sequenced before $B$, and $X$ precedes $Y$ in $S$, then $B$ observes either the effects of $A$ or a later modification of $M$ in its modification order.

\[ \text{Note:} \ \texttt{memory\_order\_seq\_cst} \text{ ensures sequential consistency only for a program that is free of data races and uses exclusively} \ \texttt{memory\_order\_seq\_cst} \text{ operations. Any use of weaker ordering will invalidate this guarantee unless extreme care is used. In particular,} \ \texttt{memory\_order\_seq\_cst} \text{ fences ensure a total order only for the fences themselves. Fences cannot, in general, be used to restore sequential consistency for atomic operations with weaker ordering specifications. — end note} \]

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.\textsuperscript{342} 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 $B$ does not happen before $A$, and
- if an evaluation $A$ is included in the sequence, then every evaluation that assigns to the same variable and happens before $A$ is included.

\[ \text{Note:} \ \text{The second requirement disallows “out-of-thin-air” or “speculative” stores of atomics when relaxed atomics are used. Since unordered operations are involved, evaluations may appear in this sequence out of thread order. For example, with $x$ and $y$ initially zero,} \]

```c
// 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:

\[ \text{Among other implications, atomic variables shall not decay.} \]
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]

Atomic read-modify-write operations shall always read the last value (in the modification order) written before the write associated with the read-modify-write operation.

Implementations should make atomic stores visible to atomic loads within a reasonable amount of time.

**template** <typename T>
    T kill_dependency(T y);

    **Effects:** The argument does not carry a dependency to the return value (1.10).

    **Returns:** y.

### 29.4 Lock-free Property

[atomics.lockfree]

- \#define ATOMIC_CHAR_LOCK_FREE implementation-defined
- \#define ATOMIC_CHAR16_T_LOCK_FREE implementation-defined
- \#define ATOMIC_CHAR32_T_LOCK_FREE implementation-defined
- \#define ATOMIC_WCHAR_T_LOCK_FREE implementation-defined
- \#define ATOMIC_SHORT_LOCK_FREE implementation-defined
- \#define ATOMIC_INT_LOCK_FREE implementation-defined
- \#define ATOMIC_LONG_LOCK_FREE implementation-defined
- \#define ATOMIC_LLONG_LOCK_FREE implementation-defined
- \#define ATOMIC_ADDRESS_LOCK_FREE implementation-defined
The Atomic macros indicate the lock-free property of the corresponding atomic types, with the signed and unsigned variants grouped together. The properties also apply to the corresponding 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 (29.6) indicates whether the object is lock-free. In any given program execution, the result of the lock-free query shall be consistent for all pointers of the same type.

[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]

29.5 Atomic Types

29.5.1 Integral Types

namespace std {
  
  typedef struct atomic_bool {
    bool is_lock_free() const volatile;
    bool is_lock_free() const;
    void store(bool, memory_order = memory_order_seq_cst) volatile;
    void store(bool, memory_order = memory_order_seq_cst);
    bool load(memory_order = memory_order_seq_cst) const volatile;
    bool load(memory_order = memory_order_seq_cst) const;
    operator bool() const volatile;
    operator bool() const;
    bool exchange(bool, memory_order = memory_order_seq_cst) volatile;
    bool exchange(bool, memory_order = memory_order_seq_cst);
    bool compare_exchange_weak(bool&, bool, memory_order, memory_order) volatile;
    bool compare_exchange_weak(bool&, bool, memory_order, memory_order);
    bool compare_exchange_strong(bool&, bool, memory_order, memory_order) volatile;
    bool compare_exchange_strong(bool&, bool, memory_order, memory_order);
    bool compare_exchange_weak(bool&, bool, memory_order = memory_order_seq_cst) volatile;
    bool compare_exchange_weak(bool&, bool, memory_order = memory_order_seq_cst);
    bool operator=(bool) volatile;
  } atomic_bool;

  bool atomic_is_lock_free(const volatile atomic_bool*);
  bool atomic_is_lock_free(const atomic_bool*);
  void atomic_init(volatile atomic_bool*, bool);
  void atomic_init(atomic_bool*, bool);
  void atomic_store(volatile atomic_bool*, bool);
  void atomic_store(atomic_bool*, bool);
  void atomic_store_explicit(volatile atomic_bool*, bool, memory_order);
  void atomic_store_explicit(atomic_bool*, bool, memory_order);
  bool atomic_load(const volatile atomic_bool*);
  bool atomic_load(const atomic_bool*);

§ 29.5.1 1113
bool atomic_load_explicit(const volatile atomic_bool*, memory_order);
bool atomic_load_explicit(const atomic_bool*, memory_order);
bool atomic_exchange(volatile atomic_bool*, bool);
bool atomic_exchange(atomic_bool*, bool);
bool atomic_exchange_explicit(volatile atomic_bool*, bool, memory_order);
bool atomic_exchange_explicit(atomic_bool*, bool, memory_order);
bool atomic_compare_exchange_weak(volatile atomic_bool*, bool*, bool);
bool atomic_compare_exchange_weak(atomic_bool*, bool*, bool);
bool atomic_compare_exchange_strong(volatile atomic_bool*, bool*, bool);
bool atomic_compare_exchange_strong(atomic_bool*, bool*, bool);
bool atomic_compare_exchange_weak_explicit(volatile atomic_bool*, bool*, bool,
memory_order, memory_order);
bool atomic_compare_exchange_weak_explicit(atomic_bool*, bool*, bool,
memory_order, memory_order);
bool atomic_compare_exchange_strong_explicit(volatile atomic_bool*, bool*, bool,
memory_order, memory_order);
bool atomic_compare_exchange_strong_explicit(atomic_bool*, bool*, bool,
memory_order, memory_order);

// For each of the integral types listed below:
typedef struct atomic_iotype {
    bool is_lock_free() const volatile;
    bool is_lock_free() const;
    void store(integral, memory_order = memory_order_seq_cst) volatile;
    void store(integral, memory_order = memory_order_seq_cst);
    integral load(memory_order = memory_order_seq_cst) const volatile;
    integral load(memory_order = memory_order_seq_cst) const;
    operator integral() const volatile;
    operator integral() const;
    integral exchange(integral,
        memory_order = memory_order_seq_cst) volatile;
    integral exchange(integral,
        memory_order = memory_order_seq_cst);
    bool compare_exchange_weak(integral&, integral,
        memory_order, memory_order) volatile;
    bool compare_exchange_weak(integral&, integral,
        memory_order, memory_order);
    bool compare_exchange_strong(integral&, integral,
        memory_order, memory_order) volatile;
    bool compare_exchange_strong(integral&, integral,
        memory_order, memory_order);
    bool compare_exchange_weak(integral&, integral,
        memory_order = memory_order_seq_cst) volatile;
    bool compare_exchange_weak(integral&, integral,
        memory_order = memory_order_seq_cst);
    bool compare_exchange_strong(integral&, integral,
        memory_order = memory_order_seq_cst) volatile;
    bool compare_exchange_strong(integral&, integral,
        memory_order = memory_order_seq_cst);
    integral fetch_add(integral,
        memory_order = memory_order_seq_cst) volatile;
    integral fetch_add(integral,
        memory_order = memory_order_seq_cst);
    integral fetch_sub(integral,
        memory_order = memory_order_seq_cst) volatile;
integral fetch_sub(integral,
    memory_order = memory_order_seq_cst);
integral fetch_and(integral,
    memory_order = memory_order_seq_cst) volatile;
integral fetch_and(integral,
    memory_order = memory_order_seq_cst);
integral fetch_or(integral,
    memory_order = memory_order_seq_cst) volatile;
integral fetch_or(integral,
    memory_order = memory_order_seq_cst);
integral fetch_xor(integral,
    memory_order = memory_order_seq_cst);
integral fetch_xor(integral,
    memory_order = memory_order_seq_cst);

atomic_itype() = default;
constexpr atomic_itype(integral);
atomic_itype(const atomic_itype&) = delete;
atomic_itype& operator=(const atomic_itype&) = delete;
atomic_itype& operator=(const atomic_itype& volatile) = delete;
integral operator=(integral) volatile;
integral operator=(integral);
integral operator++(int) volatile;
integral operator++(int);
integral operator--(int) volatile;
integral operator--(int);
integral operator++() volatile;
integral operator++();
integral operator--() volatile;
integral operator--();
integral operator++(integral) volatile;
integral operator++(integral);
integral operator--(integral) volatile;
integral operator--(integral);
integral operator&=(integral) volatile;
integral operator&=(integral);
integral operator|=(integral) volatile;
integral operator|=(integral);
integral operator^=(integral) volatile;
integral operator^=(integral);

bool atomic_is_lock_free(const volatile atomic_itype*);
bool atomic_is_lock_free(const atomic_itype*);
void atomic_init(volatile atomic_itype*, integral);
void atomic_init(atomic_itype*, integral);
void atomic_store(volatile atomic_itype*, integral);
void atomic_store(atomic_itype*, integral);
void atomic_store_explicit(volatile atomic_itype*, integral,
    memory_order);
void atomic_store_explicit(atomic_itype*, integral,
    memory_order);
integral atomic_load(const volatile atomic_itype*);
integral atomic_load(const atomic_itype*);
integral atomic_load_explicit(const volatile atomic_itype*, memory_order);
integral atomic_load_explicit(const atomic_type*, memory_order);
integral atomic_exchange(volatile atomic_type*, integral);
integral atomic_exchange(atomic_type*, integral);
integral atomic_exchange_explicit(volatile atomic_type*, integral, memory_order);
integral atomic_exchange_explicit(atomic_type*, integral, memory_order);
bool atomic_compare_exchange_weak(volatile atomic_type*, integral*, integral);
bool atomic_compare_exchange_weak(atomic_type*, integral*, integral);
bool atomic_compare_exchange_strong(volatile atomic_type*, integral*, integral);
bool atomic_compare_exchange_strong(atomic_type*, integral*, integral);
bool atomic_compare_exchange_weak_explicit(volatile atomic_type*, integral*, integral, memory_order, memory_order);
bool atomic_compare_exchange_weak_explicit(atomic_type*, integral*, integral, memory_order, memory_order);
bool atomic_compare_exchange_strong_explicit(volatile atomic_type*, integral*, integral, memory_order, memory_order);
bool atomic_compare_exchange_strong_explicit(atomic_type*, integral*, integral, memory_order, memory_order);
integral atomic_fetch_add(volatile atomic_type*, integral);
integral atomic_fetch_add(atomic_type*, integral);
integral atomic_fetch_add_explicit(volatile atomic_type*, integral, memory_order);
integral atomic_fetch_add_explicit(atomic_type*, integral, memory_order);
integral atomic_fetch_sub(volatile atomic_type*, integral);
integral atomic_fetch_sub(atomic_type*, integral);
integral atomic_fetch_sub_explicit(volatile atomic_type*, integral, memory_order);
integral atomic_fetch_sub_explicit(atomic_type*, integral, memory_order);
integral atomic_fetch_and(volatile atomic_type*, integral);
integral atomic_fetch_and(atomic_type*, integral);
integral atomic_fetch_and_explicit(volatile atomic_type*, integral, memory_order);
integral atomic_fetch_and_explicit(atomic_type*, integral, memory_order);
integral atomic_fetch_or(volatile atomic_type*, integral);
integral atomic_fetch_or(atomic_type*, integral);
integral atomic_fetch_or_explicit(volatile atomic_type*, integral, memory_order);
integral atomic_fetch_or_explicit(atomic_type*, integral, memory_order);
integral atomic_fetch_xor(volatile atomic_type*, integral);
integral atomic_fetch_xor(atomic_type*, integral);
integral atomic_fetch_xor_explicit(volatile atomic_type*, integral, memory_order);
integral atomic_fetch_xor_explicit(atomic_type*, integral, memory_order);
}

1 The name atomic_type and the functions operating on it in the preceding synopsis are placeholders for a set of classes and functions. Throughout the preceding synopsis, atomic_type should be replaced by each of the class names in table 142 and integral should be replaced by the integral type corresponding to the

§ 29.5.1 1116
Table 142 — Atomics for built-in types

<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>

class name. Table 143 shows typedefs to atomic integral classes and the corresponding `<cstdint>` typedefs.

The atomic integral types shall have standard layout. They shall each have a trivial default constructor, a constexpr 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 §29.6.

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 effort required to port existing code. — end note]

### 29.5.2 Address Type

```cpp
namespace std {
    typedef struct atomic_address {
        bool is_lock_free() const volatile;
        bool is_lock_free() const;
        void store(void*, memory_order = memory_order_seq_cst) volatile;
        void store(void*, memory_order = memory_order_seq_cst);
        void* load(memory_order = memory_order_seq_cst) const volatile;
        void* load(memory_order = memory_order_seq_cst) const;
        operator void*() const volatile;
        operator void*() const;
        void* exchange(void*, memory_order = memory_order_seq_cst) volatile;
        void* exchange(void*, memory_order = memory_order_seq_cst);
        bool compare_exchange_weak(void*, void*,
                                      memory_order, memory_order) volatile;
        bool compare_exchange_weak(void*, void*,
                                      memory_order, memory_order);
        bool compare_exchange_strong(void*, void*,
                                       memory_order, memory_order) volatile;
        bool compare_exchange_strong(void*, void*,
                                       memory_order, memory_order);
    };
}
```

§ 29.5.2
Table 143 — Atomics for standard typedef types

<table>
<thead>
<tr>
<th>atomic typedef name</th>
<th>&lt;cstdint&gt; typedef name</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>atomic_intptr_t</td>
<td>intptr_t</td>
</tr>
<tr>
<td>atomic_uintptr_t</td>
<td>uintptr_t</td>
</tr>
<tr>
<td>atomic_size_t</td>
<td>size_t</td>
</tr>
<tr>
<td>atomic_ptrdiff_t</td>
<td>ptrdiff_t</td>
</tr>
<tr>
<td>atomic_intmax_t</td>
<td>intmax_t</td>
</tr>
<tr>
<td>atomic_uintmax_t</td>
<td>uintmax_t</td>
</tr>
</tbody>
</table>

bool compare_exchange_weak(void*&, void*,
    memory_order = memory_order_seq_cst) volatile;
bool compare_exchange_weak(void*&, void*,
    memory_order = memory_order_seq_cst);
bool compare_exchange_strong(void*&, void*,
    memory_order = memory_order_seq_cst);
bool compare_exchange_strong(void*&, void*,
    memory_order = memory_order_seq_cst) volatile;
bool compare_exchange_strong(const void*&, const void*,
    memory_order, memory_order) volatile;
bool compare_exchange_strong(const void*&, const void*,
    memory_order, memory_order);
bool compare_exchange_strong(const void*&, const void*,
    memory_order, memory_order) volatile;
bool compare_exchange_strong(const void*&, const void*,
    memory_order, memory_order);
bool compare_exchange_weak(const void*&, const void*,
    memory_order = memory_order_seq_cst) volatile;
bool compare_exchange_weak(const void*&, const void*,
    memory_order = memory_order_seq_cst);
bool compare_exchange_strong(const void*&, const void*,
    memory_order = memory_order_seq_cst) volatile;
bool compare_exchange_strong(const void*&, const void*,
    memory_order = memory_order_seq_cst);
void* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst) volatile;
void* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst);
atomic_address() = default;
constexpr atomic_address(void*) = default;
atomic_address(const atomic_address&) = delete;
atomic_address& operator=(const atomic_address&) = delete;
atomic_address& operator=(const atomic_address&) volatile = delete;
void* operator=(const void*) volatile;
void* operator=(const void*);
void* operator+=(ptrdiff_t) volatile;
void* operator+=(ptrdiff_t);
void* operator-=(ptrdiff_t) volatile;
void* operator-=(ptrdiff_t);
} atomic_address;

bool atomic_is_lock_free(const volatile atomic_address*);
bool atomic_is_lock_free(const atomic_address*);
void atomic_init(volatile atomic_address*, void*);
void atomic_init(atomic_address*, void*);
void atomic_store(volatile atomic_address*, void*);
void atomic_store(atomic_address*, void*);
void atomic_store_explicit(volatile atomic_address*, void*, memory_order);
void atomic_store_explicit(atomic_address*, void*, memory_order);
void* atomic_load(const volatile atomic_address*);
void* atomic_load(const atomic_address*);
void* atomic_load_explicit(const volatile atomic_address*, memory_order);
void* atomic_load_explicit(const atomic_address*, memory_order);
void* atomic_exchange(volatile atomic_address*, void*);
void* atomic_exchange(atomic_address*, void*);
void* atomic_exchange_explicit(volatile atomic_address*, void*, memory_order, memory_order);
void* atomic_exchange_explicit(atomic_address*, void*, memory_order, memory_order);
bool atomic_compare_exchange_weak(volatile atomic_address*, void**, void*);
bool atomic_compare_exchange_weak(atomic_address*, void**, void*);
bool atomic_compare_exchange_strong(volatile atomic_address*, void**, void*);
bool atomic_compare_exchange_strong(atomic_address*, void**, void*);
bool atomic_compare_exchange_weak_explicit(volatile atomic_address*, void**, void*, memory_order, memory_order);
bool atomic_compare_exchange_weak_explicit(atomic_address*, void**, void*, memory_order, memory_order);
bool atomic_compare_exchange_strong_explicit(volatile atomic_address*, void**, void*, memory_order, memory_order);
bool atomic_compare_exchange_strong_explicit(atomic_address*, void**, void*, memory_order, memory_order);
void* atomic_fetch_add(volatile atomic_address*, ptrdiff_t);
void* atomic_fetch_add(atomic_address*, ptrdiff_t);
void* atomic_fetch_add_explicit(volatile atomic_address*, ptrdiff_t, memory_order);
void* atomic_fetch_add_explicit(atomic_address*, ptrdiff_t, memory_order);
void* atomic_fetch_sub(volatile atomic_address*, ptrdiff_t);
void* atomic_fetch_sub(atomic_address*, ptrdiff_t);
void* atomic_fetch_sub_explicit(volatile atomic_address*, ptrdiff_t, memory_order);
void* atomic_fetch_sub_explicit(atomic_address*, ptrdiff_t, memory_order);
The type `atomic_address` shall have standard layout. It shall have a trivial default constructor, a constexpr value constructor, a deleted copy constructor, a deleted copy assignment operator, and a trivial destructor. It shall support aggregate initialization syntax.

The semantics of the operations on this type are defined in 29.6.

The `atomic_address` type provides atomic void* operations. The unit of addition/subtraction shall be one byte.

[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]

### 29.5.3 Generic Types

```cpp
namespace std {
    template <class T> struct atomic {
        bool is_lock_free() const volatile;
        bool is_lock_free() const;
        void store(T, memory_order = memory_order_seq_cst) volatile;
        void store(T, memory_order = memory_order_seq_cst);
        T load(memory_order = memory_order_seq_cst) const volatile;
        T load(memory_order = memory_order_seq_cst) const;
        operator T() const volatile;
        operator T() const;
        T exchange(T, memory_order = memory_order_seq_cst) volatile;
        T exchange(T, memory_order = memory_order_seq_cst);
        bool compare_exchange_weak(T&, T, memory_order, memory_order) volatile;
        bool compare_exchange_weak(T&, T, memory_order, memory_order);
        bool compare_exchange_strong(T&, T, memory_order, memory_order) volatile;
        bool compare_exchange_strong(T&, T, memory_order, memory_order);
        bool compare_exchange_weak(T&, T, memory_order = memory_order_seq_cst) volatile;
        bool compare_exchange_weak(T&, T, memory_order = memory_order_seq_cst);
        bool compare_exchange_strong(T&, T, memory_order = memory_order_seq_cst) volatile;
        bool compare_exchange_strong(T&, T, memory_order = memory_order_seq_cst);
    }
    template <> struct atomic<integral> : atomic_<int_type> {
        atomic() = default;
        constexpr atomic(integral);
        atomic(const atomic&) = delete;
        atomic_k operator=(const atomic&) = delete;
        atomic_k operator=(const atomic&) volatile = delete;
        integral operator=(integral) volatile;
        integral operator=(integral);
    }
}
```

§ 29.5.3 1120
There is a generic class template `atomic<T>`. The type of the template argument `T` shall be trivially copyable (3.9). [Note: Type arguments that are not also statically initializable may be difficult to use. — end note]

Specializations of the `atomic` template shall have a deleted copy constructor, a deleted copy assignment operator, and a `constexpr` value constructor.

§ 29.5.3
There are full specializations over the integral types on the `atomic` class template. For each integral type `integral` in the second column of table 142 or table 143, the specialization `atomic<integral>` shall be publicly derived from the corresponding atomic integral type in the first column of the table. In addition, the specialization `atomic<bool>` shall be publicly derived from `atomic_bool`. 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.

### 29.6 Operations on Atomic Types

There are only a few kinds of operations on atomic types, though there are many instances on those kinds. This section specifies each general kind. The specific instances are defined in 29.5.1, 29.5.2, and 29.5.3.

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
constexpr A::A(C desired);
```

**Effects:** Initializes the object with the value `desired`. **[Note:** Construction is not atomic. — end note]

```cpp
#define ATOMIC_VAR_INIT(value) see below
```

**Remarks:** A macro that expands to a token sequence suitable for initializing an atomic variable of a type that is initialization-compatible with `value`. Concurrent access to the variable being initialized, even via an atomic operation, constitutes a data race. **[Example:**

```cpp
atomic_int v = ATOMIC_VAR_INIT(5);
```

— end example]

```cpp
bool atomic_is_lock_free(const volatile A *object);
bool atomic_is_lock_free(const A *object);
A::is_lock_free() const volatile;
A::is_lock_free() const;
```

**Returns:** True if the object’s operations are lock-free, false otherwise.

```cpp
void atomic_init(volatile A *object, C desired);
void atomic_init(A *object, C desired);
```
Effects: Non-atomically assigns the value \textit{desired} to \texttt{\*object}. Concurrent access from another thread, even via an atomic operation, constitutes a data race.

\begin{verbatim}
void atomic_store(volatile A* object, C desired);
void atomic_store(A* object, C desired);
void atomic_store_explicit(volatile A* object, C desired, memory_order order);
void atomic_store_explicit(A* object, C desired, memory_order order);
void A::store(C desired, memory_order order = memory_order_seq_cst) volatile;
void A::store(C desired, memory_order order = memory_order_seq_cst);
\end{verbatim}

Requires: The \texttt{order} argument shall not be \texttt{memory_order_consume}, \texttt{memory_order_acquire}, or \texttt{memory_order_acq_rel}.

Effects: Atomically replaces the value pointed to by \texttt{object} or by \texttt{this} with the value of \textit{desired}. Memory is affected according to the value of \texttt{order}.

\begin{verbatim}
C A::operator=(C desired) volatile;
C A::operator=(C desired);
Effects: \texttt{store(desired)}
Returns: \textit{desired}
\end{verbatim}

\begin{verbatim}
C atomic_load(const volatile A* object);
C atomic_load(const A* object);
C atomic_load_explicit(const volatile A* object, memory_order);
C atomic_load_explicit(const A* object, memory_order);
C A::load(memory_order order = memory_order_seq_cst) const volatile;
C A::load(memory_order order = memory_order_seq_cst) const;
\end{verbatim}

Requires: The \texttt{order} argument shall not be \texttt{memory_order_release} nor \texttt{memory_order_acq_rel}.

Effects: Memory is affected according to the value of \texttt{order}.

Returns: Atomically returns the value pointed to by \texttt{object} or by \texttt{this}.

\begin{verbatim}
A::operator C() const volatile;
A::operator C() const;
Effects: \texttt{load()}
Returns: the result of \texttt{load()}
\end{verbatim}

\begin{verbatim}
C atomic_exchange(volatile A* object, C desired);
C atomic_exchange(A* object, C desired);
C atomic_exchange_explicit(volatile A* object, C desired, memory_order);
C atomic_exchange_explicit(A* object, C desired, memory_order);
C A::exchange(C desired, memory_order order = memory_order_seq_cst) volatile;
C A::exchange(C desired, memory_order order = memory_order_seq_cst);
\end{verbatim}

Effects: Atomically replaces the value pointed to by \texttt{object} or by \texttt{this} with \textit{desired}. Memory is affected according to the value of \texttt{order}. These operations are atomic read-modify-write operations (1.10).

Returns: Atomically returns the value pointed to by \texttt{object} or by \texttt{this} immediately before the effects.

\begin{verbatim}
bool atomic_compare_exchange_weak(volatile A* object, C* expected, C desired);
bool atomic_compare_exchange_weak(A* object, C* expected, C desired);
\end{verbatim}
bool atomic_compare_exchange_strong(volatile A* object, C* expected, C desired);
bool atomic_compare_exchange_strong(A* object, C* expected, C desired);
bool atomic_compare_exchange_weak_explicit(volatile A* object, C* expected, C desired,
  memory_order success, memory_order failure);
bool atomic_compare_exchange_weak_explicit(A* object, C* expected, C desired,
  memory_order success, memory_order failure);
bool atomic_compare_exchange_strong_explicit(volatile A* object, C* expected, C desired,
  memory_order success, memory_order failure);
bool atomic_compare_exchange_strong_explicit(A* object, C* expected, C desired,
  memory_order success, memory_order failure);
bool A::compare_exchange_weak(C& expected, C desired,
  memory_order order = memory_order_seq_cst) volatile;
bool A::compare_exchange_weak(C& expected, C desired,
  memory_order order = memory_order_seq_cst);
bool A::compare_exchange_strong(C& expected, C desired,
  memory_order order = memory_order_seq_cst) volatile;
bool A::compare_exchange_strong(C& expected, C desired,
  memory_order order = memory_order_seq_cst);

19 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.

20 Effects: Atomically, compares the contents of the memory pointed to by object or by this for equality with that in expected, and if true, replaces the contents of the memory pointed to by object or by this with that in desired, and if false, updates the contents of the memory in expected with the contents of the memory pointed to by object or by this. Further, if the comparison is true, memory is affected according to the value of success, and if the comparison is false, memory is affected according to the value of failure. When only one memory_order argument is supplied, the value of success is order, and the value of failure is order except that a value of memory_order_acq_rel shall be replaced by the value memory_order_acquire and a value of memory_order_release shall be replaced by the value memory_order_relaxed. If the operation returns true, these operations are atomic read-modify-write operations (1.10). Otherwise, these operations are atomic load operations.

21 Returns: The result of the comparison.

22 [ Note: The effect of the compare-and-exchange operations is
  if (memcmp(object, expected, sizeof(*object)) == 0)
    memcpy(object, &desired, sizeof(*object));
  else
    memcpy(expected, object, sizeof(*object));
  — end note] [ Example: the expected use of the compare-and-exchange operations is as follows. The compare-and-exchange operations will update expected when another iteration of the loop is needed.

    expected = current.load();
    do {

§ 29.6
desired = function(expected);
} while (!current.compare_exchange_weak(expected, desired));

— end example

Remark: The weak compare-and-exchange operations may fail spuriously, that is, return false while leaving the contents of memory pointed to by expected before the operation is the same that same as that of the object and the same as that of expected after the operation. [Note: This spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g., load-locked store-conditional machines. A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will be in a loop.

When a compare-and-exchange is in a loop, the weak version will yield better performance on some platforms. When a weak compare-and-exchange would require a loop and a strong one would not, the strong one is preferable. — end note]

[Note: The memcpy and memcmp semantics of the compare-and-exchange operations may result in failed comparisons for values that compare equal with operator== if the underlying type has padding bits, trap bits, or alternate representations of the same value. Thus, compare_exchange_strong should be used with extreme care. On the other hand, compare_exchange_weak should converge rapidly. — end note]

The following operations perform arithmetic computations. The key, operator, and computation correspondence is:

<table>
<thead>
<tr>
<th>Key</th>
<th>Op</th>
<th>Computation</th>
<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 atomic_fetch_key(volatile A *object, M operand);
C atomic_fetch_key(A* object, M operand);
C atomic_fetch_key_explicit(volatile A *object, M operand, memory_order order);
C atomic_fetch_key_explicit(A* object, M operand, memory_order order);
C A::fetch_key(M operand, memory_order order = memory_order_seq_cst) volatile;
C A::fetch_key(M operand, memory_order order = memory_order_seq_cst);

Effects: Atomically replaces the value pointed to by object or by this with the result of the computation applied to the value pointed to by object or by this and the given operand. Memory is affected according to the value of order. These operations are atomic read-modify-write operations (1.10).

Returns: Atomically, the value pointed to by object or by this immediately before the effects.

Remark: For signed 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::operator op=(M operand) volatile;
C A::operator op=(M operand);

Effects: fetch_key(operand)

Returns: fetch_key(operand) op operand

§ 29.6
namespace std {
    typedef struct atomic_flag {
        bool test_and_set(memory_order = memory_order_seq_cst) volatile;
        bool test_and_set(memory_order = memory_order_seq_cst);
        void clear(memory_order = memory_order_seq_cst) volatile;
        void clear(memory_order = memory_order_seq_cst);
        
        atomic_flag() = default;
        atomic_flag(const atomic_flag&) = delete;
        atomic_flag& operator=(const atomic_flag&) = delete;
        atomic_flag& operator=(const atomic_flag&) volatile = delete;
    } atomic_flag;

    bool atomic_flag_test_and_set(volatile atomic_flag*);
    bool atomic_flag_test_and_set(atomic_flag*);
    bool atomic_flag_test_and_set_explicit(volatile atomic_flag*, memory_order);
    bool atomic_flag_test_and_set_explicit(atomic_flag*, memory_order);
    void atomic_flag_clear(volatile atomic_flag*);
    void atomic_flag_clear(atomic_flag*);
    void atomic_flag_clear_explicit(volatile atomic_flag*, memory_order);
    void atomic_flag_clear_explicit(atomic_flag*, memory_order);

    #define ATOMIC_FLAG_INIT see below
}

1 The `atomic_flag` type provides the classic test-and-set functionality. It has two states, set and clear.

2 Operations on an object of type `atomic_flag` shall be lock-free. [Note: Hence the operations should also
be address-free. 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. It is unspecified whether an unitialized `atomic_flag` object has an initial state of set or clear. [Example:

```c
atomic_flag guard = ATOMIC_FLAG_INIT;
```
— end example]

```c
bool atomic_flag_test_and_set(volatile atomic_flag *object);
bool atomic_flag_test_and_set(atomic_flag *object);
bool atomic_flag_test_and_set_explicit(volatile atomic_flag *object, memory_order order);
bool atomic_flag_test_and_set_explicit(atomic_flag *object, memory_order order);
bool atomic_flag::test_and_set(memory_order order = memory_order_seq_cst) volatile;
bool atomic_flag::test_and_set(memory_order order = memory_order_seq_cst);
```

**Effects:** Atomically sets the value pointed to by `object` or by `this` to true. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (1.10).

```c
void atomic_flag_clear(volatile atomic_flag *object);
void atomic_flag_clear(atomic_flag *object);
void atomic_flag_clear_explicit(volatile atomic_flag *object, memory_order order);
void atomic_flag_clear_explicit(atomic_flag *object, memory_order order);
void atomic_flag::clear(memory_order order = memory_order_seq_cst) volatile;
void atomic_flag::clear(memory_order order = memory_order_seq_cst);
```

**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 false. Memory is affected according to the value of `order`.

### 29.8 Fences

This section introduces synchronization primitives called *fences*. Fences can have acquire semantics, release semantics, or both. A fence with acquire semantics is called an *acquire fence*. A fence with release semantics is called a *release fence*.

A release fence `A` synchronizes with an acquire fence `B` if there exist atomic operations `X` and `Y`, both operating on some atomic object `M`, such that `A` is sequenced before `X`, `X` modifies `M`, `Y` is sequenced before `B`, and `Y` reads the value written by `X` or a value written by any side effect in the hypothetical release sequence `X` would head if it were a release operation.

A release fence `A` synchronizes with an atomic operation `B` that performs an acquire operation on an atomic object `M` if there exists an atomic operation `X` such that `A` is sequenced before `X`, `X` modifies `M`, and `B` reads the value written by `X` or a value written by any side effect in the hypothetical release sequence `X` would head if it were a release operation.

An atomic operation `A` that is a release operation on an atomic object `M` synchronizes with an acquire fence `B` if there exists some atomic operation `X` on `M` such that `X` is sequenced before `B` and reads the value written by `A` or a value written by any side effect in the release sequence headed by `A`.

```c
void atomic_thread_fence(memory_order order);
```
Effects: depending on the value of order, this operation:

- has no effects, if \( \text{order} == \text{memory\_order\_relaxed} \);
- is an acquire fence, if \( \text{order} == \text{memory\_order\_acquire} \) \( \| \) \( \text{order} == \text{memory\_order\_consume} \);
- is a release fence, if \( \text{order} == \text{memory\_order\_release} \);
- is both an acquire fence and a release fence, if \( \text{order} == \text{memory\_order\_acq\_rel} \);
- is a sequentially consistent acquire and release fence, if \( \text{order} == \text{memory\_order\_seq\_cst} \).

\[
\text{void atomic\_signal\_fence(memory\_order \text{order});}
\]

Effects: equivalent to \text{atomic\_thread\_fence(order)}, except that synchronizes with relationships are established only between a thread and a signal handler executed in the same thread.

Note: \text{atomic\_signal\_fence} can be used to specify the order in which actions performed by the thread become visible to the signal handler.

Note: compiler optimizations and reorderings of loads and stores are inhibited in the same way as with \text{atomic\_thread\_fence}, but the hardware fence instructions that \text{atomic\_thread\_fence} would have inserted are not emitted.
30 Thread support library

30.1 General

The following subclauses describe components to create and manage threads (1.10), perform mutual exclusion, and communicate conditions between threads, as summarized in Table 145.

Table 145 — Thread support library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.3 Threads</td>
<td>&lt;thread&gt;</td>
</tr>
<tr>
<td>30.4 Mutual exclusion</td>
<td>&lt;mutex&gt;</td>
</tr>
<tr>
<td>30.5 Condition variables</td>
<td>&lt;condition_variable&gt;</td>
</tr>
<tr>
<td>30.6 Futures</td>
<td>&lt;future&gt;</td>
</tr>
</tbody>
</table>

30.2 Requirements

30.2.1 Template parameter names

Throughout this Clause, the names of template parameters are used to express type requirements.

If a parameter is **Predicate**, `operator()` applied to the actual template argument shall return a value that is convertible to `bool`.

30.2.2 Exceptions

Some functions described in this Clause are specified to throw exceptions of type `system_error` (19.5.6). Such exceptions shall be thrown if any of the function’s error conditions is detected or a call to an operating system or other underlying API results in an error that prevents the library function from meeting its specifications. Failure to allocate storage shall be reported as described in 17.6.4.11.

*Example:* Consider a function in this clause that is specified to throw exceptions of type `system_error` and specifies error conditions that include `operation_not_permitted` for a thread that does not have the privilege to perform the operation. Assume that, during the execution of this function, an `errno` of `EPERM` is reported by a POSIX API call used by the implementation. Since POSIX specifies an `errno` of `EPERM` when “the caller does not have the privilege to perform the operation”, the implementation maps `EPERM` to an error condition of `operation_not_permitted` (19.5) and an exception of type `system_error` is thrown.

The `error_code` reported by such an exception’s `code()` member function shall compare equal to one of the conditions specified in the function’s error condition element.

30.2.3 Native handles

Several classes described in this Clause have members `native_handle_type` and `native_handle`. The presence of these members and their semantics is implementation-defined. [Note: These members allow...]

§ 30.2.3
implementations to provide access to implementation details. Their names are specified to facilitate portable compile-time detection. Actual use of these members is inherently non-portable. — end note

30.2.4 Timing specifications

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 (20.10).

2 The member functions whose names end in _for take an argument that specifies a relative time. Implementations should use a monotonic clock to measure time for these functions. [Note: Implementations are not required to use a monotonic clock because such a clock may not be available. — end note]

3 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.

30.3 Threads

1 30.3 describes components that can be used to create and manage threads. [Note: These threads are intended to map one-to-one with operating system threads. — end note]

Header <thread> synopsis

```cpp
namespace std {
  #define __STDCPP_THREADS __cplusplus
  class thread;

  void swap(thread& x, thread& y);

  namespace this_thread {
    thread::id get_id();
    void yield();
    template <class Clock, class Duration>
      void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
    template <class Rep, class Period>
      void sleep_for(const chrono::duration<Rep, Period>& rel_time);
  }
}
```

30.3.1 Class thread

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]
```cpp
typedef implementation-defined native_handle_type; // See 30.2.3

// construct/copy/destroy:
thread();
template <class F, class ...Args> explicit thread(F&& f, Args&&... args);
t~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 30.2.3

// static members:
static unsigned hardware_concurrency();
};
```
thread::id shall be a trivially copyable class (Clause 9). The library may reuse the value of a thread::id of a terminated thread that can no longer be joined.

[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 25.4.

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
template <> struct hash<thread::id>;

21  Requires: the template specialization shall meet the requirements of class template hash (20.8.15).

30.3.1.2 thread constructors

thread();

1  Effects: Constructs a thread object that does not represent a thread of execution.
2  Postcondition: get_id() == id()
3  Throws: Nothing.

template <class F, class ...Args> explicit thread(F&& f, Args&&... args);

Given a function as follows:

```cpp
template <class T> typename decay<T>::type decay_copy(T&& v)
{ return std::forward<T>(v); }
```

4  Requires: F and each Ti in Args shall satisfy the MoveConstructible requirements. INVOKE(decay_copy(std::forward<F>(f)), decay_copy(std::forward<Args>(args))...) (20.8.2) shall be a valid expression.

5  Effects: Constructs an object of type thread. The new thread of execution executes INVOKE(decay_copy(std::forward<F>(f)), decay_copy(std::forward<Args>(args))...) with the calls to decay_copy being evaluated in the constructing thread. Any return value from this invocation is ignored. [Note: this implies that any exceptions not thrown from the invocation of the copy of f will be thrown in the constructing thread, not the new thread. — end note] If the invocation of INVOKE(decay_copy(std::forward<F>(f)), decay_copy(std::forward<Args>(args))...) terminates with an uncaught exception, std::terminate shall be called.

6  Synchronization: The invocation of the constructor happens before the invocation of the copy of f.
7  Postconditions: get_id() != id(). *this represents the newly started thread.
8  Throws: std::system_error if unable to start the new thread.
9  Error conditions:

   — resource_unavailable_try_again — the system lacked the necessary resources to create another thread, or the system-imposed limit on the number of threads in a process would be exceeded.

thread(thread&& x);

10  Effects: Constructs an object of type thread from x, and sets x to a default constructed state.
11  Postconditions: x.get_id() == id() and get_id() returns the value of x.get_id() prior to the start of construction.
12  Throws: Nothing.

30.3.1.3 thread destructor

~thread();
If joinable() then terminate(), otherwise no effects. [Note: Either implicitly detaching or joining a joinable() thread in its destructor could result in difficult to debug correctness (for detach) or performance (for join) bugs encountered only when an exception is raised. Thus the programmer must ensure that the destructor is never executed while the thread is still joinable. — end note]

**Throws:** Nothing.

### 30.3.1.4 thread assignment

```cpp
thread& operator=(thread&& x);
```

**Effects:** If joinable(), calls terminate(). Otherwise, assigns the state of x to *this and sets x to a default constructed state.

**Postconditions:** x.get_id() == id() and get_id() returns the value of x.get_id() prior to the assignment.

**Throws:** Nothing.

### 30.3.1.5 thread members

```cpp
void swap(thread& x);
```

**Effects:** Swaps the state of *this and x.

**Throws:** Nothing.

```cpp
bool joinable() const;
```

**Returns:** get_id() != id()

**Throws:** Nothing.

```cpp
void join();
```

**Requires:** joinable() is true.

**Effects:** Blocks until the thread represented by *this has completed.

**Synchronization:** The completion of the thread represented by *this happens before (1.10) join() returns. [Note: Operations on *this are not synchronized. — end note]

**Postconditions:** The thread represented by *this has completed. get_id() == id().

**Throws:** std::system_error when an exception is required (30.2.2).

**Error conditions:**
- resource_deadlock_would_occur — if deadlock is detected or this->get_id() == std::this_thread::get_id().
- no_such_process — if the thread is not valid.
- invalid_argument — if the thread is not joinable.

```cpp
void detach();
```
Requires: `joinable()` is `true`.

Effects: The thread represented by `*this` continues execution without the calling thread blocking. When `detach()` returns, `*this` no longer represents the possibly continuing thread of execution. When the thread previously represented by `*this` ends execution, the implementation shall release any owned resources.

Postcondition: `get_id() == id()`.

Throws: `std::system_error` when an exception is required (30.2.2).

Error conditions:
- `no_such_process` — if the thread is not valid.
- `invalid_argument` — if the thread is not joinable.

```cpp
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.

30.3.1.6 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.

Throws: Nothing.

30.3.1.7 thread specialized algorithms

```cpp
void swap(thread& x, thread& y);
```

Effects: `x.swap(y)`

30.3.2 Namespace this_thread

```cpp
namespace std {
    namespace this_thread {
        thread::id get_id();

        void yield();
        template <class Clock, class Duration>
            void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
        template <class Rep, class Period>
            void sleep_for(const chrono::duration<Rep, Period>& rel_time);
    }
}
```

```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 implementation the opportunity to reschedule.

Synchronization: None.

Throws: Nothing.

```cpp
template <class Clock, class Duration>
void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
```

Effects: Blocks the calling thread at least until the time specified by `abs_time`.

Synchronization: None.

Throws: Nothing.

```cpp
template <class Rep, class Period>
void sleep_for(const chrono::duration<Rep, Period>& rel_time);
```

Effects: Blocks the calling thread for at least the time specified by `rel_time`.

Synchronization: None.

Throws: Nothing.

### 30.4 Mutual exclusion

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**

```cpp
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 { }

    constexpr defer_lock_t defer_lock { };  
    constexpr try_to_lock_t try_to_lock { };  
    constexpr adopt_lock_t adopt_lock { };  

    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 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;
};

template<class Callable, class ...Args>
void call_once(once_flag& flag, Callable func, Args&&... args);

30.4.1 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 error conditions for error codes, if any, reported by member functions of a Mutex type shall be:

- resource_unavailable_try_again — if any native handle type manipulated is not available.
- operation_not_permitted — if the thread does not have the privilege to perform the operation.
- device_or_resource_busy — if any native handle type manipulated is already locked.
- invalid_argument — if any native handle type manipulated as part of mutex construction is incorrect.

The implementation shall provide lock and unlock operations, as described below. 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 an exception is required (30.2.2).
Error conditions:

- operation_not_permitted — if the thread does not have the privilege to perform the operation.
- resource_deadlock_would_occur — if the implementation detects that a deadlock would occur.
- device_or_resource_busy — if the mutex is already locked and blocking is not possible.

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_exchange` (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.

The expression `m.unlock()` shall be well-formed and have the following semantics:

Requires: The calling thread shall own the mutex.

Effects: Releases the calling thread's ownership of the mutex.

Return type: `void`

Synchronization: This operation synchronizes with (1.10) subsequent lock operations that obtain ownership on the same object.

Throws: Nothing.

### 30.4.1.1 Class mutex

```cpp
namespace std {

class mutex {

public:

    constexpr mutex();
    ~mutex();

    mutex(const mutex&) = delete;
    mutex& operator=(const mutex&) = delete;

    void lock();
    bool try_lock();
    void unlock();

    typedef implementation-defined native_handle_type; // See 30.2.3

    native_handle_type native_handle(); // See 30.2.3

};
}
```

§ 30.4.1.1
The class \texttt{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 \texttt{try\_lock()}) or block (for \texttt{lock()}) until the owning thread has released ownership with a call to \texttt{unlock()}. 

[Note: After a thread \texttt{A} has called \texttt{unlock()}, releasing a mutex, it is possible for another thread \texttt{B} to lock the same mutex, observe that it is no longer in use, unlock it, and destroy it, before thread \texttt{A} appears to have returned from its unlock call. Implementations are required to handle such scenarios correctly, as long as thread \texttt{A} doesn’t access the mutex after the unlock call returns. These cases typically occur when a reference-counted object contains a mutex that is used to protect the reference count. — end note]

The class \texttt{mutex} shall satisfy all the \texttt{Mutex} requirements (30.4.1). It shall be a standard-layout class (9).

[Note: A program may deadlock if the thread that owns a \texttt{mutex} object calls \texttt{lock()} on that object. If the implementation can detect the deadlock, a \texttt{resource\_deadlock\_would\_occur} error condition may be observed. — end note]

The behavior of a program is undefined if it destroys a \texttt{mutex} object owned by any thread or a thread terminates while owning a \texttt{mutex} object.

### 30.4.1.2 Class \texttt{recursive\_mutex} [thread.mutex.recursive]

```cpp
namespace std {
    class recursive_mutex {
    public:
        recursive_mutex();
        ~recursive_mutex();
        recursive_mutex(const recursive_mutex&) = delete;
        recursive_mutex& operator=(const recursive_mutex&) = delete;

        void lock();
        bool try_lock();
        void unlock();

        typedef implementation-defined native_handle_type; // See 30.2.3
        native_handle_type native_handle(); // See 30.2.3
    };
}
```

The class \texttt{recursive\_mutex} provides a recursive mutex with exclusive ownership semantics. If one thread owns a \texttt{recursive\_mutex} object, attempts by another thread to acquire ownership of that object will fail (for \texttt{try\_lock()}) or block (for \texttt{lock()}) until the first thread has completely released ownership.

The class \texttt{recursive\_mutex} shall satisfy all the \texttt{Mutex} requirements (30.4.1). It shall be a standard-layout class (9).

A thread that owns a \texttt{recursive\_mutex} object may acquire additional levels of ownership by calling \texttt{lock()} or \texttt{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 \texttt{recursive\_mutex} object, additional calls to \texttt{try\_lock()} shall fail, and additional calls to \texttt{lock()} shall throw an exception of type \texttt{std::system\_error}. A thread shall call \texttt{unlock()} once for each level of ownership acquired by calls to \texttt{lock()} and \texttt{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 \texttt{recursive\_mutex} object owned by any thread or
— a thread terminates while owning a recursive_mutex object.

### 30.4.2 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 30.4.2, where rel_time denotes an instantiation of duration (20.10.3) and abs_time denotes an instantiation of time_point (20.10.4).

The expression `m.try_lock_for(rel_time)` shall be well-formed and have the following semantics:

- **Requires:** If the tick period of `rel_time` is not exactly convertible to the native tick period, the duration shall be rounded up to the nearest native tick period.
- **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 be obtained if the lock is available, but implementations are expected to make a strong effort to do so. — end note]

- **Return type:** `bool`
- **Returns:** `true` if ownership was obtained, otherwise `false`.
- **Synchronization:** If `try_lock_for()` returns `true`, prior `unlock()` operations on the same object `synchronize with (1.10)` this operation.
- **Throws:** Nothing.

The expression `m.try_lock_until(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 `try_lock_until()` returns `true`, prior `unlock()` operations on the same object `synchronize with (1.10)` this operation.
- **Throws:** Nothing.

### 30.4.2.1 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 Rep, class Period>
  bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
template <class Clock, class Duration>
  bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
void unlock();

typedef implementation-defined native_handle_type;  // See 30.2.3
native_handle_type native_handle();  // See 30.2.3
};

1 The class `timed_mutex` provides a non-recursive mutex with exclusive ownership semantics. If one thread owns a `timed_mutex` object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()`, `try_lock_for()`, and `try_lock_until()`) until the owning thread has released ownership with a call to `unlock()` or the call to `try_lock_for()` or `try_lock_until()` times out (having failed to obtain ownership).

2 The class `timed_mutex` shall satisfy all of the `TimedMutex` requirements (30.4.2). It shall be a standard-layout class (9).

3 The behavior of a program is undefined if:
   — it destroys a `timed_mutex` object owned by any thread,
   — a thread that owns a `timed_mutex` object calls `lock()`, `try_lock()`, `try_lock_for()`, or `try_lock_until()` on that object, or
   — a thread terminates while owning a `timed_mutex` object.

30.4.2.2 Class `recursive_timed_mutex`  

namespace std {
  class recursive_timed_mutex {
    public:
      recursive_timed_mutex();
      ~recursive_timed_mutex();

      recursive_timed_mutex(const recursive_timed_mutex&) = delete;
      recursive_timed_mutex& operator=(const recursive_timed_mutex&) = delete;

      void lock();
      bool try_lock();
      template <class Rep, class Period>
        bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
      template <class Clock, class Duration>
        bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
      void unlock();

      typedef implementation-defined native_handle_type;  // See 30.2.3
      native_handle_type native_handle();  // See 30.2.3
    }
  }
}
The class `recursive_timed_mutex` provides a recursive mutex with exclusive ownership semantics. If one thread owns a `recursive_timed_mutex` object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()` or block (for `lock()`, `try_lock_for()`, and `try_lock_until()`)) until the owning thread has completely released ownership or the call to `try_lock_for()` or `try_lock_until()` times out (having failed to obtain ownership).

The class `recursive_timed_mutex` shall satisfy all of the `TimedMutex` requirements (30.4.2). 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()`, `try_lock_for()`, or `try_lock_until()` on that object. It is unspecified how many levels of ownership may be acquired by a single thread. If a thread has already acquired the maximum level of ownership for a `recursive_timed_mutex` object, additional calls to `try_lock()`, `try_lock_for()`, or `try_lock_until()` shall fail, and additional calls to `lock()` shall throw an exception of type `std::system_error`. A thread shall call `unlock()` once for each level of ownership acquired by calls to `lock()`, `try_lock()`, `try_lock_for()`, and `try_lock_until()`. Only when all levels of ownership have been released may ownership of the object be acquired by another thread.

The behavior of a program is undefined if:
- it destroys a `recursive_timed_mutex` object owned by any thread, or
- a thread terminates while owning a `recursive_timed_mutex` object.

30.4.3 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 { }; // do not acquire ownership of the mutex
    extern const try_to_lock_t try_to_lock { }; // try to acquire ownership of the mutex
    // without blocking
    extern const adopt_lock_t adopt_lock { }; // assume the calling thread has already
    // obtained mutex ownership and manage it
}
```

30.4.3.1 Class template lock_guard

```cpp
namespace std {
    template <class Mutex>
    class lock_guard {
    public:
        typedef Mutex mutex_type;
    
    }
}
```

§ 30.4.3.1 1142
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:
    mutex_type& pm; // exposition only
};

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.

explicit lock_guard(mutex_type& m);

Requires: If mutex_type is not a recursive mutex, the calling thread does not own the mutex m.

Effects: m.lock()

Postcondition: &pm == &m

lock_guard(mutex_type& m, adopt_lock_t);

Requires: The calling thread owns the mutex m.

Postcondition: &pm == &m

Throws: Nothing.

~lock_guard();

Effects: pm.unlock()

Throws: Nothing.

30.4.3.2 Class template unique_lock

[thread.lock.unique]

namespace std {
    template <class Mutex>
    class unique_lock {
        public:
            typedef Mutex mutex_type;

            // 30.4.3.2.1 construct/copy/destroy
            unique_lock();
            explicit unique_lock(mutex_type& m);
            unique_lock(mutex_type& m, defer_lock_t);
            unique_lock(mutex_type& m, try_to_lock_t);
            unique_lock(mutex_type& m, adopt_lock_t);
            template <class Clock, class Duration>
                unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);
            template <class Rep, class Period>
                unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);

§ 30.4.3.2
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 `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.

**30.4.3.2.1 unique_lock constructors, destructor, and assignment**

```cpp
unique_lock();

Effects: Constructs an object of type `unique_lock`.

Postconditions: `pm == 0` and `owns == false`.

Throws: Nothing.
```

```cpp
explicit unique_lock(mutex_type& m);

Requires: If `mutex_type` is not a recursive mutex the calling thread does not own the mutex.
```
Effects: Constructs an object of type `unique_lock` and calls `m.lock()`.

Postconditions: \( pm == \&m \) and \( owns == \text{true} \).

\[ \text{unique_lock(mutex_type}\& m, \text{defer_lock_t}); \]

Effects: Constructs an object of type `unique_lock`.

Postconditions: \( pm == \&m \) and \( owns == \text{false} \).

Throws: Nothing.

\[ \text{unique_lock(mutex_type}\& m, \text{try_to_lock_t}); \]

Requires: If `mutex_type` is not a recursive mutex the calling thread does not own the mutex.

Effects: Constructs an object of type `unique_lock` and calls `m.try_lock()`.

Postconditions: \( pm == \&m \) and \( owns == \text{res} \), where \( \text{res} \) is the value returned by the call to `m.try_lock()`.

Throws: Nothing.

\[ \text{unique_lock(mutex_type}\& m, \text{adopt_lock_t}); \]

Requires: The calling thread own the mutex.

Effects: Constructs an object of type `unique_lock`.

Postconditions: \( pm == \&m \) and \( owns == \text{true} \).

Throws: Nothing.

\[ \text{template <class Clock, class Duration> unique_lock(mutex_type}\& m, const \text{chrono::time_point}<\text{Clock, Duration}>\& \text{abs_time}); \]

Requires: If `mutex_type` is not a recursive mutex the calling thread does not own the mutex.

Effects: Constructs an object of type `unique_lock` and calls `m.try_lock_until(abs_time)`.

Postconditions: \( pm == \&m \) and \( owns == \text{res} \), where \( \text{res} \) is the value returned by the call to `m.try_lock_until(abs_time)`.

Throws: Nothing.

\[ \text{template <class Rep, class Period> unique_lock(mutex_type}\& m, const \text{chrono::duration}<\text{Rep, Period}>\& \text{rel_time}); \]

Requires: If `mutex_type` is not a recursive mutex the calling thread does not own the mutex.

Effects: Constructs an object of type `unique_lock` and calls `m.try_lock_for(rel_time)`.

Postconditions: \( pm == \&m \) and \( owns == \text{res} \), where \( \text{res} \) is the value returned by the call to `m.try_lock_for(rel_time)`.

Throws: Nothing.

\[ \text{unique_lock(unique_lock}\&\& u); \]

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 \( u.owns == \text{false} \).

Throws: Nothing.
28  Effects: If owns calls pm->unlock().
29Postconditions: 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.
30   Throws: Nothing.
31   [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]

32   ~unique_lock();
33   Effects: If owns calls pm->unlock().
34   Throws: Nothing.

30.4.3.2.2 unique_lock locking

void lock();
1   Effects: pm->lock()
2Postcondition: owns == true
3   Throws: Any exception thrown by pm->lock(). std::system_error if an exception is required (30.2.2).
   std::system_error with an error condition of operation_not_permitted if pm is 0. std::system_error
   with an error condition of resource_deadlock_would_occur if on entry owns is true.

bool try_lock();
4   Effects: pm->try_lock()
5Returns: The value returned by the call to try_lock().
6Postcondition: owns == res, where res is the value returned by the call to try_lock().
7   Throws: Any exception thrown by pm->try_lock(). std::system_error if an exception is re-
   quired (30.2.2). std::system_error with an error condition of operation_not_permitted if pm
   is 0. std::system_error with an error condition of resource_deadlock_would_occur if on entry
   owns is true.

template <class Clock, class Duration>
bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
8   Effects: pm->try_lock_until(abs_time)
9Returns: The value returned by the call to try_lock_until(abs_time).
10Postcondition: owns == res, where res is the value returned by the call to try_lock_until(abs_time).
11   Throws: Any exception thrown by pm->try_lock_until(). std::system_error if an exception is
   required (30.2.2). std::system_error with an error condition of operation_not_permitted if pm is
   0. std::system_error with an error condition of resource_deadlock_would_occur if on entry owns
   is true.

template <class Rep, class Period>
bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
Effects: \texttt{pm->try\_lock\_for(rel\_time)}.

Returns: The value returned by the call to \texttt{try\_lock\_until(rel\_time)}.

Postcondition: \texttt{owns == res}, where \texttt{res} is the value returned by the call to \texttt{try\_lock\_for(rel\_time)}.

Throws: Any exception thrown by \texttt{pm->try\_lock\_for()} \texttt{std::system\_error} if an exception is required (30.2.2). \texttt{std::system\_error} with an error condition of \texttt{operation\_not\_permitted} if \texttt{pm} is 0. \texttt{std::system\_error} with an error condition of \texttt{resource\_deadlock\_would\_occur} if on entry \texttt{owns} is true.

\begin{verbatim}
void unlock();
\end{verbatim}

Effects: \texttt{pm->unlock()}

Postcondition: \texttt{owns == false}

Throws: \texttt{std::system\_error} when an exception is required (30.2.2).

Error conditions:
   \texttt{operation\_not\_permitted} — if on entry \texttt{owns} is false.

### 30.4.3.2.3 \texttt{unique\_lock} modifiers

\begin{verbatim}
void swap(unique\_lock& u);
\end{verbatim}

Effects: Swaps the data members of \texttt{*this} and \texttt{u}.

Throws: Nothing.

\begin{verbatim}
mutex\_type *release();
\end{verbatim}

Returns: The previous value of \texttt{pm}.

Postconditions: \texttt{pm == 0} and \texttt{owns == false}.

Throws: Nothing.

\begin{verbatim}
template <class Mutex>
void swap(unique\_lock<Mutex>& x, unique\_lock<Mutex>& y);
\end{verbatim}

Effects: \texttt{x.swap(y)}

Throws: Nothing.

### 30.4.3.2.4 \texttt{unique\_lock} observers

\begin{verbatim}
bool owns\_lock() const;
\end{verbatim}

Returns: \texttt{owns}

Throws: Nothing.

\begin{verbatim}
explicit operator bool() const;
\end{verbatim}

Returns: \texttt{owns}

Throws: Nothing.
mutex_type *mutex() const;

Returns: pm

Throws: Nothing.

### 30.4.4 Generic locking algorithms

[thread.lock.algorithm]

```cpp
template <class L1, class L2, class... L3> int try_lock(L1&, L2&, L3&...);
```

1. **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]
2. **Effects:** Calls try_lock() for each argument in order beginning with the first until all arguments have been processed or a call to try_lock() fails, either by returning false or by throwing an exception. If a call to try_lock() fails, unlock() shall be called for all prior arguments and there shall be no further calls to try_lock().
3. **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.

```cpp
template <class L1, class L2, class... L3> void lock(L1&, L2&, L3&...);
```

4. **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]
5. **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().

### 30.4.5 Call once

[thread.once]

The class once_flag is an opaque data structure that call_once uses to initialize data without causing a data race or deadlock.

#### 30.4.5.1 Struct once_flag

[thread.once.onceflag]

```cpp
constexpr once_flag();
```

1. **Effects:** Constructs an object of type once_flag.
2. **Synchronization:** The construction of a once_flag object is not synchronized.
3. **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.
4. **Throws:** nothing.
30.4.5.2 Function call_once

```cpp
template<class Callable, class ...Args>
void call_once(once_flag& flag, Callable&& func, Args&&... args);
```

Given a function as follows:

```cpp
template <class T> typename decay<T>::type decay_copy(T&& v)
{ return std::forward<T>(v); }
```

1 Requires: Callable and each Ti in Args shall satisfy the MoveConstructible requirements. INVOKE(decay_copy(std::forward<Callable>(func)), decay_copy(std::forward<Args>(args))...) (20.8.2) shall be a valid expression.

2 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, INVOKE(decay_copy(std::forward<Callable>(func)), decay_copy(std::forward<Args>(args))...) is executed. The call to call_once is effective if and only if INVOKE(decay_copy(std::forward<Callable>(func)), decay_copy(std::forward<Args>(args))...) returns without throwing an exception. If an exception is thrown it is propagated to the caller.

3 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.

4 Throws: std::system_error when an exception is required (30.2.2), or any exception thrown by func.

5 Error conditions:

— invalid_argument — if the once_flag object is no longer valid.

[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); }
};
```
30.5 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 an object of type unique_lock<mutex>, allowing maximum efficiency on some platforms. Class condition_variable_any provides a general condition variable that can wait on objects of user-supplied lock types.

Condition variables permit concurrent invocation of the wait, wait_for, wait_until, notify_one and notify_all member functions.

The execution of notify_one and notify_all shall be atomic. The execution of wait, wait_for, and wait_until shall be performed in three atomic parts:
1. the release of the mutex, and entry into the waiting state;
2. the unblocking of the wait; and
3. the reacquisition of the lock.

The implementation shall behave as if notify_one, notify_all, and each part of the wait, wait_for, and wait_until 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;

  void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);

  enum class cv_status { no_timeout, timeout };
}

void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);
```

Requires: lk is locked by the calling thread and either
- no other thread is waiting on cond, or
- lk.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, or wait_until) threads.

Effects: transfers ownership of the lock associated with lk into internal storage and schedules cond to be notified when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed. This notification shall be as if

```cpp
lk.unlock();
cond.notify_all();
```

Note: The supplied lock will be held until the thread exits, and care must be taken to ensure that this does not cause deadlock due to lock ordering issues. After calling notify_all_at_thread_exit it is recommended that the thread should be exited as soon as possible, and that no blocking or time-consuming tasks are run on that thread.
Note: It is the user’s responsibility to ensure that waiting threads do not erroneously assume that the thread has finished if they experience spurious wakeups. This typically requires that the condition being waited for is satisfied while holding the lock on lk, and that this lock is not released and reacquired prior to calling notify_all_at_thread_exit.

30.5.1 Class condition_variable

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 Clock, class Duration>
        cv_status wait_until(unique_lock<mutex>& lock,
                              const chrono::time_point<Clock, Duration>& abs_time);
        template <class Clock, class Duration, class Predicate>
        bool wait_until(unique_lock<mutex>& lock,
                        const chrono::time_point<Clock, Duration>& abs_time,
                        Predicate pred);

        template <class Rep, class Period>
        cv_status wait_for(unique_lock<mutex>& lock,
                            const chrono::duration<Rep, Period>& rel_time);
        template <class Rep, class Period, class Predicate>
        bool wait_for(unique_lock<mutex>& lock,
                      const chrono::duration<Rep, Period>& rel_time,
                      Predicate pred);

        typedef implementation-defined native_handle_type; // See 30.2.3
        native_handle_type native_handle(); // See 30.2.3
    };
}

The class condition_variable shall be a standard-layout class (9).

condition_variable();

Effects: Constructs an object of type condition_variable.

Throws: std::system_error when an exception is required (30.2.2).

Error conditions:

— resource_unavailable_try_again — if some non-memory resource limitation prevents initialization.
Requires: There shall be no thread blocked on *this.  [Note: That is, all threads shall have been notified; they may subsequently block on the lock specified in the wait.  This relaxes the usual rules, which would have required all wait calls to happen before destruction.  Only the notification to unblock the wait must happen before destruction.  The user must take care to ensure that no threads wait on *this once the destructor has been started, especially when the waiting threads are calling the wait functions in a loop or using the overloads of wait, wait_for, or wait_until that take a predicate.  — end note]

Effects: Destroys the object.

Throws: Nothing.

void notify_one();

Effects: If any threads are blocked waiting for *this, unblocks one of those threads.

Throws: Nothing.

void notify_all();

Effects: Unblocks all threads that are blocked waiting for *this.

Throws: Nothing.

void wait(unique_lock<mutex>& lock);

Requires: 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 (via wait or timed_wait) threads.

Effects:
— Atomically calls lock.unlock() and blocks on *this.
— When unblocked, calls lock.lock() (possibly blocking on the lock), then returns.
— The function will unblock when signaled by a call to notify_one() or a call to notify_all(), or spuriously.
— If the function exits via an exception, lock.lock() shall be called prior to exiting the function scope.

Postcondition: lock is locked by the calling thread.

Throws: std::system_error when an exception is required (30.2.2).

Error conditions:
— equivalent error condition from lock.lock() or lock.unlock().

template <class Predicate>
void wait(unique_lock<mutex>& lock, Predicate pred);

Effects:
while (!pred())
    wait(lock);
template <class Clock, class Duration>
  cv_status wait_until(unique_lock<mutex>& lock,
    const chrono::time_point<Clock, Duration>& abs_time);

Requires: 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 (via wait, wait_for, or wait_until) threads.

Effects:
  — Atomically calls lock.unlock() and blocks on *this.
  — When unblocked, calls lock.lock() (possibly blocking on the lock), then returns.
  — The function will unblock when signaled by a call to notify_one() or a call to notify_all(), if
    abs_time <= Clock::now(), or spuriously.
  — If the function exits via an exception, lock.lock() shall be called prior to exiting the function
    scope.

Postcondition: lock is locked by the calling thread.

Returns: cv_status::timeout if the function unblocked because abs_time was reached, otherwise
  cv_status::no_timeout.

Error conditions:
  — operation_not_permitted — if the thread does not own the lock.
  — equivalent error condition from lock.lock() or lock.unlock().

template <class Rep, class Period>
  cv_status wait_for(unique_lock<mutex>& lock,
    const chrono::duration<Rep, Period>& rel_time);

Requires: 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 (via wait, wait_for, or wait_until) threads.

Effects:
  — Atomically calls lock.unlock() and blocks on *this.
  — When unblocked, calls lock.lock() (possibly blocking on the lock), then returns.
  — The function will unblock when signaled by a call to notify_one() or a call to notify_all(), by the elapsed time rel_time passing (30.2.4), or spuriously.
  — If the function exits via an exception, lock.lock() shall be called prior to exiting the function
    scope.

Returns: cv_status::timeout if the function unblocked because rel_time elapsed, otherwise cv_-
  status::no_timeout.

Postcondition: lock is locked by the calling thread.
Throws: system_error when an exception is required (30.2.2).

Error conditions:

— operation_not_permitted — if the thread does not own the lock.
— equivalent error condition from lock.lock() or lock.unlock().

template <class Clock, class Duration, class Predicate>
bool wait_until(unique_lock<mutex>& lock,
    const chrono::time_point<Clock, Duration>& abs_time,
    Predicate pred);

Effects:

while (!pred())
    if (wait_until(lock, abs_time) == cv_status::timeout)
        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 Rep, class Period, class Predicate>
bool wait_for(unique_lock<mutex>& lock,
    const chrono::duration<Rep, Period>& rel_time,
    Predicate pred);

Requires: 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 (via wait, wait_for, or wait_until) threads.

Effects:

— Executes a loop: Within the loop the function first evaluates pred() and exits the loop if the result is true.
— Atomically calls lock.unlock() and blocks on *this.
— When unblocked, calls lock.lock() (possibly blocking on the lock).
— The function will unblock when signaled by a call to notify_one() or a call to notify_all(), by the elapsed time rel_time passing (30.2.4), or spuriously.
— If the function exits via an exception, lock.lock() shall be called prior to exiting the function scope.
— The loop terminates when pred() returns true or when the time duration specified by rel_time has elapsed.

[Note: There is no blocking if pred() is initially true, even if the timeout has already expired. — end note]

Postcondition: lock is locked by the calling thread.

Returns: pred()
Note: The returned value indicates whether the predicate evaluates to true regardless of whether the timeout was triggered. — end note

Throws: system_error when an exception is required (30.2.2).

Error conditions:
- operation_not_permitted — if the thread does not own the lock.
- equivalent error condition from lock.lock() or lock.unlock().

30.5.2 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;

    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, class Clock, class Duration>
      cv_status wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time);
    template <class Lock, class Clock, class Duration, class Predicate>
      void wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time, Predicate pred);
    template <class Lock, class Rep, class Period>
      cv_status wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time);
    template <class Lock, class Rep, class Period, class Predicate>
      bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time, Predicate pred);

    typedef implementation-defined native_handle_type; // See 30.2.3
    native_handle_type native_handle(); // See 30.2.3
  }
}

condition_variable_any();

Effects: Constructs an object of type condition_variable_any.

~condition_variable_any();

Requires: There shall be no thread blocked on *this. [Note: That is, all threads shall have been notified; they may subsequently block on the lock specified in the wait. This relaxes the usual rules, which would have required all wait calls to happen before destruction. Only the notification to unblock the wait must happen before destruction. The user must take care to ensure that no threads wait on

§ 30.5.2
*this once the destructor has been started, especially when the waiting threads are calling the wait functions in a loop or using the overloads of wait, wait_for, or wait_until that take a predicate.
— end note]

Effects: Destroys the object.

Throws: Nothing.

void notify_one();

Effects: If any threads are blocked waiting for *this, unblocks one of those threads.

Throws: Nothing.

void notify_all();

Effects: Unblocks all threads that are blocked waiting for *this.

Throws: Nothing.

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.lock() shall be called prior to exiting the function scope.

Postcondition: lock is locked by the calling thread.

Throws: std::system_error when an exception is required (30.2.2).

Error conditions:
— equivalent error condition from lock.lock() or lock.unlock().

template <class Lock, class Predicate>
void wait(Lock& lock, Predicate pred);

Effects:
while (!pred())
    wait(lock);

template <class Lock, class Clock, class Duration>
cv_status wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time);

Effects:
— Atomically calls lock.unlock() and blocks on *this.
— When unblocked, calls lock.lock() (possibly blocking on the lock) and returns.
— The function will unblock when signaled by a call to notify_one() or a call to notify_all(), if abs_time <= Clock::now(), or spuriously.
— If the function exits via an exception, `lock.lock()` shall be called prior to exiting the function scope.

Postcondition: `lock` is locked by the calling thread.

Returns: `cv_status::timeout` if the function unblocked because `abs_time` was reached, otherwise `cv_status::no_timeout`.

Throws: `std::system_error` when an exception is required (30.2.2).

Error conditions:
— equivalent error condition from `lock.lock()` or `lock.unlock()`.

```cpp
template <class Lock, class Rep, class Period>
void wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time);
```

Effects:
— Atomically calls `lock.unlock()` and blocks on `*this`.
— When unblocked, calls `lock.lock()` (possibly blocking on the lock), then returns.
— The function will unblock when signaled by a call to `notify_one()` or a call to `notify_all()`, by the elapsed time `rel_time` passing (30.2.4), or spuriously.
— If the function exits via an exception, `lock.unlock()` shall be called prior to exiting the function scope.

Returns: `cv_status::timeout` if the function unblocked because `rel_time` elapsed, otherwise `cv_status::no_timeout`.

Postcondition: `lock` is locked by the calling thread.

Throws: `system_error` when an exception is required (30.2.2).

Error conditions:
— equivalent error condition from `lock.lock()` or `lock.unlock()`.

```cpp
template <class Lock, class Duration, class Predicate>
bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time, Predicate pred);
```

Effects:
while (!pred())
  if (wait_until(lock, abs_time) == cv_status::timeout)
    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]

```cpp
template <class Lock, class Rep, class Period, class Predicate>
bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time, Predicate pred);
```

Effects:
— Executes a loop: Within the loop the function first evaluates `pred()` and exits the loop if the result is `true`.

§ 30.5.2
— Atomically calls `lock.unlock()` and blocks on `*this`.
— When unblocked, calls `lock.lock()` (possibly blocking on the lock).
— The function will unblock when signaled by a call to `notify_one()` or a call to `notify_all()`, by the elapsed time `rel_time` passing (30.2.4), or spuriously.
— If the function exits via an exception, `lock.unlock()` shall be called prior to exiting the function scope.
— The loop terminates when `pred()` returns `true` or when the time duration specified by `rel_time` has elapsed.

[Note: There is no blocking if `pred()` is initially true, even if the timeout has already expired. — end note]

Postcondition: `lock` is locked by the calling thread.

Returns: `pred()`

[Note: The returned value indicates whether the predicate evaluates to `true` regardless of whether the timeout was triggered. — end note]

Throws: `system_error` when an exception is required (30.2.2).

Error conditions:
— `operation_not_permitted` — if the thread does not own the lock.
— equivalent error condition from `lock.lock()` or `lock.unlock()`.

### 30.6 Futures

#### 30.6.1 Overview

30.6 describes components that a C++ program can use to retrieve in one thread the result (value or exception) from a function that has run in another thread. [Note: these components are not restricted to multi-threaded programs but can be useful in single-threaded programs as well. — end note]

Header `<future>` synopsis

```cpp
namespace std {
    enum class future_errc {
        broken_promise,
        future_already_retrieved,
        promise_already_satisfied,
        no_state
    };

    enum class launch {
        any,
        async,
        sync
    };

    enum class future_status {
        ready,
        timeout,
        deferred
    }
}
```
template <> struct is_error_code_enum<future_errc> : public true_type { }

error_code make_error_code(future_errc e);
error_condition make_error_condition(future_errc e);

const error_category& future_category();

class future_error;

template <class R> class promise;
template <class R> class promise<R&>;
template <> class promise<void>;

template <class R>
void swap(promise<R>& x, promise<R>& y);

template <class R, class Alloc>
struct uses_allocator<promise<R>, Alloc>;

template <class R> class future;
template <class R> class future<R&>;
template <> class future<void>;

template <class R> class shared_future;
template <class R> class shared_future<R&>;
template <> class shared_future<void>;

template <class R> class atomic_future;
template <class R> class atomic_future<R&>;
template <> class atomic_future<void>;

template <class> class packaged_task; // undefined
template <class R, class... ArgTypes>
class packaged_task<R(ArgTypes...)>;

template <class R>
void swap(packaged_task<R(ArgTypes...)>&, packaged_task<R(ArgTypes...)>&);

template <class R, class Alloc>
struct uses_allocator<packaged_task<R>, Alloc>;

template <class F, class... Args>
future<typename result_of<F(Args...>::type>
async(F&& f, Args&&... args);
template <class F, class... Args>
future<typename result_of<F(Args...>::type>
async(launch policy, F&& f, Args&&... args);
}

30.6.2 Error handling

const error_category& future_category();

Returns: A reference to an object of a type derived from class error_category.
The object's default_error_condition and equivalent virtual functions shall behave as specified for
the class error_category. The object's name virtual function shall return a pointer to the string
"future".

```cpp
error_code make_error_code(future_errc e);
```

Returns: `error_code(static_cast<int>(e), future_category())`.

```cpp
error_condition make_error_condition(future_errc e);
```

Returns: `error_condition(static_cast<int>(e), future_category())`.

### 30.6.3 Class future_error

```cpp
namespace std {

    class future_error : public logic_error {
        public:
            future_error(error_code ec);  // exposition only

            const error_code& code() const throw();
            const char* what() const throw();
    };
}
```

1. `const error_code& code() const throw();`
   Returns: the value of `ec` that was passed to the object's constructor.

2. `const char* what() const throw();`
   Returns: an NTBS incorporating `code().message()`.

### 30.6.4 Associated asynchronous state

Many of the classes introduced in this sub-clause use some state to communicate results. This associated asynchronous state consists of some state information and some (possibly not yet evaluated) result, which can be a (possibly void) value or an exception. [Note: Futures, promises, and tasks defined in this clause reference such associated asynchronous state. — end note]

[Note: The result can be any kind of object including a function to compute that result, as used by `async` when policy is `launch::sync`. — end note]

An asynchronous return object is an object that reads results from an associated asynchronous state.

An asynchronous provider is an object that provides a result to an associated asynchronous state. The result of an associated asynchronous state is set by respective functions on the asynchronous provider. [Note: Such as promises or tasks. — end note] The means of setting the result of an associated asynchronous state is specified in the description of those classes and functions that create such a state object.

When the last reference to an associated asynchronous state is given up, any resources held by that associated asynchronous state are released.

An associated asynchronous state is ready only if it holds a value or an exception ready for retrieval. Waiting for an associated asynchronous state to become ready may invoke code to compute the result on the waiting thread if so specified in the description of the class or function that creates the state object.
Calls to functions that successfully set the stored result of an associated asynchronous state synchronize with (1.10) calls to functions successfully detecting the ready state resulting from that setting. The storage of the result (whether normal or exceptional) into the associated asynchronous state happens before (1.10) that state is set to ready.

Accesses to the same associated asynchronous state conflict (1.10).

### 30.6.5 Class template promise

```cpp
namespace std {
    template <class R>
    class promise {
        public:
            promise();
            template <class Allocator>
                promise(allocator_arg_t, const Allocator& a);
            promise(promise&& rhs);
            promise(const promise& rhs) = delete;
            ~promise();

            // assignment
            promise& operator=(promise&& rhs);
            promise& operator=(const promise& rhs) = delete;
            void swap(promise& other);

            // retrieving the result
            future<R> get_future();

            // setting the result
            void set_value(see below);
            void set_exception(exception_ptr p);

            // setting the result with deferred notification
            void set_value_at_thread_exit(const R& r);
            void set_exception_at_thread_exit(exception_ptr p);
    }

    template <class R>
    void swap(promise<R>& x, promise<R>& y);
    template <class R, class Alloc>
    struct uses_allocator<promise<R>, Alloc>:
        true_type { };;
}
```

The implementation shall provide the template `promise` and two specializations, `promise<R&>` and `promise<void>`. These differ only in the argument type of the member function `set_value`, as set out in its description, below.

```cpp
template <class R, class Alloc>
    struct uses_allocator<promise<R>, Alloc>:
        true_type { };;
```

**Requires:** Alloc shall be an Allocator (20.2.5).

```cpp
promise();
template <class Allocator>
    promise(allocator_arg_t, const Allocator& a);
```
Effects: constructs a promise object and an associated asynchronous state. The second constructor uses the allocator \( \texttt{a} \) to allocate memory for the associated asynchronous state.

\[
\text{promise}(\text{promise}&& \ \texttt{rhs});
\]

Effects: constructs a new promise object and transfers ownership of the associated asynchronous state of \( \texttt{rhs} \) (if any) to the newly-constructed object.

\textit{Postcondition:} \( \texttt{rhs} \) has no associated asynchronous state.

\textit{Throws:} nothing.

\[
\text{~promise}();
\]

Effects: if the associated asynchronous state of \( \ast\texttt{this} \) is not ready, stores an exception object of type future\_error with an error condition of broken\_promise. Any threads blocked in a function waiting for the asynchronous state associated with \( \ast\texttt{this} \) to become ready are unblocked. Destroys \( \ast\texttt{this} \) and releases its reference to its associated asynchronous state if any. If this is the last reference to that associated asynchronous state, destroys that state.

\[
\text{promise}&& \ \text{operator=} (\text{promise}&& \ \texttt{rhs});
\]

Effects: promise\(<\texttt{R}>\)(std::move(\texttt{rhs})).swap(\( \ast\texttt{this} \)).

\textit{Postcondition:} \( \texttt{rhs} \) has no associated asynchronous state. \( \ast\texttt{this} \) has the associated asynchronous state of \( \texttt{rhs} \) prior to the assignment.

\textit{Returns:} \( \ast\texttt{this} \).

\textit{Throws:} nothing.

\[
\text{void swap(promise}&& \ \texttt{other});
\]

Effects: Exchanges the associated asynchronous state of \( \ast\texttt{this} \) and \( \texttt{other} \).

\textit{Postcondition:} \( \ast\texttt{this} \) has the associated asynchronous state (if any) that \( \texttt{other} \) had prior to the call to swap. \( \texttt{other} \) has the associated asynchronous state (if any) that \( \ast\texttt{this} \) had prior to the call to swap.

\textit{Throws:} Nothing.

\[
\text{future}<\texttt{R}> \text{get\_future}();
\]

\textit{Returns:} a future\(<\texttt{R}>\) object with the same associated asynchronous state as \( \ast\texttt{this} \).

\textit{Throws:} future\_error if \( \ast\texttt{this} \) has no associated asynchronous state or if get\_future has already been called on a promise with the same associated asynchronous state as \( \ast\texttt{this} \).

\textit{Error conditions:}

\begin{itemize}
  \item future\_already\_retrieved if get\_future has already been called on a promise with the same associated asynchronous state as \( \ast\texttt{this} \).
  \item no\_state if \( \ast\texttt{this} \) has no associated asynchronous state.
\end{itemize}

\[
\text{void promise}::\text{set\_value(const} \texttt{R} \& \texttt{r});
\]

\[
\text{void promise}::\text{set\_value(}\texttt{R}&& \texttt{r});
\]

\[
\text{void promise<}\texttt{R}>::\text{set\_value(}\texttt{R}&& \texttt{r});
\]

\[
\text{void promise<}\texttt{void}>::\text{set\_value}();
\]
Effects: atomically stores \( r \) in the associated asynchronous state and sets that state to ready. Any threads blocked in a call of a blocking function of any future that refers to the same associated asynchronous state as \(*this\) are unblocked.

Throws:

- `future_error` if its associated asynchronous state already has a stored value or exception, or
- for the first version, any exception thrown by the copy constructor of \( R \), or
- for the second version, any exception thrown by the move constructor of \( R \).

Error conditions:

- `promise_already_satisfied` if its associated asynchronous state already has a stored value or exception.
- `no_state` if \(*this\) has no associated asynchronous state.

Synchronization: calls to `set_value` and `set_exception` on a single `promise` object are serialized. [Note: and they synchronize and serialize with other functions through the referred associated asynchronous state. — end note]

```cpp
void set_exception(exception_ptr p);
```

Effects: atomically stores \( p \) in the associated asynchronous state and sets that state to ready. Any threads blocked in a call of a blocking function of any future that refers to the same associated asynchronous state as \(*this\) are unblocked.

Throws: `future_error` if its associated asynchronous state already has a stored value or exception.

Error conditions:

- `promise_already_satisfied` if its associated asynchronous state already has a stored value or exception.
- `no_state` if \(*this\) has no associated asynchronous state.

Synchronization: calls to `set_value` and `set_exception` on a single `promise` object are serialized. [Note: and they synchronize and serialize with other functions through the referred associated asynchronous state. — end note]

```cpp
void promise::set_value_at_thread_exit(const R& r);
void promise::set_value_at_thread_exit(R&& r);
void promise<R&>::set_value_at_thread_exit(R& r);
void promise<void>::set_value_at_thread_exit();
```

Effects: Stores \( r \) in the associated asynchronous state without making the associated asynchronous state ready immediately. Schedules the associated asynchronous state to be made ready when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed.

Throws: `future_error` if an error condition occurs.

Error conditions:

- `promise_already_satisfied` if its associated asynchronous state already has a stored value or exception.
- `no_state` if \(*this\) has no associated asynchronous state.
void promise::set_exception_at_thread_exit(exception_ptr p);

Effects: Stores `p` in the associated asynchronous state without making the associated asynchronous state ready immediately. Schedules the associated asynchronous state to be made ready when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed.

Throws: `future_error` if an error condition occurs.

Error conditions:

- `promise_already_satisfied` if its associated asynchronous state already has a stored value or exception.
- `no_state` if `*this` has no associated asynchronous state.

```cpp
template <class R>
void swap(promise<R>& x, promise<R>& y);
```

```cpp
Effects: `x.swap(y)`.
```

### 30.6.6 Class template `future`

The class template `future` defines a type for asynchronous return objects which do not share their associated asynchronous state with other asynchronous return objects. A default-constructed `future` object has no associated asynchronous state. A `future` object with associated asynchronous state can be created by functions on asynchronous providers (30.6.4) or by the move constructor and shares its associated asynchronous state with the original asynchronous provider. The result (value or exception) of a `future` object can be set by calling a respective function on an object that shares the same associated asynchronous state.

[Note: member functions of `future` do not synchronize with themselves or with member function of `shared_future`. — end note]

The effect of calling any member function other than the destructor, the move-assignment operator, or `valid` on a `future` object for which `valid() == false` is undefined.

```cpp
namespace std {
    template <class R>
    class future {
    public:
        future();
        future(future &&);
        future(const future& rhs) = delete;
        ~future();
        future& operator=(const future& rhs) = delete;
        future& operator=(future&&);

        // retrieving the value
        see below get();

        // functions to check state
        bool valid() const;

        void wait() const;
        template <class Rep, class Period>
        future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
        template <class Clock, class Duration>
```
future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
}

The implementation shall provide the template `future` and two specializations, `future<R&>` and `future<void>`. These differ only in the return type and return value of the member function `get`, as set out in its description, below.

`future();`

`Effects:` constructs an empty `future` object that does not refer to an associated asynchronous state.

`Postcondition:` `valid() == false`.

`future(future&& rhs);`

`Effects:` move constructs a `future` object that refers to the associated asynchronous state that was originally referred to by `rhs` (if any).

`Postconditions:`
- `valid()` returns the same value as `rhs.valid()` prior to the constructor invocation.
- `rhs.valid() == false`.

`Throws:` nothing.

`~future();`

`Effects:`
- gives up the reference to its associated asynchronous state.
- destroys `*this`.

`future& operator=(future&& rhs);`

`Effects:`
- if `*this` referred to an associated asynchronous state prior to the assignment it gives up this reference.
- assigns the contents of `rhs` to `*this`.

`Postcondition:`
- `valid()` returns the same value as `rhs.valid()` prior to the assignment.
- `rhs.valid() == false`.

`R future::get();`
`R& future<R&>::get();`
`void future<void>::get();`

`Note:` as described above, the template and its two required specializations differ only in the return type and return value of the member function `get`.

`Requires:` `valid() == true`.

`Effects:` waits until the associated asynchronous state is ready, then retrieves the value stored in the associated asynchronous state.

§ 30.6.6
Returns:

— `future::get()` returns the value stored in the object’s associated asynchronous state. If the type of the value is `MoveAssignable` the returned value is moved, otherwise it is copied.

— `future<R&>::get()` returns the reference stored as value in the object’s associated asynchronous state.

— `future<void>::get()` returns nothing.

Throws: the stored exception, if an exception was stored in the associated asynchronous state.

Postcondition: valid() == false.

bool valid() const;

Returns: true only if *this refers to an associated asynchronous state.

void wait() const;

Requires: valid() == true.

Effects: blocks until the associated asynchronous state is ready.

```cpp
template <class Rep, class Period>
future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
```

Requires: valid() == true.

Effects: blocks until the associated asynchronous state is ready or until `rel_time` has elapsed.

Returns:

— `future_status::deferred` if the associated asynchronous state contains a deferred function that is not running.

— `future_status::ready` if the associated state is ready.

— `future_status::timeout` if the function is returning because the time period specified by `rel_time` has elapsed.

```cpp
template <class Clock, class Duration>
future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
```

Requires: valid() == true.

Effects: blocks until the associated asynchronous state is ready or until the current time exceeds `abs_time`.

— `future_status::deferred` if the associated asynchronous state contains a deferred function that is not running.

— `future_status::ready` if the associated state is ready.

— `future_status::timeout` if the function is returning because the time point specified by `abs_time` has been reached.
30.6.7 Class template shared_future

The class template shared_future defines a type for asynchronous return objects which may share their associated asynchronous state with other asynchronous return objects. A default-constructed shared_future object has no associated asynchronous state. A shared_future object with associated asynchronous state can be created by conversion from a future object and shares its associated asynchronous state with the original asynchronous provider (30.6.4) of the associated asynchronous state. The result (value or exception) of a shared_future object can be set by calling a respective function on an object that shares the same associated asynchronous state.

[Note: member functions of shared_future do not synchronize with themselves, but they synchronize with the shared asynchronous associated state. — end note]

The effect of calling any member function other than the destructor, the move-assignment operator, or valid() on a shared_future object for which valid() == false is undefined.

```cpp
namespace std {
    template <class R>
    class shared_future {
    public:
        shared_future();
        shared_future(const shared_future& rhs);
        shared_future(future<R>&& rhs);
        "shared_future();
        shared_future& operator=(const shared_future& rhs);
        shared_future& operator=(shared_future&& rhs);

        // retrieving the value
        see below get() const;

        // functions to check state
        bool valid() const;
        void wait() const;
        template <class Rep, class Period>
        future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
        template <class Clock, class Duration>
        future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
    }
}
```

The implementation shall provide the template shared_future and two specializations, shared_future<R&> and shared_future<void>. These differ only in the return type and return value of the member function get, as set out in its description, below.

shared_future();

Effects: constructs an empty shared_future object that does not refer to an associated asynchronous state.

Postcondition: valid() == false.

Throws: Nothing.
Effects: constructs a \texttt{shared\_future} object that refers to the same associated asynchronous state as \texttt{rhs} (if any).

\textit{Postcondition: valid()} returns the same value as \texttt{rhs.valid()}. 

\begin{flushleft}
\begin{verbatim}
shared_future(future<R>&& rhs);
shared_future(shared_future&& rhs);
\end{verbatim}
\end{flushleft}

Effects: move constructs a \texttt{shared\_future} object that refers to the associated asynchronous state that was originally referred to by \texttt{rhs} (if any).

\textit{Postconditions:}

\begin{itemize}
\item valid() returns the same value as \texttt{rhs.valid()} returned prior to the constructor invocation.
\item \texttt{rhs.valid()} == false.
\end{itemize}

\textit{Throws:} nothing.

\begin{verbatim}
~shared_future();
\end{verbatim}

Effects:

\begin{itemize}
\item gives up the reference to its associated asynchronous state.
\item destroys \texttt{*this}.
\end{itemize}

\begin{verbatim}
shared_future& operator=(shared_future&& rhs);
\end{verbatim}

Effects:

\begin{itemize}
\item if \texttt{*this} refers to an associated asynchronous state it gives up this reference.
\item assigns the contents of \texttt{rhs} to \texttt{*this}.
\end{itemize}

\textit{Postconditions:}

\begin{itemize}
\item valid() returns the same value as \texttt{rhs.valid()} returned prior to the assignment.
\item \texttt{rhs.valid()} == false.
\end{itemize}

\begin{verbatim}
shared_future& operator=(const shared_future& rhs);
\end{verbatim}

Effects:

\begin{itemize}
\item if \texttt{*this} refers to an associated asynchronous state it gives up this reference.
\item assigns the contents of \texttt{rhs} to \texttt{*this}. [\textit{Note:} as a result, \texttt{*this} refers to the same associated asynchronous state as \texttt{rhs} (if any). \textit{— end note}]
\end{itemize}

\textit{Postconditions: valid()} == \texttt{rhs.valid()}. 

\begin{verbatim}
const R& shared_future::get() const;
R& shared_future<R>::get() const;
void shared_future< Void>::get() const;
\end{verbatim}

\textit{Note:} as described above, the template and its two required specializations differ only in the return type and return value of the member function \texttt{get}.

\textit{Requires:} valid() == true.
Effects: `wait()`s until the associated asynchronous state is ready, then retrieves the value stored in the associated asynchronous state.

Returns:

- `shared_future::get()` returns a const reference to the value stored in the object’s associated asynchronous state.
- `shared_future<R&>::get()` returns the reference stored as value in the object’s associated asynchronous state.
- `shared_future<void>::get()` returns nothing.

Throws: the stored exception, if an exception was stored in the associated asynchronous state.

```cpp
bool valid() const;
```

Returns: `true` only if `*this` refers to an associated asynchronous state.

```cpp
void wait() const;
```

Requires: `valid() == true`.

Effects: blocks until the associated asynchronous state is ready.

```cpp
template <class Rep, class Period>
future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
```

Requires: `valid() == true`.

Effects: blocks until the associated asynchronous state is ready or until `rel_time` has elapsed.

Returns:

- `future_status::deferred` if the associated asynchronous state contains a deferred function that is not running.
- `future_status::ready` if the associated state is ready.
- `future_status::timeout` if the function is returning because the time period specified by `rel_time` has elapsed.

```cpp
template <class Clock, class Duration>
future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
```

Requires: `valid() == true`.

Effects: blocks until the associated asynchronous state is ready or until the current time exceeds `abs_time`.

Returns:

- `future_status::deferred` if the associated asynchronous state contains a deferred function that is not running.
- `future_status::ready` if the associated state is ready.
- `future_status::timeout` if the function is returning because the time point specified by `abs_time` has been reached.

§ 30.6.7
30.6.8  Class template atomic_future

The class template `atomic_future` defines a type for asynchronous return objects which may share their associated asynchronous state with other asynchronous return objects. A single `atomic_future` may be shared between threads. A default-constructed `atomic_future` object has no associated asynchronous state. An `atomic_future` object with associated asynchronous state can be created by conversion from a `future` object and shares its associated asynchronous state with the original asynchronous provider of the associated state. The result (value or exception) of an `atomic_future` object can be set by calling a respective function on an object that shares the same associated asynchronous state.

Unlike `future` and `shared_future`, member functions of `atomic_future` other than constructors and destructors are synchronization operations (1.10). Accessor member functions perform acquire operations on the object. All member function calls shall be included in the order of `memory_order_seq_cst` operations (29.3).

```cpp
nenamespace std {
    template <class R>
    class atomic_future {
        public:
            atomic_future();
            atomic_future(const atomic_future& rhs);
            atomic_future(future<R>&& rhs);
            ~atomic_future();
            atomic_future& operator=(const atomic_future& rhs);

            // retrieving the value
            see below get() const;

            // functions to check state
            bool valid() const;

            void wait() const;
            template <class Rep, class Period>
                future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
            template <class Clock, class Duration>
                future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
    };
}
```

atomic_future();
3  
**Effects**: constructs an empty `atomic_future` object that does not refer to an associated asynchronous state.

**Postcondition**: `valid() == false`.

atomic_future(const atomic_future& rhs);
4  
**Effects**: constructs an `atomic_future` object that refers to the same associated asynchronous state as `rhs` (if any).

**Postcondition**: `valid() == rhs.valid()`.

atomic_future(future<R>&& rhs);
5  
**Effects**: constructs an `atomic_future` object that refers to the associated asynchronous state that was originally referred to by `rhs` (if any).
Postcondition:
— valid() returns the same value as rhs.valid() returned prior to the constructor invocation.
— rhs.valid() == false.

~atomic_future();

Effects:
— gives up the reference to its associated asynchronous state.
— destroys *this.

atomic_future& operator=(const atomic_future& rhs);

Effects: assigns the contents of rhs to *this. [Note: as a result, *this refers to the same associated asynchronous state as rhs (if any) after the assignment. — end note]

Synchronization: the assignment performs an acquire operation on rhs and a release operation on *this.

Postcondition: valid() == rhs.valid().

Throws: future_error with an error condition of no_state if the precondition is not met.

const R& atomic_future::get() const;
R& atomic_future<R&>::get() const;
void atomic_future<void>::get() const;

Note: as described above, the template and its two required specializations differ only in the return type and return value of the member function get.

Effects: wait()s until the associated asynchronous state is ready, then retrieves the value stored in the associated asynchronous state.

Returns:
— atomic_future::get() returns a const reference to the value stored in the object’s associated asynchronous state.
— atomic_future<R&>::get() returns the reference stored as value in the object’s associated asynchronous state.
— atomic_future<void>::get() returns nothing.

Throws: the exception in the object’s associated asynchronous state if an exception was stored there or a future_error exception object if an error condition occurs.

Error conditions: no_state if valid() == false. [Note: unlike future, calling get more than once on the same atomic_future object is well defined and produces the result again.

bool valid() const;

Returns: true only if *this refers to an associated asynchronous state.

void wait() const;

Effects: blocks until the associated asynchronous state is ready.

Throws: future_error if an error condition occurs.

§ 30.6.8
22 Error conditions:
   — no_state if *this has no associated asynchronous state.

template <class Rep, class period>
future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;

Effects: blocks until the associated asynchronous state is ready or until rel_time has elapsed.

Returns:
   — future_status::deferred if the associated asynchronous state contains a deferred function that
     is not running.
   — future_status::ready if the associated asynchronous state is ready.
   — future_status::timeout if the function is returning because the time period specified by rel_time
     has elapsed.

Throws: future_error if an error condition occurs.

Error conditions:
   — no_state if *this has no associated asynchronous state.

template <class Clock, class Duration>
future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;

Effects: blocks until the associated asynchronous state is ready or until the current time exceeds
abs_time.

Returns:
   — future_status::deferred if the associated asynchronous state contains a deferred function that
     is not running.
   — future_status::ready if the associated asynchronous state is ready.
   — future_status::timeout if the function is returning because the time point specified by abs_time
     has been reached.

Throws: future_error if an error condition occurs.

Error conditions:
   — no_state if *this has no associated asynchronous state.

30.6.9 Function template async

The template function async provides a mechanism to launch a function potentially in a new thread and
provides the result of the function in a future object with which it shares an associated asynchronous state.

template <class F, class... Args>
future<typename result_of<F(Args...)>::type>
async(F&& f, Args&&... args);

template <class F, class... Args>
future<typename result_of<F(Args...)>::type>
async(launch policy, F&& f, Args&&... args);
Requires: F and each Ti in Args shall satisfy the MoveConstructible requirements. \texttt{INVOKE} (\texttt{decay\_copy(std::forward<F>(f)), decay_copy(std::forward<Args>(args))...}) (20.8.2, 30.3.1.2) shall be a valid expression.

Effects: The first function behaves the same as a call to the second function with a policy argument of \texttt{launch::any} and the same arguments for F and Args. The second function creates an associated asynchronous state that is associated with the returned $\texttt{future}$ object. The further behavior of the second function depends on the policy argument as follows:

- \texttt{launch::async} — executes \texttt{INVOKE} (\texttt{decay_copy(std::forward<F>(f)), decay_copy(std::forward<Args>(args))...}) (20.8.2, 30.3.1.2) as if in a new thread of execution represented by a thread object with the calls to \texttt{decay_copy()} being evaluated in the thread that called \texttt{async}. Any return value is stored as the result in the associated asynchronous state. Any exception propagated from the execution of \texttt{INVOKE} (\texttt{decay_copy(std::forward<F>(f)), decay_copy(std::forward<Args>(args))...}) is stored as the exceptional result in the associated asynchronous state. The thread object is stored in the associated asynchronous state and affects the behavior of any \texttt{future} objects that reference that state.

- \texttt{launch::sync} — Stores \texttt{decay_copy(std::forward<F>(f))} and \texttt{decay_copy(std::forward<Args>(args))...} in the associated asynchronous state. These copies of f and args constitute a deferred function. Invocation of the deferred function evaluates \texttt{INVOKE} (g, xyz) where g is the stored value of \texttt{decay_copy(std::forward<F>(f))} and xyz is the stored copy of \texttt{decay_copy(std::forward<Args>(args))...}. The associated asynchronous state is not ready until the function has completed. The first call to a function waiting for the associated asynchronous state created by this \texttt{async} call to become ready shall invoke the deferred function in the thread that called the waiting function; all other calls waiting for the same associated asynchronous state to become ready shall block until the deferred function has completed.

- \texttt{launch::any} — the implementation may choose either policy above at any call to \texttt{async}. [Note: implementations should defer invocations when no more concurrency can be effectively exploited. — end note]

Returns: an object of type $\texttt{future<\texttt{typename result_of<F(Args...)>::type}}$ that refers to the associated asynchronous state created by this call to \texttt{async}.

Synchronization: the invocation of \texttt{async} happens before (1.10) the invocation of f. [Note: this statement applies even when the corresponding \texttt{future} object is moved to another thread. — end note] If the invocation is not deferred, a call to a waiting function on an asynchronous return object that shares the associated asynchronous state created by this \texttt{async} call shall block until the associated thread has completed. If the invocation is not deferred, the \texttt{join()} on the created thread happens-before (1.10) the first function that successfully detects the ready status of the associated asynchronous state returns or before the function that gives up the last reference to the associated asynchronous state returns, whichever happens first. If the invocation is deferred, the completion of the invocation of the deferred function happens-before the calls to the waiting functions return.

Throws: \texttt{system\_error} if policy is \texttt{launch::async} and the implementation is unable to start a new thread.

Error conditions:

- \texttt{resource\_unavailable\_try\_again} — if policy is \texttt{launch::async} and the system is unable to start a new thread.

Remarks: The first signature shall not participate in overload resolution if \texttt{decay<F>::type} is \texttt{std::launch}.
[Example:

```cpp
int work1(int value);
int work2(int value);
int work(int value) {
    auto handle = std::async([=]{ return work2(value); });
    int tmp = work1(value);
    return tmp + handle.get();  // #1
}
```

[Note: line #1 might not result in concurrency because the `async` call uses the default `launch::any` policy, which may use `launch::sync`, in which case the lambda might not be invoked until the `get()` call; in that case, `work1` and `work2` are called on the same thread and there is no concurrency. — end note] — end example]

### 30.6.10 Class template `packaged_task` [futures.task]

1 The class template `packaged_task` defines a type for wrapping a function or callable object so that the return value of the function or callable object is stored in a future when it is invoked.

2 When the `packaged_task` object is invoked, its stored task is invoked and the result (whether normal or exceptional) stored in the associated asynchronous state. Any futures that share the associated asynchronous state will then be able to access the stored result.

```cpp
namespace std {
    template<class> class packaged_task; // undefined

    template<class R, class... ArgTypes>
    class packaged_task<R(ArgTypes...)> {
        public:
            typedef R result_type;

        // construction and destruction
        packaged_task();
        template <class F>
        explicit packaged_task(F f);
        template <class F, class Allocator>
        explicit packaged_task(allocator_arg_t, const Allocator& a, F f);
        explicit packaged_task(R(*f)(ArgTypes...));
        template <class F>
        explicit packaged_task(F&& f);
        template <class F, class Allocator>
        explicit packaged_task(allocator_arg_t, const Allocator& a, F&& f);

        // no copy
        packaged_task(packaged_task&) = delete;
        packaged_task& operator=(packaged_task&) = delete;

        // move support
        packaged_task(packaged_task&& other);
        packaged_task& operator=(packaged_task&& other);
        void swap(packaged_task& other);

        explicit operator bool() const;
    }
}
```

§ 30.6.10 1174
packaged_task();

Effects: constructs a packaged_task object with no associated asynchronous state and no stored task.

Throws: nothing.

template <class F>
packaged_task(F f);

Effects: constructs a new packaged_task object with an associated asynchronous state and stores a copy of f as the object’s stored task. The constructors that take an Allocator argument use it to allocate memory needed to store the internal data structures.

Throws: any exceptions thrown by the copy or move constructor of f, or std::bad_alloc if memory for the internal data structures could not be allocated.

packaged_task(packaged_task&& other);

Effects: constructs a new packaged_task object and transfers ownership of other’s associated asynchronous state to *this, leaving other with no associated asynchronous state.

Postcondition: other has no associated asynchronous state.

Throws: nothing.

packaged_task& operator=(packaged_task&& other);

Effects: packaged_task< R, ArgTypes... > (other). swap(*this).
`packaged_task();

Effects: if the associated asynchronous state of *this is not ready, stores an exception object of type future_error with an error code of broken_promise. Any threads blocked in a function waiting for the associated asynchronous state of *this to become ready are unblocked. Destroys *this and releases its reference to its associated asynchronous state (if any). If this is the last reference to that associated asynchronous state, destroys that state.

Throws: nothing.

void swap(packaged_task& other);

Effects: exchanges the associated asynchronous states and stored tasks of *this and other.

Postcondition: *this has the same associated asynchronous state and stored task (if any) as other prior to the call to swap. other has the same associated asynchronous state and stored task (if any) as *this prior to the call to swap.

Throws: nothing.

explicit operator bool() const;

Returns: true only if *this has an associated asynchronous state.

Throws: nothing.

future<R> get_future();

Returns: a future object that shares the same associated asynchronous state as *this.

Throws: a future_error object if an error occurs.

Error conditions:

— future_already_retrieved if get_future has already been called on a packaged_task object with the same associated asynchronous state as *this.

— no_state if *this has no associated asynchronous state.

void operator()(ArgTypes... args);

Effects: INVOKE(f, t1, t2, ..., tN, R), where f is the stored task of *this and t1, t2, ..., tN are the values in args... If the task returns normally, the return value is stored as the asynchronous result in the associated asynchronous state of *this, otherwise the exception thrown by the task is stored. The associated asynchronous state of *this is made ready, and any threads blocked in a function waiting for the associated asynchronous state of *this to become ready are unblocked.

Throws: a future_error exception object if there is no associated asynchronous state or the stored task has already been invoked.

Error conditions:

— promise_already_satisfied if the associated asynchronous state is already ready.

— no_state if *this has no associated asynchronous state.

Synchronization: a successful call to operator() synchronizes with (1.10) a call to any member function of a future, shared_future, or atomic_future object that shares the associated asynchronous state of *this. The completion of the invocation of the stored task and the storage of the result (whether normal or exceptional) into the associated asynchronous state happens before (1.10) the
state is set to ready.  \[ Note: \text{operator()}\) synchronizes and serializes with other functions through the associated asynchronous state.  \text{— end note} \]

```cpp
void make_ready_at_thread_exit(ArgTypes... args);
```

**Effects:** \text{INVOKE}(f, t_1, t_2, \ldots, t_N, R), where \( f \) is the stored task and \( t_1, t_2, \ldots, t_N \) are the values in \text{args}... If the task returns normally, the return value is stored as the asynchronous result in the associated asynchronous state of \text{*this}, otherwise the exception thrown by the task is stored. In either case, this shall be done without making the state ready immediately. Schedules the associated asynchronous state to be made ready when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed.

**Throws:** \text{future\_error} if an error condition occurs.

**Error conditions:**
- \text{promise\_already\_satisfied} if the associated asynchronous state already has a stored value or exception.
- \text{no\_state} if \text{*this} has no associated asynchronous state.

```cpp
void reset();
```

**Effects:** returns the object to a state as if a newly-constructed instance had just been assigned to \text{*this} by \text{*this} = \text{packaged\_task}(std::move(f)), where \( f \) is the task stored in \text{*this}.  \[ Note: \) this constructs a new associated asynchronous state for \text{*this}. The old state is discarded, as described in the destructor for \text{packaged\_task}. \text{get\_future} may now be called again for \text{*this}.  \text{— end note} \]

**Throws:**
- \text{bad\_alloc} if memory for the new associated state could not be allocated.
- any exception thrown by the copy constructor of the task stored in the associated state.
- \text{future\_error} with an error condition of \text{no\_state} if \text{*this} has no associated state.

### 30.6.10.2 packaged_task globals

```cpp
template <class R, class... ArgTypes>
void swap(packaged_task<R(ArgTypes...)>& x, packaged_task<R(ArgTypes...)>& y);
```

**Effects:** \text{x.swap(y)}

**Throws:** Nothing.

```cpp
template <class R, class Alloc>
struct uses_allocator<packaged_task<R>, Alloc>;
```

**Requires:** Alloc shall be an Allocator (20.2.5).
Annex 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
denum-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:
    \a hex-quad
    \U hex-quad hex-quad

§  A.2
preprocessing-token:
  header-name
  identifier
  pp-number
  character-literal
  user-defined-character-literal
  string-literal
  user-defined-string-literal
  preprocessing-op-or-punc
  each non-white-space character that cannot be one of the above

token:
  identifier
  keyword
  literal
  operator
  punctuator

header-name:
  < h-char-sequence >
  " q-char-sequence "

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

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

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

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

pp-number:
  digit
  . digit
  pp-number digit
  pp-number identifier-nondigit
  pp-number 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
user-defined-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

integer-suffix:
unsigned-suffix long-suffix_{opt}
unsigned-suffix long-long-suffix_{opt}
long-suffix unsigned-suffix_{opt}
long-long-suffix unsigned-suffix_{opt}

unsigned-suffix: one of
u U

long-suffix: one of
1 L

long-long-suffix: one of
11 LL

§ A.2
character-literal:
  ' c-char-sequence '
  u' c-char-sequence '
  U' c-char-sequence '
  L' c-char-sequence '

c-char-sequence:
c-char
c-char-sequence c-char
c-char:
  any member of the source character set except
    the single-quote ' , backslash \, or new-line character
  escape-sequence
  universal-character-name
escape-sequence:
  simple-escape-sequence
  octal-escape-sequence
  hexadecimal-escape-sequence

simple-escape-sequence: one of
  \ " \? \\ \a \b \f \n \r \t \v
octal-escape-sequence:
  \ octal-digit
  \ octal-digit octal-digit
  \ octal-digit octal-digit octal-digit
hexadecimal-escape-sequence:
  \x hexadecimal-digit
  hexadecimal-escape-sequence hexadecimal-digit

floating-literal:
  fractional-constant exponent-part\textsuperscript{opt} floating-suffix\textsuperscript{opt}
  digit-sequence exponent-part floating-suffix\textsuperscript{opt}
fractional-constant:
  digit-sequence\textsuperscript{opt} . digit-sequence
  digit-sequence .
exponent-part:
  e sign\textsuperscript{opt} digit-sequence
  E sign\textsuperscript{opt} digit-sequence
sign: one of
  + -
digit-sequence:
  digit
digit-sequence digit
floating-suffix: one of
  f l F L
string-literal:
  encoding-prefix\textsuperscript{opt} " s-char-sequence\textsuperscript{opt} "
  encoding-prefix\textsuperscript{opt} R raw-string
encoding-prefix:
  u8
  u
  U
  L
s-char-sequence:
  s-char
  s-char-sequence s-char

s-char:
  any member of the source character set except
  the double-quote "," backslash \, or new-line character
  escape-sequence
  universal-character-name

raw-string:
  " d-char-sequence_opt ( r-char-sequence_opt ) d-char-sequence_opt "

r-char-sequence:
  r-char
  r-char-sequence r-char

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

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

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

boolean-literal:
  false
  true

pointer-literal:
  nullptr

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

user-defined-integer-literal:
  decimal-literal ud-suffix
  octal-literal ud-suffix
  hexadecimal-literal ud-suffix

user-defined-floating-literal:
  fractional-constant exponent-part_opt ud-suffix
  digit-sequence exponent-part ud-suffix

user-defined-string-literal:
  string-literal ud-suffix

user-defined-character-literal:
  character-literal ud-suffix

ud-suffix:
  identifier
A.3 Basic concepts

translation-unit:
  declaration-seq_opt

A.4 Expressions

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

id-expression:
  unqualified-id
  qualified-id

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

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

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

lambda-expression:
  lambda-introducer lambda-declarator_opt compound-statement

lambda-introducer:
  [ lambda-capture_opt ]

lambda-capture:
  capture-default
  capture-list
  capture-default , capture-list

capture-default:
  &
  =

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

    identifier
    & identifier
    this

lambda-declarator:

    ( parameter-declaration-clause ) attribute-specifier opt mutable opt
    exception-specification opt trailing-return-type opt

postfix-expression:

    primary-expression
    postfix-expression [ expression ]
    postfix-expression [ braced-init-list ]
    postfix-expression ( expression-list opt )
    simple-type-specifier ( expression-list opt )
    typename-specifier ( expression-list opt )
    simple-type-specifier braced-init-list
    typename-specifier braced-init-list
    postfix-expression . template opt id-expression
    postfix-expression -> template opt id-expression
    postfix-expression . pseudo-destructor-name
    postfix-expression -> pseudo-destructor-name
    postfix-expression ++
    postfix-expression --
    dynamic_cast < type-id > ( expression )
    static_cast < type-id > ( expression )
    reinterpret_cast < type-id > ( expression )
    const_cast < type-id > ( expression )
    typeid ( expression )
    typeid ( type-id )

expression-list:

    initializer-list

pseudo-destructor-name:

    :: 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
    ~ decltype-specifier

unary-expression:

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

unary-operator: one of

    * & + - ! ~

new-expression:

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

§ A.4 1184
new-placement:
  ( expression-list )

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

new-declarator:
  ptr-operator new-declarator_opt
  noptr-new-declarator

noptr-new-declarator:
  [ expression ] attribute-specifier_opt
  noptr-new-declarator [ constant-expression ] attribute-specifier_opt

new-initializer:
  ( expression-list_opt )
  braced-init-list

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

noexcept-expression:
  noexcept ( expression )

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

pm-expression:
  cast-expression
  pm-expression .* cast-expression
  pm-expression ->* cast-expression

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

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

shift-expression:
  additive-expression
  shift-expression << additive-expression
  shift-expression >> additive-expression

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

§ A.4
exclusive-or-expression:
  and-expression
  exclusive-or-expression \(^\wedge\) and-expression

inclusive-or-expression:
  exclusive-or-expression
  inclusive-or-expression | exclusive-or-expression

logical-and-expression:
  inclusive-or-expression
  logical-and-expression \&\& inclusive-or-expression

logical-or-expression:
  logical-and-expression
  logical-or-expression || logical-and-expression

conditional-expression:
  logical-or-expression
  logical-or-expression ? expression : assignment-expression

assignment-expression:
  conditional-expression
  logical-or-expression assignment-operator initializer-clause
  throw-expression

assignment-operator: one of
  = += /= %= += -= >>= <<= &= \(^=\) |=

equation:
  assignment-expression
  expression , assignment-expression

constant-expression:
  conditional-expression

A.5 Statements [gram.stmt]

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

labeled-statement:
  attribute-specifier_opt identifier : statement
  attribute-specifier_opt case constant-expression : statement
  attribute-specifier_opt 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
    attribute-specifier\_opt type-specifier-seq declarator = initializer-clause
    attribute-specifier\_opt type-specifier-seq declarator braced-init-list

iteration-statement:
    while ( condition ) statement
    do statement while ( expression ) ;
    for ( for-init-statement condition\_opt ; expression\_opt ) statement
    for ( for-range-declaration : expression ) statement

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

for-range-declaration:
    attribute-specifier\_opt type-specifier-seq declarator

jump-statement:
    break ;
    continue ;
    return expression\_opt ;
    return braced-init-list ;
    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
    empty-declaration
    attribute-declaration

block-declaration:
    simple-declaration
    asm-definition
    namespace-alias-definition
    using-declaration
    using-directive
    static\_assert-declaration
    alias-declaration
    opaque-enum-declaration

alias-declaration:
    using identifier = type-id ;
simple-declaration:
   attribute-specifier\textsubscript{opt} decl-specifier-seq\textsubscript{opt} init-declarator-list\textsubscript{opt};

static_assert-declaration:
   static\_assert ( constant-expression, string-literal );

empty-declaration:
   ;

attribute-declaration:
   attribute-specifier;

dcl-specifier:
   storage-class-specifier
   type-specifier
   function-specifier
   friend
typedef
cconstexpr

dcl-specifier-seq:
   dcl-specifier attribute-specifier\textsubscript{opt}
dcl-specifier dcl-specifier-seq

storage-class-specifier:
   register
   static
   thread_local
   extern
   mutable

function-specifier:
   inline
   virtual
   explicit

typedef-name:
   identifier

type-specifier:
   trailing-type-specifier
   class-specifier
   enum-specifier

trailing-type-specifier:
   simple-type-specifier
   elaborated-type-specifier
typename-specifier
cv-qualifier

type-specifier-seq:
   type-specifier attribute-specifier\textsubscript{opt}
type-specifier type-specifier-seq

trailing-type-specifier-seq:
   trailing-type-specifier attribute-specifier\textsubscript{opt}
   trailing-type-specifier trailing-type-specifier-seq
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-specifier

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

dcltype-specifier:
  decltype ( expression )

elaborated-type-specifier:
  class-key attribute-specifier opt ::opt nested-name-specifier opt identifier
  class-key ::opt nested-name-specifier opt template opt simple-template-id
  enum ::opt nested-name-specifier opt identifier

denum-name:
  identifier

denum-specifier:
  enum-head { enumerator-list opt }
  enum-head { enumerator-list , }

denum-head:
  enum-key attribute-specifier opt identifier opt enum-base opt
  enum-key attribute-specifier opt nested-name-specifier identifier
  enum-base opt

denque-enum-declaration:
  enum-key attribute-specifier opt identifier enum-base opt ;

denum-key:
  enum
  enum class
  enum struct

denum-base:
  : type-specifier-seq

denqueator-list:
  enumerator-definition
  enumerator-list , enumerator-definition

denqueator-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\textsubscript{opt} namespace identifier \{ namespace-body \}
extension-namespace-definition:
  inline\textsubscript{opt} namespace original-namespace-name \{ namespace-body \}
unnamed-namespace-definition:
  inline\textsubscript{opt} namespace \{ namespace-body \}
namespace-body:
  declaration-seq\textsubscript{opt}
namespace-alias:
  identifier
namespace-alias-definition:
  namespace identifier = qualified-namespace-specifier ;
qualified-namespace-specifier:
  ::\textsubscript{opt} nested-name-specifier\textsubscript{opt} namespace-name
using-declaration:
  using typename\textsubscript{opt} ::\textsubscript{opt} nested-name-specifier unqualified-id ;
  using :: unqualified-id ;
using-directive:
  attribute-specifier\textsubscript{opt} using namespace ::\textsubscript{opt} nested-name-specifier\textsubscript{opt} namespace-name ;
asm-definition:
  asm ( string-literal ) ;
linkage-specification:
  extern string-literal \{ declaration-seq\textsubscript{opt} \}
  extern string-literal declaration
attribute-specifier:
  [ [ attribute-list ] ]
attribute-list:
  attribute\textsubscript{opt}
  attribute-list , attribute\textsubscript{opt}
  attribute\ldots
  attribute-list , attribute\ldots
attribute:
  attribute-token attribute-argument-clause\textsubscript{opt}
attribute-token:
  identifier
  attribute-scoped-token

\S A.6
attribute-scoped-token:
  attribute-namespace :: identifier

attribute-namespace:
  identifier

attribute-argument-clause:
  ( balanced-token-seq )

balanced-token-seq:
  balanced-token
  balanced-token-seq balanced-token

balanced-token:
  ( balanced-token-seq )
  [ balanced-token-seq ]
  { balanced-token-seq }
  any token other than a parenthesis, a bracket, or a brace

A.7  Declarators  [gram.decl]

init-declarator-list:
  init-declarator
  init-declarator-list , init-declarator

init-declarator:
  declarator initializer_opt

declarator:
  ptr-declarator
  nopptr-declarator parameters-and-qualifiers trailing-return-type

ptr-declarator:
  nopptr-declarator
  ptr-operator ptr-declarator

nopptr-declarator:
  declarator-id attribute-specifier_opt
  nopptr-declarator parameters-and-qualifiers
  nopptr-declarator [ constant-expression_opt ] attribute-specifier_opt
  ( ptr-declarator )

parameters-and-qualifiers:
  ( parameter-declaration-clause ) attribute-specifier_opt cv-qualifier-seq_opt
  ref-qualifier_opt exception-specification_opt

trailing-return-type:
  -> trailing-type-specifier-seq abstract-declarator_opt

ptr-operator:
  * attribute-specifier_opt cv-qualifier-seq_opt
  & attribute-specifier_opt
  && attribute-specifier_opt
  :: opt nested-name-specifier * attribute-specifier_opt cv-qualifier-seq_opt

cv-qualifier-seq:
  cv-qualifier cv-qualifier-seq_opt

cv-qualifier:
  const
  volatile

ref-qualifier:
  &
  &&
declarator-id:
  ... opt id-expression
  : : opt nested-name-specifier opt class-name

type-id:
  type-specifier-seq abstract-declarator opt

abstract-declarator:
  ptr-abstract-declarator
  noptr-abstract-declarator opt parameters-and-qualifiers trailing-return-type...

ptr-abstract-declarator:
  noptr-abstract-declarator
  ptr-operator ptr-abstract-declarator opt

nptr-abstract-declarator:
  noptr-abstract-declarator opt parameters-and-qualifiers
  noptr-abstract-declarator opt [ constant-expression ] attribute-specifier opt
  ( ptr-abstract-declarator )

parameter-declaration-clause:
  parameter-declaration-list opt ... opt
  parameter-declaration-list , ...

parameter-declaration-list:
  parameter-declaration
  parameter-declaration-list , parameter-declaration

parameter-declaration:
  attribute-specifier opt decl-specifier-seq declarator
  attribute-specifier opt decl-specifier-seq declarator = assignment-expression
  attribute-specifier opt decl-specifier-seq abstract-declarator opt
  attribute-specifier opt decl-specifier-seq abstract-declarator opt = assignment-expression

function-definition:
  attribute-specifier opt decl-specifier-seq opt declarator function-body
  attribute-specifier opt decl-specifier-seq opt declarator = default ;
  attribute-specifier opt decl-specifier-seq opt declarator = delete ;

function-body:
  ctor-initializer opt compound-statement
  function-try-block

initializer:
  brace-or-equal-initializer
  ( expression-list )

brace-or-equal-initializer:
  = initializer-clause
  braced-init-list

initializer-clause:
  assignment-expression
  braced-init-list

initializer-list:
  initializer-clause ... opt
  initializer-list , initializer-clause ... opt

braced-init-list:
  { initializer-list , opt }
  { }
A.8 Classes

```latex
\begin{verbatim}
class-name:
  identifier
  simple-template-id
class-specifier:
  class-head { member-specification\opt }
class-head:
  class-key attribute-specifier\opt identifier\opt base-clause\opt
class-key attribute-specifier\opt nested-name-specifier identifier base-clause\opt
class-key attribute-specifier\opt nested-name-specifier\opt simple-template-id base-clause\opt
class-key:
  class
  struct
  union
member-specification:
  member-declaration member-specification\opt
  access-specifier : member-specification\opt
member-declaration:
  attribute-specifier\opt decl-specifier-seq\opt
    member-declarator-list\opt ;
  function-definition \opt
    ::\opt nested-name-specifier template\opt unqualified-id ;
  using-declaration
  static_assert-declaration
template-declaration
member-declarator-list:
  member-declarator
  member-declarator-list , member-declarator
member-declarator:
  declarator pure-specifier\opt
  declarator brace-or-equal-initializer\opt
  identifier\opt attribute-specifier\opt : constant-expression
pure-specifier:
  = 0
\end{verbatim}
```

A.9 Derived classes

```latex
\begin{verbatim}
base-clause:
  : base-specifier-list
base-specifier-list:
  base-specifier \opt
  base-specifier-list , base-specifier \opt
base-specifier:
  attribute-specifier\opt base-type-specifier
  attribute-specifier\opt virtual access-specifier\opt base-type-specifier
  attribute-specifier\opt access-specifier virtual base-type-specifier
class-or-decltype:
  ::\opt nested-name-specifier\opt class-name
  decltype-specifier
\end{verbatim}
```
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 braced-init-list
mem-initializer-id:
   class-or-decltype
   identifier

A.11 Overloading

operator-function-id:
   operator operator
operator:
   one of
   new delete new[] delete[]
   + - * / % - - & | ~
   ! = < > += -= *= /= %=
   *= &= |= <<= != >>== <<= == !=
   <= >= && || ++ -- , ->* ->
   () []

literal-operator-id:
   operator "" identifier

A.12 Templates

template-declaration:
   template < template-parameter-list > declaration

template-parameter-list:
   template-parameter
   template-parameter-list , template-parameter

template-parameter:
   type-parameter
   parameter-declaration
type-parameter:
  class ...opt identifier_opt
  class identifier_opt = type-id
  typename ...opt identifier_opt
  typename identifier_opt = type-id
  template < template-parameter-list > class ...opt identifier_opt
  template < template-parameter-list > class identifier_opt = id-expression

simple-template-id:
  template-name < template-argument-list_opt >

template-id:
  simple-template-id
  operator-function-id < template-argument-list_opt >
  literal-operator-id < template-argument-list_opt >

template-name:
  identifier

template-argument-list:
  template-argument ...opt
  template-argument-list , template-argument ...opt

template-argument:
  constant-expression
  type-id
  id-expression

typename-specifier:
  typename ::opt nested-name-specifier identifier
  typename ::opt nested-name-specifier template_opt simple-template-id

explicit-instantiation:
  extern_opt template declaration

explicit-specialization:
  template < > declaration

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:
  attribute-specifier_opt type-specifier-seq declarator
  attribute-specifier_opt type-specifier-seq abstract-declarator_opt
  ...

throw-expression:
  throw assignment-expression_opt

exception-specification:
  dynamic-exception-specification
  noexcept-specification
dynamic-exception-specification:
  throw ( type-id-list_opt )

preprocessing-file:
  group_opt

  group:
  group-part
  group group-part

  group-part:
  if-section
  control-line
  text-line
  # non-directive

  if-section:
  if-group elif-groups opt else-group opt endif-line

  if-group:
  # if   constant-expression new-line group_opt
  # ifdef identifier new-line group_opt
  # ifndef identifier new-line group_opt

  elif-groups:
  elif-group
  elif-groups elif-group

  elif-group:
  # elif  constant-expression new-line group_opt

  else-group:
  # else  new-line group_opt

  endif-line:
  # endif  new-line

  control-line:
  # include pp-tokens new-line
  # define identifier replacement-list new-line
  # define identifier (paren identifier-list_opt ) replacement-list new-line
  # define identifier (paren ... ) replacement-list new-line
  # define identifier (paren identifier-list, ... ) replacement-list new-line
  # undef identifier new-line
  # line pp-tokens new-line
  # error pp-tokens_opt new-line
  # pragma pp-tokens_opt new-line
  # new-line

  text-line:
  pp-tokens_opt new-line

A.14 Preprocessing directives

[gram.cpp]
non-directive:
   pp-tokens new-line

lparen:
   a ( character not immediately preceded by white-space

identifier-list:
   identifier
   identifier-list , identifier

replacement-list:
   pp-tokens opt

pp-tokens:
   preprocessing-token
   pp-tokens preprocessing-token

ew-line:
   the new-line character
Annex B  (informative)
Implementation quantities

1 Because computers are finite, C++ implementations are inevitably limited in the size of the programs they can successfully process. Every implementation shall document those limitations where known. This documentation may cite fixed limits where they exist, say how to compute variable limits as a function of available resources, or say that fixed limits do not exist or are unknown.

2 The limits may constrain quantities that include those described below or others. The bracketed number following each quantity is recommended as the minimum for that quantity. However, these quantities are only guidelines and do not determine compliance.

- 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 [65536].
- Identifiers with block scope declared in one block [1024].
- Macro identifiers simultaneously defined in one translation unit [65536].
- 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 [65536].
- Characters in a string literal (after concatenation) [65536].
- Size of an object [262144].
- Nesting levels for #include files [256].
- Case labels for a switch statement (excluding those for any nested switch statements) [16384].
- Data members in a single class [16384].
- Enumeration constants in a single enumeration [4096].
- Levels of nested class definitions in a single member-specification [256].
- Functions registered by atexit() [32].
— Direct and indirect base classes [16384].
— Direct base classes for a single class [1024].
— Members declared in a single class [4096].
— Final overriding virtual functions in a class, accessible or not [16384].
— Direct and indirect virtual bases of a class [1024].
— Static members of a class [1024].
— Friend declarations in a class [4096].
— Access control declarations in a class [4096].
— Member initializers in a constructor definition [6144].
— Scope qualifications of one identifier [256].
— Nested external specifications [1024].
— Recursive constexpr function invocations [512].
— Template arguments in a template declaration [1024].
— Recursively nested template instantiations [1024].
— Handlers per try block [256].
— Throw specifications on a single function declaration [256].
— Number of placeholders (20.8.10.1.3) [10].
Annex C  (informative)
Compatibility

Appendix 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

2.4
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.

2.12
Change: New Keywords
New keywords are added to C++; see 2.12.
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.

2.14.3
Change: Type of character literal is changed from int to char
Rationale: This is needed for improved overloaded function argument type matching. For example:

```c
int function( int i );
int function( char c );

function( 'x' );
```

It is preferable that this call match the second version of function rather than the first.
Effect on original feature: Change to semantics of well-defined feature. ISO C programs which depend on

§ C.1.1

1200
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 2.14.5:

**Change:** String literals made const

The type of a string literal is changed from “array of char” to “array of const char.” The type of a char16_t string literal is changed from “array of some-integer-type” to “array of const char16_t.” The type of a char32_t string literal is changed from “array of some-integer-type” to “array of const char32_t.” The type of a wide string literal is changed from “array of wchar_t” to “array of const wchar_t.”

**Rationale:** This avoids calling an inappropriate overloaded function, which might expect to be able to modify its argument.

**Effect on original feature:** Change to semantics of well-defined feature.

**Difficulty of converting:** Syntactic transformation. The fix is to add a cast:

```c
char* p = "abc";       // valid in C, invalid in C++
void f(char*) {
    char* p = (char*)"abc";  // OK: cast added
    f(p);
    f((char*)"def");        // OK: cast added
}
```

**How widely used:** Programs that have a legitimate reason to treat string literals as pointers to potentially modifiable memory are probably rare.

C.1.2 Clause 3: basic concepts [diff.basic]

3.1

**Change:** C++ does not have “tentative definitions” as in C E.g., at file scope,

```c
int i;
int i;
```

is valid in C, invalid in C++. This makes it impossible to define mutually referential file-local static objects, if initializers are restricted to the syntactic forms of C. For example,

```c
struct X { int i; struct X *next; };

static struct X a;
static struct X b = { 0, &a };
static struct X a = { 1, &b };
```

**Rationale:** This avoids having different initialization rules for 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

```c
char a[10];
void *b=a;
void foo() {
    char *c=b;
}
```

ISO C will accept this usage of pointer to void being assigned to a pointer to object type. C++ will not.
Rationale: C++ tries harder than C to enforce compile-time type safety.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Could be automated. Violations will be diagnosed by the C++ translator. The fix is to add a cast For example:

```c
char *c = (char *) b;
```

How widely used: This is fairly widely used but it is good programming practice to add the cast when assigning pointer-to-void to pointer-to-object. Some ISO C translators will give a warning if the cast is not used.

§ C.1.2
4.10
Change: Only pointers to non-const and non-volatile objects may be implicitly converted to \texttt{void*}
Rationale: This improves type safety.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Could be automated. A C program containing such an implicit conversion from, e.g., pointer-to-const-object to void* will receive a diagnostic message. The correction is to add an explicit cast.
How widely used: Seldom.

C.1.3 Clause 5: expressions

5.2.2
Change: Implicit declaration of functions is not allowed
Rationale: The type-safe nature of C++.
Effect on original feature: Deletion of semantically well-defined feature. Note: the original feature was labeled as “obsolescent” in ISO C.
Difficulty of converting: Syntactic transformation. Facilities for producing explicit function declarations are fairly widespread commercially.
How widely used: Common.

5.3.3, 5.4
Change: Types must be declared in declarations, not in expressions In C, a sizeof expression or cast expression may create a new type. For example,

\begin{verbatim}
p = (void*)(struct x {int i;} *)0;
\end{verbatim}

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,

\begin{verbatim}
char arr[100];
sizeof(0, arr)
\end{verbatim}
yields 100 in C++ and \texttt{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, such as 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;
};
```

Rationale: Storage class specifiers don’t have any meaning when associated with a type. In C++, class members can be declared with the static storage class specifier. Allowing storage class specifiers on type declarations could render the code confusing for users.

Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Seldom.

7.1.3
Change: A C++ typedef name must be different from any class type name declared in the same scope (except if the typedef is a synonym of the class name with the same name). In C, a typedef name and a struct tag name declared in the same scope can have the same name (because they have different name spaces).

Example:

```c
typedef struct name1 { /*...*/ } name1;       // valid C and C++
struct name { /*...*/ };                     // valid C and C++
typedef int name;                            // valid C, invalid C++
```
Rationale: For ease of use, C++ doesn’t require that a type name be prefixed with the keywords `class`, `struct` or `union` when used in object declarations or type casts.

Example:

```c
class name { /*...*/ };   // i has type class name
name i;
```

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation. One of the 2 types has to be renamed.

How widely used: Seldom.

7.1.6 [see also 3.5]
Change: `const` objects must be initialized in C++ but can be left uninitialized in C

Rationale: A `const` object cannot be assigned to so it must be initialized to hold a useful value.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation.

How widely used: Seldom.

7.1.6 (type specifiers)
Change: Banning implicit int

In C++ a `decl-specifier-seq` must contain a `type-specifier`. In the following example, the left-hand column presents valid C; the right-hand column presents equivalent C++:

```c
void f(const parm); void f(const int parm);
const n = 3; const int n = 3;
main() int main()
    /* ... */
```

Rationale: In C++, implicit int creates several opportunities for ambiguity between expressions involving function-like casts and declarations. Explicit declaration is increasingly considered to be proper style. Liaison with WG14 (C) indicated support for (at least) deprecating implicit int in the next revision of C.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Syntactic transformation. Could be automated.

How widely used: Common.

7.1.6.4
Change: The keyword `auto` cannot be used as a storage class specifier.

```c
void f() {
    auto int x;   // valid C, invalid C++
}
```

Rationale: Allowing the use of `auto` to deduce the type of a variable from its initializer results in undesired interpretations of `auto` as a storage class specifier in certain contexts.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Syntactic transformation.

How widely used: Rare.

7.2
Change: C++ objects of enumeration type can only be assigned values of the same enumeration type. In C, objects of enumeration type can be assigned values of any integral type.
Example:

```c
enum color { red, blue, green};
enum color c = 1; // valid C, invalid C++
```

**Rationale:** The type-safe nature of C++.

**Effect on original feature:** Deletion of semantically well-defined feature.

**Difficulty of converting:** Syntactic transformation. (The type error produced by the assignment can be automatically corrected by applying an explicit cast.)

**How widely used:** Common.

7.2

**Change:** In C++, the type of an enumerator is its enumeration. In C, the type of an enumerator is `int`.

Example:

```c
enum e { A };
sizeof(A) == sizeof(int) // in C
sizeof(A) == sizeof(e)  // in C++
/* and sizeof(int) is not necessarily equal to sizeof(e) */
```

**Rationale:** In C++, an enumeration is a distinct type.

**Effect on original feature:** Change to semantics of well-defined feature.

**Difficulty of converting:** Semantic transformation.

**How widely used:** Seldom. The only time this affects existing C code is when the size of an enumerator is taken. Taking the size of an enumerator is not a common C coding practice.

C.1.6 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 “obsolete” 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:
void f( struct S { int a; } arg ) {} // valid C, invalid C++
enum E { A, B, C } f() {} // valid C, invalid C++

Rationale: When comparing types in different compilation units, C++ relies on name equivalence when
C relies on structural equivalence. Regarding parameter types: since the type defined in an parameter list
would be in the scope of the function, the only legal calls in C++ would be from within the function itself.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. The type definitions must be moved to file scope, or
in header files.
How widely used: Seldom. This style of type definitions is seen as poor coding style.

8.4
Change: In C++, the syntax for function definition excludes the “old-style” C function. In C, “old-style”
syntax is allowed, but deprecated as “obsolescent.”
Rationale: Prototypes are essential to type safety.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Common in old programs, but already known to be obsolescent.

8.5.2
Change: In C++, when initializing an array of character with a string, the number of characters in the
string (including the terminating ‘\0’) must not exceed the number of elements in the array. In C, an array
can be initialized with a string even if the array is not large enough to contain the string-terminating ‘\0’
Example:
char array[4] = "abcd"; // valid C, invalid C++

Rationale: When these non-terminated arrays are manipulated by standard string routines, there is poten-
tial for major catastrophe.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. The arrays must be declared one element bigger to
contain the string terminating ‘\0’.
How widely used: Seldom. This style of array initialization is seen as poor coding style.

C.1.7 Clause 9: classes

9.1 [see also 7.1.3]
Change: In C++, a class declaration introduces the class name into the scope where it is declared and
hides any object, function or other declaration of that name in an enclosing scope. In C, an inner scope
declaration of a struct tag name never hides the name of an object or function in an outer scope
Example:
int x[99];
void f() {
    struct x { int a; };
    sizeof(x); /* size of the array in C */
    /* size of the struct in C++ */
}

Rationale: This is one of the few incompatibilities between C and C++ that can be attributed to the new
C++ name space definition where a name can be declared as a type and as a non-type in a single scope causing the non-type name to hide the type name and requiring that the keywords class, struct, union or enum be used to refer to the type name. This new name space definition provides important notational conveniences to C++ programmers and helps making the use of the user-defined types as similar as possible to the use of built-in types. The advantages of the new name space definition were judged to outweigh by far the incompatibility with C described above.

**Effect on original feature:** Change to semantics of well-defined feature.

**Difficulty of converting:** Semantic transformation. If the hidden name that needs to be accessed is at global scope, the :: C++ operator can be used. If the hidden name is at block scope, either the type or the struct tag has to be renamed.

**How widely used:** Seldom.

### 9.7

**Change:** In C++, the name of a nested class is local to its enclosing class. In C the name of the nested class belongs to the same scope as the name of the outermost enclosing class.

Example:

```c
struct X {  
  struct Y { /* ... */ } y;  
};  
struct Y yy;  // valid C, invalid C++
```

**Rationale:** C++ classes have member functions which require that classes establish scopes. The C rule would leave classes as an incomplete scope mechanism which would prevent C++ programmers from maintaining locality within a class. A coherent set of scope rules for C++ based on the C rule would be very complicated and C++ programmers would be unable to predict reliably the meanings of nontrivial examples involving nested or local functions.

**Effect on original feature:** Change of semantics of well-defined feature.

**Difficulty of converting:** Semantic transformation. To make the struct type name visible in the scope of the enclosing struct, the struct tag could be declared in the scope of the enclosing struct, before the enclosing struct is defined. Example:

```c
struct Y;  // struct Y and struct X are at the same scope  
struct X {  
  struct Y { /* ... */ } y;  
};
```

1 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++
};
```

§ C.1.7 1208
Rationale: When classes become complicated, allowing such a redefinition after the type has been used can create confusion for C++ programmers as to what the meaning of T really is.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation. Either the type or the struct member has to be renamed.

How widely used: Seldom.

C.1.8 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; }
volatile struct X x1 = {0};
struct X x2(x1); // invalid C++
struct X x3;
x3 = x1; // also invalid C++
```

Rationale: Several alternatives were debated at length. Changing the parameter to volatile const X& would greatly complicate the generation of efficient code for class objects. Discussion of providing two alternative signatures for these implicitly-defined operations raised unanswered concerns about creating ambiguities and complicating the rules that specify the formation of these operators according to the bases and members.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation. If volatile semantics are required for the copy, a user-declared constructor or assignment must be provided. [Note: this user-declared constructor may be explicitly defaulted. — end note] If non-volatile semantics are required, an explicit const_cast can be used.

How widely used: Seldom.

C.1.9 Clause 16: preprocessing directives

16.8 (predefined names)

Change: Whether _STDC__ is defined and if so, what its value is, are implementation-defined

Rationale: C++ is not identical to ISO C. Mandating that _STDC__ be defined would require that translators make an incorrect claim. Each implementation must choose the behavior that will be most useful to its marketplace.

Effect on original feature: Change to semantics of well-defined feature.

Difficulty of converting: Semantic transformation.

How widely used: Programs and headers that reference _STDC__ are quite common.

C.2 Standard C library

This subclause summarizes the contents of the C++ standard library included from the Standard C library. It also summarizes the explicit changes in definitions, declarations, or behavior from the Standard C library noted in other subclauses (17.6.1.2, 18.2, 21.7).

The C++ standard library provides 57 standard macros from the C library, as shown in Table 146.
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).

<table>
<thead>
<tr>
<th>Table 146 — Standard macros</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert</td>
</tr>
<tr>
<td>BUFSIZ</td>
</tr>
<tr>
<td>CLOCK_PER_SEC</td>
</tr>
<tr>
<td>EDOM</td>
</tr>
<tr>
<td>EILSEQ</td>
</tr>
<tr>
<td>EOF</td>
</tr>
<tr>
<td>ERANGE</td>
</tr>
<tr>
<td>errno</td>
</tr>
<tr>
<td>EXIT_FAILURE</td>
</tr>
<tr>
<td>EXIT_SUCCESS</td>
</tr>
<tr>
<td>FILENAME_MAX</td>
</tr>
<tr>
<td>FOPEN_MAX</td>
</tr>
</tbody>
</table>

4 The C++ standard library provides 57 standard values from the C library, as shown in Table 147.

<table>
<thead>
<tr>
<th>Table 147 — Standard values</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR_BIT</td>
</tr>
<tr>
<td>CHAR_MAX</td>
</tr>
<tr>
<td>CHAR_MIN</td>
</tr>
<tr>
<td>DBL_DIG</td>
</tr>
<tr>
<td>DBL_EPSILON</td>
</tr>
<tr>
<td>DBL_MANT_DIG</td>
</tr>
<tr>
<td>DBL_MAX</td>
</tr>
<tr>
<td>DBL_MAX_10_EXP</td>
</tr>
<tr>
<td>DBL_MAX_EXP</td>
</tr>
<tr>
<td>DBL_MIN</td>
</tr>
<tr>
<td>DBL_MIN_10_EXP</td>
</tr>
<tr>
<td>DBL_MIN_EXP</td>
</tr>
</tbody>
</table>

5 The C++ standard library provides 20 standard types from the C library, as shown in Table 148.

<table>
<thead>
<tr>
<th>Table 148 — Standard types</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock_t</td>
</tr>
<tr>
<td>div_t</td>
</tr>
<tr>
<td>FILE</td>
</tr>
<tr>
<td>fpos_t</td>
</tr>
<tr>
<td>jmp_buf</td>
</tr>
</tbody>
</table>

6 The C++ standard library provides 2 standard structs from the C library, as shown in Table 149.

<table>
<thead>
<tr>
<th>Table 149 — Standard structs</th>
</tr>
</thead>
<tbody>
<tr>
<td>lconv</td>
</tr>
</tbody>
</table>

7 The C++ standard library provides 209 standard functions from the C library, as shown in Table 150.
Table 150 — Standard functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>fmod</td>
<td>isupper</td>
<td>mktime</td>
<td>strftime</td>
<td>wcrtomb</td>
</tr>
<tr>
<td>abs</td>
<td>fopen</td>
<td>iswalnum</td>
<td>modf</td>
<td>strlen</td>
<td>wcscat</td>
</tr>
<tr>
<td>acos</td>
<td>fprintf</td>
<td>iswalalpha</td>
<td>perror</td>
<td>strncat</td>
<td>wcscchr</td>
</tr>
<tr>
<td>asctime</td>
<td>fputc</td>
<td>iswchar</td>
<td>printf</td>
<td>strncpy</td>
<td>wcscoll</td>
</tr>
<tr>
<td>asin</td>
<td>fputs</td>
<td>iswctype</td>
<td>printf</td>
<td>strncpy</td>
<td>wcscoll</td>
</tr>
<tr>
<td>atan</td>
<td>fputwc</td>
<td>iswdigit</td>
<td>putc</td>
<td>strpbrk</td>
<td>wcscpy</td>
</tr>
<tr>
<td>atan2</td>
<td>fputws</td>
<td>iswgraph</td>
<td>putchar</td>
<td>strrchr</td>
<td>wcscspn</td>
</tr>
<tr>
<td>atexit</td>
<td>fread</td>
<td>iswlower</td>
<td>puts</td>
<td>strspn</td>
<td>wcftime</td>
</tr>
<tr>
<td>atof</td>
<td>free</td>
<td>iswprint</td>
<td>putwc</td>
<td>strstr</td>
<td>wcslen</td>
</tr>
<tr>
<td>atoi</td>
<td>freopen</td>
<td>iswpunct</td>
<td>putwchar</td>
<td>strtod</td>
<td>wcscnt</td>
</tr>
<tr>
<td>atol</td>
<td>frexp</td>
<td>iswspace</td>
<td>qsort</td>
<td>strtok</td>
<td>wcscncmp</td>
</tr>
<tr>
<td>bsearch</td>
<td>fscanf</td>
<td>iswupper</td>
<td>raise</td>
<td>strtol</td>
<td>wcscpy</td>
</tr>
<tr>
<td>btowc</td>
<td>fseek</td>
<td>iswxdigit</td>
<td>rand</td>
<td>strtol</td>
<td>wcspbrk</td>
</tr>
<tr>
<td>calloc</td>
<td>fsetpos</td>
<td>isxdigit</td>
<td>realloc</td>
<td>strxfrm</td>
<td>wcsrcr</td>
</tr>
<tr>
<td>ceil</td>
<td>ftell</td>
<td>labs</td>
<td>remove</td>
<td>swprintf</td>
<td>wcsrtombs</td>
</tr>
<tr>
<td>clearerr</td>
<td>fwrite</td>
<td>ldexp</td>
<td>rename</td>
<td>swscanf</td>
<td>wcsspn</td>
</tr>
<tr>
<td>clock</td>
<td>fprintf</td>
<td>div</td>
<td>rewind</td>
<td>system</td>
<td>wcsstr</td>
</tr>
<tr>
<td>cos</td>
<td>fwrite</td>
<td>localeconv</td>
<td>scanf</td>
<td>tan</td>
<td>wcstod</td>
</tr>
<tr>
<td>cosh</td>
<td>fscanf</td>
<td>localtime</td>
<td>setbuf</td>
<td>tanh</td>
<td>wcstok</td>
</tr>
<tr>
<td>ctime</td>
<td>getc</td>
<td>log</td>
<td>setlocale</td>
<td>time</td>
<td>wcstol</td>
</tr>
<tr>
<td>difftime</td>
<td>getchar</td>
<td>log10</td>
<td>setvbuf</td>
<td>tmpfile</td>
<td>wcsctombs</td>
</tr>
<tr>
<td>div</td>
<td>getenv</td>
<td>longjmp</td>
<td>signal</td>
<td>tmpnam</td>
<td>wcstoul</td>
</tr>
<tr>
<td>exit</td>
<td>gets</td>
<td>malloc</td>
<td>sin</td>
<td>tolower</td>
<td>wcxfreq</td>
</tr>
<tr>
<td>exp</td>
<td>getwc</td>
<td>mblen</td>
<td>sinh</td>
<td>toupper</td>
<td>wctob</td>
</tr>
<tr>
<td>fabs</td>
<td>getwchar</td>
<td>mbrien</td>
<td>sprintf</td>
<td>towctrans</td>
<td>wctomb</td>
</tr>
<tr>
<td>fclose</td>
<td>gmtime</td>
<td>mbtowc</td>
<td>sqrt</td>
<td>tolower</td>
<td>wcstrans</td>
</tr>
<tr>
<td>feof</td>
<td>isalnum</td>
<td>mbstinit</td>
<td>srand</td>
<td>towupper</td>
<td>wctype</td>
</tr>
<tr>
<td>ferror</td>
<td>isalpha</td>
<td>mbsttowcs</td>
<td>sscanf</td>
<td>ungetc</td>
<td>wmemchr</td>
</tr>
<tr>
<td>fflush</td>
<td>iscntrl</td>
<td>mbstowcs</td>
<td>strcat</td>
<td>ungetwc</td>
<td>wmemcmp</td>
</tr>
<tr>
<td>fgetc</td>
<td>isdigit</td>
<td>mbtowc</td>
<td>strchr</td>
<td>vfprintf</td>
<td>wmemncpy</td>
</tr>
<tr>
<td>fgetpos</td>
<td>isgraph</td>
<td>memchr</td>
<td>strcmp</td>
<td>vfprintf</td>
<td>wmemmove</td>
</tr>
<tr>
<td>fgets</td>
<td>islower</td>
<td>memchr</td>
<td>strcoll</td>
<td>vprintf</td>
<td>wmemset</td>
</tr>
<tr>
<td>fgetwc</td>
<td>isprint</td>
<td>memcmp</td>
<td>strcpyp</td>
<td>vsprintf</td>
<td>wprintf</td>
</tr>
<tr>
<td>fgetws</td>
<td>ispunct</td>
<td>memmove</td>
<td>strcspn</td>
<td>vswprintf</td>
<td>wscanf</td>
</tr>
<tr>
<td>floor</td>
<td>isspace</td>
<td>memset</td>
<td>strerror</td>
<td>vfprintf</td>
<td>wscanf</td>
</tr>
</tbody>
</table>
C.2.1 Modifications to headers

For compatibility with the Standard C library, the C++ standard library provides the 18 C headers (D.6), but their use is deprecated in C++.

C.2.2 Modifications to definitions

C.2.2.1 Types char16_t and char32_t

The types char16_t and char32_t are distinct types rather than typedefs to existing integral types.

C.2.2.2 Type wchar_t

wchar_t is a keyword in this International Standard (2.12). It does not appear as a type name defined in any of <cstdio>, <cstdlib>, or <cwchar> (21.7).

C.2.2.3 Header <iso646.h>

The tokens and, and_eq, bitand, bitor, compl, not_eq, not, or, or_eq, xor, and xor_eq are keywords in this International Standard (2.12). They do not appear as macro names defined in <ciso646>.

C.2.2.4 Macro NULL

The macro NULL, defined in any of <clocale>, <cstdlib>, <cstdio>, <cstdlib>, <cstring>, <ctime>, or <cwchar>, is an implementation-defined C++ null pointer constant in this International Standard (18.2).

C.2.3 Modifications to declarations

Header <cstring>: The following functions have different declarations:

- strchr
- strpbrk
- strrchr
- strstr
- memchr

21.7 describes the changes.

C.2.4 Modifications to behavior

Header <cstdlib>: The following functions have different behavior:

- atexit
- exit
- abort

18.5 describes the changes.

Header <csetjmp>: The following functions have different behavior:

- longjmp

18.10 describes the changes.
C.2.4.1 Macro `offsetof(type,member-designator)`

The macro `offsetof`, defined in `<cstddef>`, accepts a restricted set of `type` arguments in this International Standard. 18.2 describes the change.

C.2.4.2 Memory allocation functions

The functions `calloc`, `malloc`, and `realloc` are restricted in this International Standard. 20.9.14 describes the changes.
Annex 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 3.3.6).

D.3 Access declarations  

Access declarations are deprecated (see 11.3).

D.4 register keyword  

The use of the register keyword as a storage-class-specifier (7.1.1) is deprecated.

D.5 Dynamic exception specifications  

The use of dynamic-exception-specifications is deprecated.

D.6 C standard library headers  

For compatibility with the C standard library and the C Unicode TR, the C++ standard library provides the 25 C headers, as shown in Table 151.

| `<assert.h>` | `<float.h>` | `<math.h>` | `<stddef.h>` | `<tgmath.h>` |
| `<complex.h>` | `<inttypes.h>` | `<setjmp.h>` | `<stdio.h>` | `<time.h>` |
| `<ctype.h>` | `<iso646.h>` | `<signal.h>` | `<stdint.h>` | `<uchar.h>` |
| `<errno.h>` | `<limits.h>` | `<stdarg.h>` | `<stdlib.h>` | `<wchar.h>` |
| `<fenv.h>` | `<locale.h>` | `<stdbool.h>` | `<string.h>` | `<wctype.h>` |

Every C header, each of which has a name of the form name.h, behaves as if each name placed in the standard library namespace by the corresponding name header is placed within the global namespace scope. It is unspecified whether these names are first declared or defined within namespace scope (3.3.6) of the namespace std and are then injected into the global namespace scope by explicit using-declarations (7.3.3).

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.7 Old iostreams members

The following member names are in addition to names specified in Clause 27:

```cpp
namespace std {
  class ios_base {
    public:
      typedef T1 io_state;
      typedef T2 open_mode;
      typedef T3 seek_dir;
      typedef 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` (27.5.1).

6. The type `streampos` is an implementation-defined type that satisfies the requirements of type `POS_T` (27.3).

7. An implementation may provide the following additional member function, which has the effect of calling `sbumpc()` (27.6.2.2.3):

```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:

```cpp
namespace std {
  template<class charT, class traits> class basic_ios {
    public:
      void clear(io_state state);
      void setstate(io_state state);
      void exceptions(io_state);
      // remainder unchanged
  };
  class ios_base {
```

§ D.7
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
    }
}

9 The effects of these functions is to call the corresponding member function specified in Clause 27.

D.8 char* streams

1 The header `<strstream>` defines three types that associate stream buffers with character array objects and assist reading and writing such objects.

D.8.1 Class strstreambuf

namespace std {
    class strstreambuf : public basic_streambuf<char> {
        public:
            explicit strstreambuf(streamsize alsize_arg = 0);
            strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
            strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = 0);
            strstreambuf(const char* gnext_arg, streamsize n);
            strstreambuf(signed char* gnext_arg, streamsize n, 
                signed char* pbeg_arg = 0);
            // remainder unchanged
    }
The class `strstreambuf` associates the input sequence, and possibly the output sequence, with an object of some character array type, whose elements store arbitrary values. The array object has several attributes.

[Note: For the sake of exposition, these are represented as elements of a bitmask type (indicated here as T1) called `strstate`. The elements are:

- `allocated`, set when a dynamic array object has been allocated, and hence should be freed by the destructor for the `strstreambuf` object;
- `constant`, set when the array object has `const` elements, so the output sequence cannot be written;
- `dynamic`, set when the array object is allocated (or reallocated) as necessary to hold a character sequence that can change in length;
- `frozen`, set when the program has requested that the array object not be altered, reallocated, or freed.

— end note]

[Note: For the sake of exposition, the maintained data is presented here as:

- `strstate strmode`, the attributes of the array object associated with the `strstreambuf` object;
- `int 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]

Each object of class `strstreambuf` has a *seekable area*, delimited by the pointers `seeklow` and `seekhigh`. If `gnext` is a null pointer, the seekable area is undefined. Otherwise, `seeklow` equals `gbeg` and `seekhigh` is either `pend`, if `pend` is not a null pointer, or `gend`.

D.8.1.1 `strstreambuf` constructors

```plaintext
explicit strstreambuf(streamsize alsize_arg = 0);
```

1. **Effects:** Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 152.

<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>

```plaintext
strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
```

2. **Effects:** Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 153.

<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>

```plaintext
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);
```

3. **Effects:** Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 154.

`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)`.
Table 154 — `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>alsiz</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>

— If \( n < 0 \), \( N \) is `INT_MAX`.\(^{343}\)

5 If `pbeg_arg` is a null pointer, the function executes:

```c
setg(gnext_arg, gnext_arg, gnext_arg + N);
```

6 Otherwise, the function executes:

```c
setg(gnext_arg, gnext_arg, pbeg_arg);
setp(pbeg_arg, pbeg_arg + N);
```

```c
strstreambuf(const char* gnext_arg, streamsize n);
strstreambuf(const signed char* gnext_arg, streamsize n);
strstreambuf(const unsigned char* gnext_arg, streamsize n);
```

7 **Effects:** Behaves the same as `strstreambuf((char*)gnext_arg,n)`, except that the constructor also sets `constant` in `strmode`.

```c
virtual ~strstreambuf();
```

8 **Effects:** Destroys an object of class `strstreambuf`. The function frees the dynamically allocated array object only if `strmode & allocated != 0` and `strmode & frozen == 0`. (D.8.1.3 describes how a dynamically allocated array object is freed.)

### D.8.1.2 Member functions

[depr.strstreambuf.members]

```c
void freeze(bool freezefl = true);
```

1 **Effects:** If `strmode & dynamic` is non-zero, alters the freeze status of the dynamic array object as follows:

— If `freezefl` is `true`, the function sets `frozen` in `strmode`.

— Otherwise, it clears `frozen` in `strmode`.

```c
char* str();
```

2 **Effects:** Calls `freeze()`, then returns the beginning pointer for the input sequence, `gbeg`.

3 **Remarks:** The return value can be a null pointer.

```c
int pcount() const;
```

4 **Effects:** If the next pointer for the output sequence, `pnext`, is a null pointer, returns zero. Otherwise, returns the current effective length of the array object as the next pointer minus the beginning pointer for the output sequence, `pnext - pbeg`.

\(^{343}\) The function signature `strlen(const char*)` is declared in `<cstring>`. (21.7). The macro `INT_MAX` is defined in `<climits>` (18.3).
D.8.1.3 strstreambuf overridden virtual functions

`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. 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 (reallocating it with greater length) to make a write position available.

`int_type pbackfail(int_type c = EOF);`

**Puts back the character designated by `c` to the input sequence, if possible, in one of three ways:**

- If `c != 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.

**Remarks:** If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

`int_type underflow();`

---

344) An implementation should consider `alsize` in making this decision.
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 (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 (unsigned char*)gnext.

Remarks: The function can alter the number of read positions available as a result of any call.

Returns EOF to indicate failure.

pos_type seekoff(off_type off, seekdir way, openmode which = in | out);

Effects: Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 155.

Table 155 — 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 156.

Table 156 — 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.8.2 Class istrstream

namespace std {
    class istrstream : public basic_istream<char> {
        public:
            explicit istrstream(const char* s);
            explicit istrstream(char* s);
            istrstream(const char* s, streamsize n);
            istrstream(char* s, streamsize n);
            virtual ~istrstream();

            strstreambuf* rdbuf() const;
            char *str();
        private:
            strstreambuf sb; // exposition only
    };
}

The class istrstream supports the reading of objects of class strstreambuf. It supplies a strstreambuf
object to control the associated array object. For the sake of exposition, the maintained data is presented
here as:

— sb, the strstreambuf object.

D.8.2.1 istrstream constructors

explicit istrstream(const char* s);
explicit istrstream(char* s);
Effects: Constructs an object of class `istrstream`, initializing the base class with `istream(&sb)` and initializing `sb` with `strstreambuf(s,0)`.

```
istrstream(const char* s, streamsize n);
```

Effects: Constructs an object of class `istrstream`, initializing the base class with `istream(&sb)` and initializing `sb` with `strstreambuf(s,n)`.

`s` shall designate the first element of an `ntbs`.

**D.8.2.2 Member functions**

```
strstreambuf* rdbuf() const;
char* str();
```

`rdbuf()` returns the `strstreambuf` object.

`str()` returns a pointer to the underlying `strstreambuf`.

**D.8.3 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.8.3.1 ostrstream constructors**

```
ostrstream();
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 `strstreambuf()`.

Effects: Constructs an object of class `ostrstream`, initializing the base class with `ostream(&sb)`, and initializing `sb` with one of two constructors:
— If \((\text{mode} \& \text{app}) == 0\), then \(s\) shall designate the first element of an array of \(n\) elements.

The constructor is \(\text{strstreambuf}(s, n, s)\).

— If \((\text{mode} \& \text{app}) != 0\), then \(s\) shall designate the first element of an array of \(n\) elements that contains an \(\text{NTBS}\) whose first element is designated by \(s\). The constructor is \(\text{strstreambuf}(s, n, s + \text{std::strlen}(s))\).  

\[\text{D.8.3.2 Member functions} \ [\text{depr.ostrstream.members}]\]

\[
\text{strstreambuf* rdbuf() const;}
\]

\(\text{Returns: (strstreambuf*)}&sb .\)

\[
\text{void freeze(bool freezefl = true);}
\]

\(\text{Effects: Calls rdbuf()->freeze(freezefl).}\)

\[
\text{char* str();}
\]

\(\text{Returns: rdbuf()->str().}\)

\[
\text{int pcount() const;}
\]

\(\text{Returns: rdbuf()->pcount().}\)

\[\text{D.8.4 Class strstream} \ [\text{depr.strstream}]\]

\[
\text{namespace std \{}
\]

\[
\text{class strstream}
\]

\[
:\text{public basic_iostream<char> \{}
\]

\[
\text{public:}
\]

\[
\text{// Types}
\]

\[
\text{typedef char char_type;}
\]

\[
\text{typedef typename char_traits<char>::int_type int_type;}
\]

\[
\text{typedef typename char_traits<char>::pos_type pos_type;}
\]

\[
\text{typedef typename char_traits<char>::off_type off_type;}
\]

\[
\text{// constructors/destructor}
\]

\[
\text{strstream();}
\]

\[
\text{strstream(char* s, int n,}
\]

\[
\text{ios_base::openmode mode = ios_base::in|ios_base::out);}
\]

\[
\text{virtual ~strstream();}
\]

\[
\text{// Members:}
\]

\[
\text{strstreambuf* rdbuf() const;}
\]

\[
\text{void freeze(bool freezefl = true);}
\]

\[
\text{int pcount() const;}
\]

\[
\text{char* str();}
\]

\[
\text{private:}
\]

\[
\text{strstreambuf sb; // exposition only}
\]

\[
\}\}
\]

345) The function signature \(\text{strlen(const char*)}\) is declared in \(<\text{cstring}\>\ (21.7).
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.8.4.1 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.8.4.2 strstream destructor

```cpp
virtual ~strstream();
```

**Effects:** Destroys an object of class `strstream`.

```cpp
strstreambuf* rdbuf() const;
```

**Returns:** `&sb`.

D.8.4.3 strstream operations

```cpp
void freeze(bool freeze1 = true);
```

**Effects:** Calls `rdbuf()->freeze(freeze1)`.

```cpp
char* str();
```

**Returns:** `rdbuf()->str()`.

```cpp
int pcount() const;
```

**Returns:** `rdbuf()->pcount()`.

D.9 Binders

The binders `binder1st, bind1st, binder2nd, and bind2nd` are deprecated. [Note: The function template `bind` (20.8.10) provides a better solution. — end note]

D.9.1 Class template binder1st

§ D.9.1
template <class Fn>
class binder1st
  : public unary_function<
    typename Fn::second_argument_type,
    typename Fn::result_type>
  {
protected:
  Fn* op;
  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 op with x and value with y.
2 operator() returns op(value,x).

D.9.2 bind1st
[depr.lib.bind.1st]

template <class Fn, class T>
binder1st<Fn> bind1st(const Fn& fn, const T& x);

1 Returns: binder1st<Fn>(fn, typename Fn::first_argument_type(x)).

D.9.3 Class template binder2nd
[depr.lib.binder.2nd]

template <class Fn>
class binder2nd
  : public unary_function<
    typename Fn::first_argument_type,
    typename Fn::result_type>
  {
protected:
  Fn* op;
  typename Fn::second_argument_type value;
public:
  binder2nd(const Fn& x,
            const typename Fn::second_argument_type& y);
  typename Fn::result_type
  operator()(const typename Fn::first_argument_type& x) const;
  typename Fn::result_type
  operator()(typename Fn::first_argument_type& x) const;
};

1 The constructor initializes op with x and value with y.
2 operator() returns op(x,value).

D.9.4 bind2nd
[depr.lib.bind.2nd]

template <class Fn, class T>
binder2nd<Fn> bind2nd(const Fn& op, const T& x);

1 Returns: binder2nd<Fn>(op, typename Fn::second_argument_type(x)).
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(), bind1st(greater<int>(), 5));
```

finds the first integer in v less than 5. — end example

D.10 auto_ptr

The class template `auto_ptr` is deprecated. [Note: The class template `unique_ptr` (20.9.10) provides a better solution. — end note]

D.10.1 Class template auto_ptr

The class 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).

The class template `auto_ptr_ref` is for exposition only. An implementation is permitted to provide equivalent functionality without providing a template with this name. The template 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; // exposition only

template <class X> class auto_ptr {
    public:
        typedef X element_type;

        // D.10.1.1 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();
        auto_ptr& operator=(auto_ptr_ref<X> r) throw();
        ~auto_ptr() throw();

        // D.10.1.2 members:
        X& operator*() const throw();
        X* operator->() const throw();
        X* get() const throw();
        X* release() throw();
        void reset(X* p =0) throw();

        // D.10.1.3 conversions:
        auto_ptr(auto_ptr_ref<X>) throw();
        template<class Y> operator auto_ptr_ref<Y>() throw();
        template<class Y> operator auto_ptr<Y>() throw();
    };

template <> class auto_ptr<void>
{
}
```

§ D.10.1
The class template \texttt{auto\_ptr} provides a semantics of strict ownership. An \texttt{auto\_ptr} owns the object it holds a pointer to. Copying an \texttt{auto\_ptr} copies the pointer and transfers ownership to the destination. If more than one \texttt{auto\_ptr} owns the same object at the same time the behavior of the program is undefined. [\textit{Note}: The uses of \texttt{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. Instances of \texttt{auto\_ptr} meet the requirements of \texttt{MoveConstructible} and \texttt{MoveAssignable}, but do not meet the requirements of \texttt{CopyConstructible} and \texttt{CopyAssignable}. — end note]}

\textbf{D.10.1.1 auto\_ptr constructors} \hfill [auto.ptr.cons]

\begin{verbatim}
explicit auto_ptr(X* p = 0) throw();
   // Postconditions: *this holds the pointer p.

auto_ptr(auto_ptr& a) throw();
   // Effects: Calls a.release().
   // Postconditions: *this holds the pointer returned from a.release().

template<class Y> auto_ptr(auto_ptr<Y>& a) throw();
   // Requires: Y* can be implicitly converted to X*.
   // Effects: Calls a.release().
   // Postconditions: *this holds the pointer returned from a.release().

auto_ptr& operator=(auto_ptr& a) throw();
   // Requires: The expression delete get() is well formed.
   // Effects: reset(a.release()).
   // Returns: *this.

template<class Y> auto_ptr& operator=(auto_ptr<Y>& a) throw();
   // Requires: Y* can be implicitly converted to X*.
   // The expression delete get() is well formed.
   // Effects: reset(a.release()).
   // Returns: *this.

~auto_ptr() throw();
   // Requires: The expression delete get() is well formed.
   // Effects: delete get().
\end{verbatim}

§ D.10.1.1
D.10.1.2  auto_ptr members

\[ \text{auto_ptr.members} \]

\[ \text{X& operator*() const throw();} \]
1  \[ Requires: \text{get()} \neq 0 \]
2  \[ Returns: \text{*get()} \]

\[ \text{X* operator->() const throw();} \]
3  \[ Returns: \text{get()} \]

\[ \text{X* get() const throw();} \]
4  \[ Returns: The pointer *this holds. \]

\[ \text{X* release() throw();} \]
5  \[ Returns: \text{get()} \]
6  \[ Postcondition: *this holds the null pointer. \]

\[ \text{void reset(X* p=0) throw();} \]
7  \[ Effects: If get() \neq p then delete get(). \]
8  \[ Postconditions: *this holds the pointer p. \]

D.10.1.3  auto_ptr conversions

\[ \text{auto_ptr(auto_ptr_ref<X> r) throw();} \]
1  \[ Effects: Calls p.release() for the auto_ptr p that r holds. \]
2  \[ Postconditions: *this holds the pointer returned from release(). \]

\[ \text{template<class Y> operator auto_ptr_ref<Y>() throw();} \]
3  \[ Returns: An auto_ptr_ref<Y> that holds *this. \]

\[ \text{template<class Y> operator auto_ptr<Y>() throw();} \]
4  \[ Effects: Calls release(). \]
5  \[ Returns: An auto_ptr<Y> that holds the pointer returned from release(). \]

\[ \text{auto_ptr& operator=(auto_ptr_ref<X> r) throw()} \]
6  \[ Effects: Calls reset(p.release()) for the auto_ptr p that r holds a reference to. \]
7  \[ Returns: *this \]
Annex E  (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 18.3.1.

A
accumulate 26.7.1
adjacent.difference 26.7.4
adjustfield.manip 27.5.5.2
alg.adjacent.find 25.2.8
alg.all_of 25.2.1
alg.any_of 25.2.2
alg.binary.search 25.4.3
alg.c.library 25.5
alg.copy 25.3.1
alg.count 25.2.9
alg.equal 25.2.11
alg.fill 25.3.6
alg.find 25.2.5
alg.find.end 25.2.6
alg.find.first.of 25.2.7
alg.foreach 25.2.4
alg.generate 25.3.7
alg.heap.operations 25.4.6
alg.is_permutation 25.2.12
alg.lex.comparison 25.4.8
alg.merge 25.4.4
alg.min.max 25.4.7
alg.modifying.operations 25.3
alg.move 25.3.2
alg.none_of 25.2.3
alg.nonmodifying 25.2
alg.nth.element 25.4.2
alg.partitions 25.3.13
alg.permutation.generators 25.4.9
alg.random.shuffle 25.3.12
alg.remove 25.3.8
alg.replace 25.3.5
alg.reverse 25.3.10
alg.rotate 25.3.11
alg.search 25.2.13
alg.set.operations 25.4.5
alg.sort 25.4.1
alg.sorting 25.4
alg.swap 25.3.3
alg.transform 25.3.4
alg.unique 25.3.9
algorithms 25
algorithms.general 25.1
alloc.errors 18.6.2
allocator.adaptor 20.9.6
allocator.adaptor.cnstr 20.9.6.2
allocator.adaptor.members 20.9.6.3
allocator.adaptor.types 20.9.6.1
allocator.globals 20.9.5.2
allocator.members 20.9.5.1
allocator.requirements 20.2.5
allocator.tag 20.9.1
allocator.traits 20.9.4
allocator.traits.members 20.9.4.2
allocator.traits.types 20.9.4.1
allocator.uses 20.9.2
allocator.uses.construction 20.9.2.2
allocator.uses.trait 20.9.2.1
alt.headers 17.6.3.4
arithmetic.operations 20.8.5
array 23.3.1
array.cons 23.3.1.1
array.data 23.3.1.4
array.fill 23.3.1.5
array.size 23.3.1.3
array.special 23.3.1.2
array.swap 23.3.1.6
array.tuple 23.3.1.8
array.zero 23.3.1.7
assertions 19.3
associative 23.4
associative.reqmts 23.2.4
associative.reqmts.except 23.2.4.1
atomics 29
atomics.fences 29.8
atomics.flag 29.7
atomics.general 29.1
atomics.lockfree 29.4
atomics.order 29.3
atomics.syn 29.2
atomics.types 29.5
atomics.types.address 29.5.2
atomics.types.generic 29.5.3
atomics.types.integral 29.5.1
atomics.types.operations 29.6
auto.ptr D.10.1
auto.ptr.cons D.10.1.1
auto.ptr.conv D.10.1.3
auto.ptr.members D.10.1.2

B
back.insert.iter.cons 24.5.2.2.1
back.insert.iter.op* 24.5.2.2.3
back.insert.iter.op++ 24.5.2.2.4
back.insert.iter.op= 24.5.2.2.2
back.insert.iter.ops 24.5.2.2
back.insert.iterator 24.5.2.1
back.inserter 24.5.2.2.5
bad.alloc 18.6.2.1
bad.cast 18.7.2
bad.exception 18.8.2.1
bad.typeid 18.7.3
base 20.8.3
basefield.manip 27.5.5.3
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 3.3.5
basic.ios.cons 27.5.4.1
basic.ios.members 27.5.4.2
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 3.3.7
basic.scope.declarative 3.3.1
basic.scope.enum 3.3.8
basic.scope.hiding 3.3.10
basic.scope.local 3.3.3
basic.scope.namespace 3.3.6
basic.scope.pdecl 3.3.2
basic.scope.proto 3.3.4
basic.scope.temp 3.3.9
basic.start 3.6
basic.start.init 3.6.2
basic.start.main 3.6.1
basic.start.term 3.6.3
basic.stc 3.7
basic.stc.auto 3.7.3
basic.stc.dynamic 3.7.4
basic.stc.dynamic.allocation 3.7.4.1
basic.stc.dynamic.deallocation 3.7.4.2
basic.stc.dynamic.safety 3.7.4.3
basic.stc.inherit 3.7.5
basic.stc.static 3.7.1
basic.stc.thread 3.7.2
basic.string 21.4
basic.string.hash 21.6
basic.type.qualifier 3.9.3
basic.types 3.9
bidirectional.iterators 24.2.6
binary.search 25.4.3.4
bind 20.8.10
C

c.files 27.9.2
c.limits 18.3.2
c.malloc 20.9.14
c.math 26.8
c.strings 21.7
category.collate 22.4.4
category.ctype 22.4.1
category.messages 22.4.7
category.monetary 22.4.6
category.numeric 22.4.2
category.time 22.4.5
cmplx 26.4.10
cfenv 26.3
cfenv.syn 26.3.1
char.traits 21.2
char.traits.require 21.2.1
char.traits.specializations 21.2.3
char.traits.specializations.char 21.2.3.1
char.traits.specializations.char16_t 21.2.3.2
char.traits.specializations.char32_t 21.2.3.3
char.traits.specializations.wchar.t 21.2.3.4
char.traits.typedefs 21.2.2
char16_t.seq 17.5.2.1.4.3
char32_t.seq 17.5.2.1.4.4
character.seq 17.5.2.1.4
class 9
class.abstract 10.4
class.access 11
class.access.base 11.2
class.access.del 11.3
class.access.nest 11.8
class.access.spec 11.1
class.access.virt 11.6
class.base.init 12.6.2
class.bit 9.6
class.constructor 12.7
class.copy 12.8
class.derived 10
class.dtor 12.4
class.expl.init 12.6.1
class.free 12.5
class.friend 11.4
class.gslice 26.6.6
class.inhctor 12.9
class.init 12.6
class.local 9.8
class.mem 9.2
class.member.lookup 10.2
class.mfct 9.3
class.mfct.non-static 9.3.1
class.mi 10.1
class.name 9.1
class.nest 9.7
class.nested.type 9.9
class.paths 11.7
class.protected 11.5
class.qual 3.4.3.1
class.slice 26.6.4
class.static 9.4
class.static.data 9.4.2
class.static.mfct 9.4.1
class.temporal 12.2
class.this 9.3.2
class.union 9.5
class.virtual 10.3
classification 22.3.3.1
cmplx.over 26.4.9
comparisons 20.8.6
complex 26.4.2
complex.member.ops 26.4.5
complex.members 26.4.4
complex.numbers 26.4
complex.ops 26.4.6
complex.special 26.4.3
complex.syn 26.4.1
complex.transcendentals 26.4.8
complex.value.ops 26.4.7
compliance 17.6.1.3
conforming 17.6.4
conforming.overview 17.6.4.1
core.slice 26.6.4.1
constexpr.functions 17.6.4.6
constraints 17.6.3
constraints.overview 17.6.3.1
container.adaptors 23.3.5
container.requirements 23.2
container.requirements.dataraces 23.2.2
container.requirements.general  23.2.1
containers   23
containers.general  23.1
contents  17.6.1.1
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  17.5.2
conversions  22.3.3.2
conversions.buffer  22.3.3.2.3
conversions.character  22.3.3.2.1
conversions.string  22.3.3.2.2
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
cpp.pragma  16.6
cpp.pragma.op  16.9
cpp.predefined  16.8
cpp.replace  16.3
cpp.rescan  16.3.4
cpp.scope  16.3.5
cpp.stringify  16.3.2
cpp.subst  16.3.1
cstdint  18.4
cstdint.syn  18.4.1
dcl.attr.grammar  7.6.1
dcl.attr.noreturn  7.6.3
dcl.attr.override  7.6.5
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.def.default  8.4.2
dcl.fct.def.delete  8.4.3
dcl.fct.def.general  8.4.1
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.list  8.5.4
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.stc  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
declval  20.3.4
default.allocator  20.9.5
definitions  17.3
defns.additional  17.4
defns.arbitrary.stream
defns.argument
defns.blocked
defns.character
defns.character.container
defns.comparison
defns.component
defns.cond.supp
defns.deadlock
defns.default.behavior
defns.diagnostic
defns.dynamic.type
defns.handler

date.time  20.11
dcl.align  7.6.2
dcl.ambig.res  8.2
dcl.array  8.3.4
dcl.asm  7.4
dcl.attr  7.6
dcl.attr.depend  7.6.6
dcl.attr.final  7.6.4
defns.ill.formed
defns.impl.defined
defns.impl.limits
defns.iostream.templates
defns.locale.specific
defns.modifier
defns.move.assign
defns.move.assign.op
defns.move.ctor
defns.ntcts
defns.obj.state
defns.observer
defns.parameter
defns.regex.collating.element
defns.regex.finite.state.machine
defns.regex.format.specifier
defns.regex.matched
defns.regex.primary.equivalence.class
defns.regex.regex.expression
defns.regex.regex.matched
defns.regex.regex.subexpression
defns.replacement
defns.repositional.stream
defns.required.behavior
defns.reserved.function
defns.static.type
defns.traits
defns.undefined
defns.unspecified
defns.well.formed
denorm.style 18.3.1.4
depr D
depr.access.dcl D.3
depr.auto.ptr D.10
depr.c.headers D.6
depr.except.spec D.5
depr.incr.bool D.1
depr.ios.members D.7
depr.istrstream D.8.2
depr.istrstream.cons D.8.2.1
depr.istrstream.members D.8.2.2
depr.lib.bind.1st D.9.2
depr.lib.bind.2nd D.9.4
depr.lib.bind.1st D.9.1
depr.lib.bind.2nd D.9.3
depr.lib.binders D.9
depr.ostrstream D.8.3
depr.ostrstream.cons D.8.3.1
depr.ostrstream.members D.8.3.2
depr.register D.4
depr.static D.2
depr.str.strstreams D.8
depr.strstream D.8.4
depr.strstream.cons D.8.4.1
depr.strstream.dest D.8.4.2
depr.strstream.oper D.8.4.3
depr.strstreambuf D.8.1
depr.strstreambuf.cons D.8.1.1
depr.strstreambuf.members D.8.1.2
depr.strstreambuf.virtuals D.8.1.3
deque 23.3.2
deque.capacity 23.3.2.2
deque.cons 23.3.2.1
deque.modifiers 23.3.2.3
deque.special 23.3.2.4
derivation 17.6.4.10
derived.classes 17.6.3.5
description 17.5
diagnostics 19
diagnostics.general 19.1
diff C
diff.basic C.1.2
diff.char16 C.2.2.1
diff.class C.1.7
diff.cpp C.1.9
diff.decl C.1.5
diff.decl C.1.6
diff.expr C.1.3
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.definitions 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 19.2.2

e
enumerated.types 17.5.2.1.2
equal.range 25.4.3.3
erro 19.4
error.reporting 27.5.5.5
intro.scope 1.1
intro.structure 1.5
invalid.argument 19.2.3
ios 27.5.4
ios.base 27.5.2
ios.base.callback 27.5.2.6
ios.base.cons 27.5.2.7
ios.baselocales 27.5.2.3
ios.base.storage 27.5.2.5
ios.members.static 27.5.2.4
ios.types 27.5.2.1
ios:failure 27.5.2.1.1
ios:fmtflags 27.5.2.1.2
ios::Init 27.5.2.1.6
ios::iostate 27.5.2.1.3
ios::openmode 27.5.2.1.4
ios::seekdir 27.5.2.1.5
iostate.flags 27.5.4.3
iostream.assign 27.7.1.5.3
iostream.cons 27.7.1.5.1
iostream.dest 27.7.1.5.2
iostream.format 27.7
iostream.forward 27.3
iostream_limits.imbue 27.2.1
iostream_objects 27.4
iostream_class 27.7.1.5
iostreams.base 27.5
iostreams Limits.pos 27.2.2
iostreams_requirements 27.2
iostreams_threadsafety 27.2.3
is_heap 25.4.6.5
is_sorted 25.4.1.5
istream 27.7.1.1
istream_assign 27.7.1.1.2
istream_cons 27.7.1.1.1
istream_formated 27.7.1.2
istream_formated_arithmetic 27.7.1.2.2
istream_formated_reqmts 27.7.1.2.1
istream_iterator 24.6.1
istream_iterator_cons 24.6.1.1
istream_iterator_ops 24.6.1.2
istream_manip 27.7.1.4
istream_rvalue 27.7.1.6
istream_unformatted 27.7.1.3
istream::extractors 27.7.1.2.3
istream::sentry 27.7.1.1.3
istreambuf_iterator 24.6.3
istreambuf_iterator_cons 24.6.3.2
istreambuf_iterator_equal 24.6.3.2
istreambuf_iterator_op_equal 24.6.3.7
istreambuf_iterator_op_star 24.6.3.3
istreambuf_iterator::op++ 24.6.3.4
istreambuf_iterator::op== 24.6.3.6
istreambuf_iterator::proxy 24.6.3.1
istreamstream 27.8.2
istreamstream_assign 27.8.2.2
istreamstream_cons 27.8.2.1
istreamstream_members 27.8.2.3
iterator_basic 24.4.2
iterator_iterators 24.2.2
iterator_operations 24.4.4
iterator_primitives 24.4
iterator_range 24.6.5
iterator_requirements 24.2
iterator_requirements_general 24.2.1
iterator_synopsis 24.3
iterator_traits 24.4.1
iterators 24
iterators_general 24.1

J

K

L

language_support 18
length_error 19.2.4
lex 2
lex_bool 2.14.6
lex_ccon 2.14.3
lex_charset 2.3
lex_comment 2.8
lex_digraph 2.6
lex_ext 2.14.8
lex_fcon 2.14.4
lex_header 2.9
lex_icon 2.14.2
lex_key 2.12
lex_literal 2.14
lex_literal_kinds 2.14.1
lex_name 2.11
lex_nullptr 2.14.7
lex_operators 2.13
lex_phases 2.2
lex_ppnumber 2.10
lex_pptoken 2.5
lex_separate 2.1
lex_string 2.14.5
lex_token 2.7
lex_trigraph 2.4
library 17
library_c 17.2
library.general  17.1
limits  18.3.1
list  23.3.4
list.capacity  23.3.4.2
list.cons  23.3.4.1
list.modifiers  23.3.4.3
list.ops  23.3.4.4
list.special  23.3.4.5
locale  22.3.1
locale.categories  22.4
locale.category  22.3.1.1.1
locale.codecvt  22.4.1.4
locale.codecvt.byname  22.4.1.5
locale.codecvt.members  22.4.1.4.1
locale.codecvt.virtuals  22.4.1.4.2
locale.collate  22.4.4.1
locale.collate.byname  22.4.4.2
locale.collate.members  22.4.4.1.1
locale.collate.virtuals  22.4.4.1.2
locale.cons  22.3.1.2
locale.convenience  22.3.3
locale.ctype  22.4.1.1
locale.ctype.byname  22.4.1.2
locale.ctype.members  22.4.1.1.1
locale.ctype.virtuals  22.4.1.1.2
locale.facet  22.3.1.1.2
locale.global.templates  22.3.2
locale.id  22.3.1.1.3
locale.members  22.3.1.3
locale.messages  22.4.7.1
locale.messages.byname  22.4.7.2
locale.messages.members  22.4.7.1.1
locale.messages.virtuals  22.4.7.1.2
locale.money.get  22.4.6.1
locale.money.get.byname  22.4.6.2
locale.money.get.members  22.4.6.1.1
locale.money.get.virtuals  22.4.6.1.2
locale.money.put  22.4.6.3
locale.money.put.byname  22.4.6.4
locale.money.put.members  22.4.6.3.1
locale.money.put.virtuals  22.4.6.3.2
locale.moneypunct  22.4.6.5
locale.moneypunct.byname  22.4.6.6
locale.moneypunct.members  22.4.6.5.1
locale.moneypunct.virtuals  22.4.6.5.2
locale.id  22.3.1.1.5
locale.operators  22.3.1.4
locale.statics  22.3.1.5
locale.stdcvt  22.5
locale.syn  22.2
locale.time.get  22.4.5.1
locale.time.get.byname  22.4.5.2
locale.time.get.members  22.4.5.1.1
locale.time.get.virtuals  22.4.5.1.2
locale.time.put  22.4.5.3
locale.time.put.byname  22.4.5.4
locale.time.put.members  22.4.5.3.1
locale.time.put.virtuals  22.4.5.3.2
locale.types  22.3.1.1
locales  22.3
localization  22
localization.general  22.1
logic.error  19.2.1
logical.operations  20.8.7
lowerbound  25.4.3.1

M
macro.names  17.6.3.3.1
make.heap  25.4.6.3
map  23.4.1
map.access  23.4.1.2
map.cons  23.4.1.1
map.modifiers  23.4.1.3
map.ops  23.4.1.4
map.special  23.4.1.5
mask.array.assign  26.6.8.1
mask.array.comp.assign  26.6.8.2
mask.array.fill  26.6.8.3
member.functions  17.6.4.5
member.pointer.adaptors  20.8.12
memory  20.9
meta  20.7
meta.help  20.7.3
meta.rel  20.7.5
meta.rqnts  20.7.1
meta.trans  20.7.6
meta.trans.arr  20.7.6.4
meta.trans.cv  20.7.6.1
meta.trans.other  20.7.6.6
meta.trans.ptr  20.7.6.5
meta.trans.ref  20.7.6.2
meta.trans.sign  20.7.6.3
meta.type.synop  20.7.2
meta.unity  20.7.4
meta.unity.cat  20.7.4.1
meta.unity.com  20.7.4.2
meta.unity.prop  20.7.4.3
mismatch  25.2.10
move.iter.nonmember  24.5.3.3.14
move.iter.op.+  24.5.3.3.8
move.iter.op.+= 24.5.3.3.9
move.iter.op.- 24.5.3.3.10
move.iter.op.-= 24.5.3.3.11
move.iter.op.comp 24.5.3.3.13
move.iter.op.const 24.5.3.3.1
move.iter.op.conv 24.5.3.3.3
move.iter.op.decr 24.5.3.3.7
move.iter.op.incr 24.5.3.3.6
move.iter.op.index 24.5.3.3.12
move.iter.op.ref 24.5.3.3.5
move.iter.op.star 24.5.3.3.4
move.iter.op.= 24.5.3.3.2
move.iter.ops 24.5.3.3
move.iter.requirements 24.5.3.2
move.iterator 24.5.3.1
move.iterators 24.5.3
multibyte.strings 17.5.2.1.4.2
multimap 23.4.2
multimap.cons 23.4.2.1
multimap.modifiers 23.4.2.2
multimap.ops 23.4.2.3
multimap.special 23.4.2.4
multiset 23.4.4
multiset.cons 23.4.4.1
multiset.special 23.4.4.2

namespace.alias 7.3.2
namespace.constraints 17.6.3.2
namespace.def 7.3.1
namespace.memdef 7.3.1.2
namespace.posix 17.6.3.2.2
namespace.qual 3.4.3.2
namespace.std 17.6.3.2.1
namespace.udecl 7.3.3
namespace.udir 7.3.4
namespace.unnamed 7.3.1.1
narrow.stream.objects 27.4.1
negators 20.8.9
new.badlength 18.6.2.2
new.delete 18.6.1
new.delete.array 18.6.1.2
new.delete.data races 18.6.1.4
new.delete.placement 18.6.1.3
new.delete.single 18.6.1.1
new.handler 18.6.2.3
nullablepointer.requirements 20.2.3
numarray 26.6
numeric.iota 26.7.5
numeric.limits 18.3.1.1
numeric.limits.members 18.3.1.2
numeric.ops 26.7
numeric.requirements 26.2
numeric.special 18.3.1.5
numers 26
numers.general 26.1

O
objects.within.classes 17.5.2.3
ofstream 27.9.1.10
ofstream.assign 27.9.1.12
ofstream.cons 27.9.1.11
ofstream.members 27.9.1.13
operators 20.3.1
organization 17.6.1
ostream 27.7.2.1
ostream.assign 27.7.2.3
ostream.cons 27.7.2.2
ostream.formatted 27.7.2.6
ostream.formatted.reqmts 27.7.2.6.1
ostream.inserters 27.7.2.6.3
ostream.inserters.arithmetic 27.7.2.6.2
ostream.inserters.character 27.7.2.6.4
ostream.iterator 24.6.2
ostream.iterator.cons.des 24.6.2.1
ostream.iterator.ops 24.6.2.2
ostream.manip 27.7.2.8
ostream.rvalue 27.7.2.9
ostream.seeks 27.7.2.5
ostream.unformatted 27.7.2.7
ostream::sentry 27.7.2.4
ostreambuf.iter.cons 24.6.4.1
ostreambuf.iter.ops 24.6.4.2
ostreambuf.iterator 24.6.4
ostreamstream 27.8.3
ostreamstream.assign 27.8.3.2
ostreamstream.cons 27.8.3.1
ostreamstream.members 27.8.3.3
out.of.range 19.2.5
output.iterators 24.2.4
output.streams 27.7.2
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
string::copy 21.4.6.7
string::erase 21.4.6.5
string::find 21.4.7.2
string::find.first.not.of 21.4.7.6
string::find.first.of 21.4.7.4
string::find.last.not.of 21.4.7.7
string::find.last.of 21.4.7.5
string::insert 21.4.6.4
string::op!= 21.4.8.3
string::op+ 21.4.8.1
string::op+= 21.4.6.1
string::op< 21.4.8.4
string::op<= 21.4.8.6
string::op> 21.4.8.5
string::op>= 21.4.8.7
string::operator== 21.4.8.2
string::replace 21.4.6.6
string::rfind 21.4.7.3
string::substr 21.4.7.8
string::swap 21.4.6.8
stringbuf 27.8.1
stringbuf.assign 27.8.1.2
stringbuf.cons 27.8.1.1
stringbuf.members 27.8.1.3
stringbuf.virtuals 27.8.1.4
strings 21
strings.general 21.1
stringstream 27.8.4
stringstream.assign 27.8.5.1
stringstream.cons 27.8.5
stringstream.members 27.8.6
structure 17.5.1
structure.elements 17.5.1.1
structure.requirements 17.5.1.3
structure.see.also 17.5.1.5
structure.specifications 17.5.1.4
structure.summary 17.5.1.2
support.dynamic 18.6
support.exception 18.8
support.general 18.1
support.initlist 18.9
support.initlist.access 18.9.2
support.initlist.cons 18.9.1
support.initlist.range 18.9.3
support.limits 18.3
support.rtti 18.7
support.runtime 18.10
support.start.term 18.5
support.types 18.2
swappable.requirements 20.2.2
syntax 1.6

syserr 19.5
syserr.compare 19.5.4
syserr.errcat 19.5.1
syserr.errcat.derived 19.5.1.4
syserr.errcat.nonvirtuals 19.5.1.3
syserr.errcat.objects 19.5.1.5
syserr.errcat.overview 19.5.1.1
syserr.errcat.virtuals 19.5.1.2
syserr.errcode 19.5.2
syserr.errcode.constructors 19.5.2.2
syserr.errcode.modifiers 19.5.2.3
syserr.errcode.nonmembers 19.5.2.5
syserr.errcode.observers 19.5.2.4
syserr.errcode.overview 19.5.2.1
syserr.errcondition 19.5.3
syserr.errcondition.constructors 19.5.3.2
syserr.errcondition.modifiers 19.5.3.3
syserr.errcondition.nonmembers 19.5.3.5
syserr.errcondition.observers 19.5.3.4
syserr.errcondition.overview 19.5.3.1
syserr.hash 19.5.5
syserr.syserr 19.5.6
syserr.syserr.members 19.5.6.2
syserr.syserr.overview 19.5.6.1

T
temp 14
temp.alias 14.5.7
temp.arg 14.3
temp.arg.explicit 14.8.1
temp.arg.nontype 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
V
valarray.access 26.6.2.3
valarray.assign 26.6.2.2
valarray.binary 26.6.3.1
valarray.cassign 26.6.2.6
valarray.comparison 26.6.3.2
valarray.cons 26.6.2.1
valarray.members 26.6.2.7
valarray.nonmembers 26.6.3
valarray.range 26.6.10
valarray.special 26.6.3.4
valarray.sub 26.6.2.4
valarray.syn 26.6.1
valarray.transcend 26.6.3.3
valarray.unary 26.6.2.5
value.error.codes 17.6.4.13
vector 23.3.6
vector.bool 23.3.7
vector.capacity 23.3.6.2
vector.cons 23.3.6.1
vector.data 23.3.6.3
vector.modifiers 23.3.6.4
vector.special 23.3.6.5

W
wide.characters 17.5.2.1.4.5
wide.stream.objects 27.4.2

X
xref E

Y

Z
Index

! , see logical negation operator
!=, see inequality operator
(), see function call operator
function declarator, 185
*, see indirection operator, see multiplication operator
pointer declarator, 180
+, 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 class member access operator
.., see class member access operator
.*, see pointer to member operator
..., see ellipsis
/, see division operator
:
field declaration, 222
label specifier, 125
::, see scope resolution operator
::*
pointer to member declarator, 183
<, see less than operator
template and, 318, 319
<=, see less than or equal to operator
<<, see left shift operator
=, see assignment operator
==, see equality operator
>, see greater than operator
>=, see greater than or equal operator
>>, see right shift operator
?:, see conditional expression operator
[], see subscripting operator
array declarator, 184
#define, 402
#else, 400
#error, 407
#elseif, 399
#endif, 400
#endif, 402
#include, 400, 423
#line, 406
#pragma, 407
#undef, 404, 424
%, see modulus operator
& , see address-of operator, see bitwise AND operator
reference declarator, 181
&&, see logical AND operator
^, see bitwise exclusive OR operator
_ _ DATE _ _, 408
_ _ FILE _ _, 408
_ _ LINE _ _, 408
_ _ STDC _ _, 408
implementation-defined, 408
_ _ STDC_HOSTED _ _, 408
implementation-defined, 408
_ _ STDC_ISO_10646 _ _, 408
implementation-defined, 408
_ _ STDC_MB_MIGHT_NEQ_WC _ _, 408
implementation-defined, 408
_ _ STDC_VERSION _ _, 408
implementation-defined, 408
_ _ TIME _ _, 408
_ _cplusplus, 407
\, see backslash
## operator, 403
# operator, 403
const object
undefined change to, 145
exception::what message
implementation-defined, 454
friend function
nested class, 224
delete, 111
{}
block statement, 125
class declaration, 210
class definition, 210
enum declaration, 150
initializer list, 199
~, see destructor
_, see character, underscore
~, see one's complement operator
|, see bitwise inclusive OR operator
||, see logical OR operator
0, see also zero, null
null character, 28
string terminator, 28

abort, 61, 131
abstract-declarator, 177, 1192
access
  union default member, 210
  adjusting base class member, 244
  base class, 242
  base class member, 226
  class member, 95
  member name, 239
  overloading and, 286
  virtual function, 249
access-specifier, 226, 1194
access control, 239
  anonymous union, 222
  member function and, 251
  overloading resolution and, 230
access specifier, 241, 242
actual argument, 2
actual parameter, 2
addition operator, 114
additive-expression, 114, 1185
address, 73, 117
address of member function
  unspecified, 429
aggregate, 199
aggregate initialization, 199
algorithm
  stable, 413
alias, 157
alias-declaration, 135, 1187
alignment
  extended, 76
  fundamental, 76
alignment requirement
  implementation-defined, 76
allocation
  alignment storage, 108
  implementation defined bit-field, 222
  unspecified, 215
allocation functions, 63
ambiguity
  base class member, 229
  class conversion, 232
  declaration type, 137
  declaration versus cast, 178
  declaration versus expression, 133
  function declaration, 197
  member access, 229
  parentheses and, 107
ambiguity detection
  overloaded function, 286
Amendment 1, 425
and-expression, 418, 1185
appertain, 170
argc, 58
argument, 2, 427–429, 463
  access checking and default, 240
  binding of default, 190
  evaluation of default, 190, 191
  example of default, 189, 190
  overloaded operator and default, 307
  reference, 94
  scope of default, 191
  template, 320
  type checking of default, 190
arguments
  implementation-defined order of evaluation of function, 191
  argument and name hiding
    default, 191
  argument and virtual function
    default, 192
  argument list
    empty, 186
    variable, 186
  argument passing, 94
    reference and, 203
  argument substitution, 402
  argument type
    unknown, 186
argv, 58
arithmetic
  pointer, 115
unsigned, 71
array, 186
  bound, 184
const, 74
delete, 111
multidimensional, 185
new, 108
overloading and pointer versus, 284
sizeof, 106
storage of, 185
array
  as aggregate, 740
  contiguous storage, 740
  initialization, 740, 742
  tuple interface to, 742
  zero sized, 742
array size
  default, 184
arrow operator, see class member access operator
asm
  implementation-defined, 166
asm-definition, 166, 1190
assembler, 166
<assert.h>, 423
assignment
  and lvalue, 120
  conversion by, 121
  move, 412
  reference, 203
assignment-expression, 121, 1186
assignment-operator, 121, 1186
assignment operator
  copy, 275
  overloaded, 308
associated asynchronous state, 1160
associative containers
  exception safety, 728
  requirements, 728
  unordered, see unordered associative containers
asynchronous provider, 1160
asynchronous return object, 1160
atexit, 61
attribute, 170
attribute, 170, 1190
attribute-argument-clause, 170, 1191
attribute-declaration, 135, 1188
attribute-list, 170, 1190
attribute-name, 170, 1191
attribute-scoped-token, 170, 1191
attribute-specifier, 170, 1190
attribute-token, 170, 1190
automatic storage duration, 62
awk, 1071
backslash character, 24
bad_alloc, 109
bad_cast, 98
bad_exception, 396
bad_typeid, 99
bad_typeid::what
  implementation-defined, 452
balanced-token, 170, 1191
balanced-token-seq, 170, 1191
base class subobject, 6
base-clause, 226, 1193
base-specifier, 226, 1193
base-specifier-list, 226, 1193
base-type-specifier, 226, 1194
BaseCharacteristic, 513
base class, 226, 227
  direct, 226
  indirect, 226
  private, 242
  protected, 242
  public, 242
base class virtual, see virtual base class
basic_ios::failure argument
  implementation-defined, 993
begin
  unordered associative containers, 735
behavior
  conditionally-supported, 2, 5
  default, 412, 416
  implementation-defined, 2, 696
  locale-specific, 3
  required, 413, 416
  undefined, 3
  unspecified, 3
Ben, 286
Bernoulli distributions, 923–926
bernoulli_distribution
  discrete probability function, 923
binary function, 534
BinaryTypeTrait, 513
binary operator
  interpretation of, 308
  overloaded, 308
bind directly, 205
binding
  reference, 203
binomial_distribution
  discrete probability function, 923
bit-field, 222
  address of, 223
  alignment of, 223
  implementation-defined sign of, 223
  implementation defined alignment of, 222
  type of, 223
  unnamed, 223
  zero width of, 223
block
  initialization in, 132
block scope, 37
block-declaration, 135, 1187
block scope; see local scope, 37
block structure, 132
body
  function, 192
INDEX
chi_squared_distribution
probability density function, 933
class, 73, 210
abstract, 236
base, 425, 430
cast to incomplete, 113
constructor and abstract, 238
definition, 33
derived, 430
linkage of, 56
linkage specification, 168
pointer to abstract, 237
polymorphic, 233
scope of enumerator, 153
standard-layout, 211
trivial, 210
unnamed, 142
class-head, 210, 1193
class-key, 210, 1193
class-name, 210, 1193
class-or-decltype, 226, 1193
class-specifier, 210, 1193
class base, see base class
class derived, see derived class
class local, see local class
class name, 177
elaborated, 149, 213
point of declaration, 213
scope of, 212
typedef, 141, 142, 213
class nested, see nested class
class object
assignment to, 121
const, 74
member, 215
sizeof, 106
class object copy, see also copy constructor
class object initialization, see also constructor
clear
unordered associative containers, 734
<locale>, 418
closure object, 88
closure type, 88
collating element, 1060
comment, 18
/ * */, 19
//, 19
comparison
pointer, 117
pointer to function, 117
undefined pointer, 115, 117
unspecified pointer, 117
void* pointer, 116
compatible, 393
compilation
separate, 15
compiler control line, see preprocessing directive
complete object, 6
completely defined, 214
component, 412
compound-statement, 125, 1186
concatenation
string, 27
undefined string literal, 28
condition, 126, 1187
conditions
rules for, 126
conditional-expression
throw-expression in, 119
conditional-expression, 119, 1186
conflict, 11
consistency
linkage, 138
linkage specification, 168
type declaration, 58
const, 73
constructor and, 219, 252
destructor and, 219, 259
linkage of, 55, 138
overloading and, 285
const_local_iterator, 729
unordered associative containers, 729
constant, 22, 85
equivalent, 151
null pointer, 81
constant iterator, 811
constant-expression, 122, 1186
constructor, 251
address of, 253
array of class objects and, 264
conversion by, 256
converting, 256
copy, 253, 254, 271, 272, 420
exception handling, 390
inheritance of, 252
move, 413
non-trivial, 252
random number distribution requirement, 903
random number engine requirement, 899, 900
type of, 253
union, 221
unspecified argument to, 109

INDEX 1251
constructor call
  explicit, 253
constructor conversion by, see also user-defined conversion
constructor default, see default constructor
context
  non-deduced, 379
control-line, 398, 1196
control line, see preprocessing directive
conversion
  argument, 186
  array pointer, 78
  array-to-pointer, 78
  Boolean, 82
  class, 255
  derived-to-base, 297
  floating point, 80
  floating-integral, 81
  function-to-pointer, 78
  implementation-defined floating point, 80
  implementation defined pointer integer, 101
  implicit, 77, 255
  implicit user-defined, 256
  inheritance of user-defined, 258
  integer, 80
  integer rank, 82
  lvalue-to-rvalue, 78, 1203
  narrowing, 209
overload resolution and, 295
overload resolution and pointer, 306
pointer, 81
  pointer to function, 78
  pointer to member, 81
  void*, 82
return type, 131
signed unsigned integer, 80
  standard, 77
static user-defined, 258
type of, 257
user-defined, 255–257
  virtual user-defined, 258
conversion operator, see conversion function
conversion rank, 298
conversion-declarator, 257, 1194
conversion-function-id, 257, 1194
conversion-type-id, 257, 1194
considerations
  qualification, 78
  usual arithmetic, 84
conversion explicit type, see casting
conversion function, see also user-defined conversion
copy
  class object, 271
copy constructor
  random number engine requirement, 899
copy elision, 278
copy-initialization, 198
copy assignment operator
  implicitly-declared, 275
copy constructor
  implicitly-declared, 273
count
  unordered associative containers, 735
<typename>, 186
<cstdlib>, 106, 115
<cstring>, 61, 422
<string>, 419
c-tor-initializer, 264, 1194
<uchar>, 419, 420, 425
cv-qualifier, 73
cv-qualifier, 177, 1191
cv-qualifier-seq, 177, 1191
<char>, 420, 425
<ctype>, 425
d-char, 26, 1182
d-char-sequence, 26, 1182
DAG
  multiple inheritance, 228, 229
  non-virtual base class, 229
  virtual base class, 228, 229
data race, 13
data member, see member
  static, 219
deadlock, 412
deallocation, see delete
deallocation functions, 63
decimal-literal, 22, 1180
decl-specifier, 137, 1188
decl-specifier-seq, 137, 1188
declaration, 31, 135
  extern reference, 203
typedef as type, 140
  access, 244
  array, 184
asm, 166
bit-field, 222
class member, 214
class name, 31
  constant pointer, 180
default argument, 189
definition versus, 31
ellipsis in function, 94, 186
eumerator point of, 36
extern, 31
forward, 139
forward class, 213
function, 31, 185
member, 214
multiple, 58
name, 31
overloaded, 283
overloaded name and friend, 247
parameter, 186
parentheses in, 178, 180
pointer, 180
reference, 181
register, 138
static member, 31
storage class, 137
type, 179
typedef, 32
declaration, 135, 1187
declaration-seq, 135, 1187
declaration-statement, 132, 1187
declaration hiding, see name hiding
declaration matching
  overloaded function, 285
declarative region, 35
declarator, 136, 176
  meaning of, 179
    multidimensional array, 184
declarator, 176, 1191
declarator-id, 177, 1192
decType-specifier, 147, 1189
decrement operator
  overloaded, 309
default
  access control, 239
default constructor
    random number distribution requirement, 903
    seed sequence requirement, 897
default-initialization, 197
defaulted, 194
default argument
  overload resolution and, 294
default constructor, 252
default initializers
  overloading and, 285
default function, 1173
definition, 31
static member, 220
alternate, 425
class, 210, 214
class name as type, 212
constructor, 192
declaration as, 136
empty class, 210
function, 192
local class, 224
member function, 216
namespace, 154
nested class, 223
pure virtual function, 237
scope of class, 212
virtual function, 235
delete, 62, 110, 111, 261
destructor and, 111, 260
operator, 426
overloading and, 64
type of, 262
undefined, 111
delete-expression, 110, 1185
deleter, 567
dependency-ordered before, 12
deprecated features, 97, 105
dereferencing, see also indirection
derivation, see inheritance
derived class, 226
  most, 6
    destructor call
      explicit, 260
      implicit, 259
    diagnosable rules, 4
digit, 20, 1179
digit-sequence, 25, 1181
digraph, 18
directed acyclic graph, see DAG
directive
  error, 407
  null, 407
pragma, 407
discard
   random number engine requirement, 900
discard_block_engine
genration algorithm, 912
state, 912
textual representation, 913
transition algorithm, 912
discrete probability function
bernoulli_distribution, 923
binomial_distribution, 923
discrete_distribution, 936
geometric_distribution, 924
negative_binomial_distribution, 925
poisson_distribution, 926
uniform_int_distribution, 921
discrete_distribution
   discrete probability function, 936
   weights, 937
distribution, see random number distribution
dominance
   virtual base class, 231
dot operator, see class member access operator
dynamic binding, see virtual function
dynamic initialization, 59
dynamic_exceptionSpecification, 392, 1196
ECMA-262, 2
ECMAScript, 1071, 1104
egrep, 1071
elaborated_typeSpecifier, 149, 1189
elaborated type specifier, see elaborated class name
elif-group, 397, 1196
elif-groups, 397, 1196
elision
   copy constructor, 278
ellipsis
   conversion sequence, 94, 299
   overload resolution and, 294
else, 126
else-group, 397, 1196
empty, 1165, 1167, 1170
empty_declaration, 135, 1188
encoding
   multibyte, 28
encoding_prefix, 26, 1181
end
   unordered associative containers, 735
end-of-file, 510
endif-line, 397, 1196
generation algorithm, 912
state, 912
textual representation, 913
transition algorithm, 912
discard
region method, 900
discard_block_engine
   generation algorithm, 912
   state, 912
   textual representation, 913
   transition algorithm, 912
discrete probability function
bernoulli_distribution, 923
binomial_distribution, 923
discrete_distribution, 936
geometric_distribution, 924
negative_binomial_distribution, 925
poisson_distribution, 926
uniform_int_distribution, 921
discrete_distribution
   discrete probability function, 936
   weights, 937
distribution, see random number distribution
dominance
   virtual base class, 231
dot operator, see class member access operator
dynamic binding, see virtual function
dynamic initialization, 59
dynamic_exceptionSpecification, 392, 1196
ECMA-262, 2
ECMAScript, 1071, 1104
egrep, 1071
elaborated_typeSpecifier, 149, 1189
elaborated type specifier, see elaborated class name
elif-group, 397, 1196
elif-groups, 397, 1196
elision
   copy constructor, 278
ellipsis
   conversion sequence, 94, 299
   overload resolution and, 294
else, 126
else-group, 397, 1196
empty, 1165, 1167, 1170
empty_declaration, 135, 1188
encoding
   multibyte, 28
encoding_prefix, 26, 1181
end
   unordered associative containers, 735
end-of-file, 510
endif-line, 397, 1196
equivalent parameter declarations, 284
overloading and, 284

erase
unordered associative containers, 734

escape-sequence, 24, 1181

escape character, see backslash
escape sequence
undefined, 24
Evaluation, 9
evaluation
order of argument, 95
unspecified order of, 10, 60
unspecified order of argument, 95
unspecified order of function call, 95

eexample
*const, 180
static member, 220
array, 184
class definition, 215
const, 180
constant pointer, 180
constructor, 253
constructor and initialization, 263
declaration, 32, 187
declarator, 177
definition, 32
delete, 262
derived class, 226
destructor and delete, 262
ellipsis, 186
enumeration, 152
explicit destructor call, 260
explicit qualification, 230
friend, 213
friend function, 245
function declaration, 187
function definition, 192
linkage consistency, 138
local class, 224
member function, 217, 245
member name access, 244
nested type name, 225
nested class, 223
nested class definition, 224, 250
nested class forward declaration, 224
pointer to member, 183
pure virtual function, 237
scope of delete, 262
scope resolution operator, 230
subscripting, 184
typedef, 140
type name, 177
unnamed parameter, 192
variable parameter list, 186
virtual function, 234, 235

exception
allowing an, 394
arithmetic, 83
handling, 387
object, 389
undefined arithmetic, 83

<exception>, 452
exception object, 389

exception-declaration, 387, 1195
exception-specification, 392, 1195
exclusive-or-expression, 118, 1186
exit, 59, 61, 131

explanation
subscripting, 184
explicit-instantiation, 360, 1195
explicit-specialization, 362, 1195
explicitly captured, 89
explicit type conversion, see casting

exponent-part, 25, 1181

exponential_distribution
probability density function, 927

expression, 83
constant, 122
lambda, 87
order of evaluation of, 8
parenthesized, 86
pointer to member constant, 104
postfix, 92
primary, 85
reference, 83
rvalue reference, 83
unary, 104

type name, 177
unnamed parameter, 192
variable parameter list, 186
virtual function, 234, 235

exception
allowing an, 394
arithmetic, 83
handling, 387
object, 389
undefined arithmetic, 83

<exception>, 452
exception object, 389

exception-declaration, 387, 1195
exception-specification, 392, 1195
exclusive-or-expression, 118, 1186
exit, 59, 61, 131

explanation
subscripting, 184
explicit-instantiation, 360, 1195
explicit-specialization, 362, 1195
explicitly captured, 89
explicit type conversion, see casting

exponent-part, 25, 1181

exponential_distribution
probability density function, 927

expression, 83
constant, 122
lambda, 87
order of evaluation of, 8
parenthesized, 86
pointer to member constant, 104
postfix, 92
primary, 85
reference, 83
rvalue reference, 83
unary, 104

type name, 177
unnamed parameter, 192
variable parameter list, 186
virtual function, 234, 235

exception
allowing an, 394
arithmetic, 83
handling, 387
object, 389
undefined arithmetic, 83

<exception>, 452
exception object, 389

exception-declaration, 387, 1195
exception-specification, 392, 1195
exclusive-or-expression, 118, 1186
exit, 59, 61, 131

explanation
subscripting, 184
explicit-instantiation, 360, 1195
explicit-specialization, 362, 1195
explicitly captured, 89
explicit type conversion, see casting

exponent-part, 25, 1181

exponential_distribution
probability density function, 927
file, 15
   source, 15, 423, 425
final overrider, 233
find
   unordered associative containers, 734
finite state machine, 1060
fisher_f_distribution
   probability density function, 935
floating-literal, 25, 1181
floating-suffix, 26, 1181
floating point type, 72
   implementation-defined, 72
for
   scope of declaration in, 130
for-init-statement, 128, 1187
for-range-declaration, 128, 1187
formal arguments, 3
formal parameters, 3
formal argument, see parameter
format specifier, 1060
forwarding call wrapper, 533
fractional-constant, 25, 1181
freestanding, 4
free store, see also new, delete
friend
   virtual and, 235
   access specifier and, 247
   class access and, 246
   inheritance and, 247
   local class and, 248
   template and, 333
friend function
   access and, 245
   inline, 247
   linkage of, 247
   member function and, 245
full-expression, 9
function, see friend function, member function, inline function, virtual function, 186
   allocation, 63, 108
   comparison, 412
   conversion, 257
deallocation, 64, 111, 261
definition, 33
global, 425, 428, 429
handler, 412
linkage specification overloaded, 169
modifier, 412
observer, 413
operator, 306
plain old, 460
   pointer to member, 114
   replacement, 413
   reserved, 413
   viable, 287
   virtual member, 425, 429
function object, 529
function object type, 529
function objects
   binders, 539–541
   mem_fn, 544
   reference_wrapper, 534
   wrapper, 544–549
function-definition, 192, 1192
function-specifier, 139, 1188
function-try-block, 387, 1195
functions
   candidate, 355
   function argument, see argument
   function call, 94
   recursive, 95
   undefined, 101
function call operator
   overloaded, 309
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, 76
fundamental type
   destructor and, 261
   fundamental type conversion, see conversion, user-defined conversion
gamma_distribution
   probability density function, 928
generate
   seed sequence requirement, 898
generated destructor, see default destructor
generation algorithm
   discard_block_engine, 912
   independent_bits_engine, 913
   linear_congruential_engine, 908
   mersenne_twister_engine, 909
   shuffle_order_engine, 914
   subtract_with_carry_engine, 911
gamma_distribution
   discrete probability function, 924
global, 39
global namespace, 39
global namespace scope, 39
global scope, 39
glvalue, 74
goto
  initialization and, 132
grammar
  regular expression, 1104
grep, 1071
group, 397, 1196
group-part, 397, 1196
h-char, 19, 1179
h-char-sequence, 19, 1179
handler
  exception, 390, 430
  incomplete type in exception, 390
handler, 387, 1195
handler-seq, 387, 1195
happens before, 12
Hash, 482
hash
  instantiation restrictions, 550
hash code, 729
hash function, 728
hash tables, see unordered associative containers
hash_function
  unordered associative containers, 732
hasher
  unordered associative containers, 729
header
  C, 423, 425, 428, 1214
header-name, 19, 1179
headers
  x C++ library, 421
hex-quad, 17, 1178
hexadecimal-digit, 22, 1180
hexadecimal-escape-sequence, 24, 1181
hexadecimal-literal, 22, 1180
hiding; see name hiding, 41
high-order bit, 5
hosted, 4
id
  qualified, 86
id-expression, 86
id-expression, 85, 1183
identifier, 20, 86, 136
identifier, 20, 1179
identifier-list, 398, 1197
identifier-nondigit, 20, 1179
if-group, 397, 1196
if-section, 397, 1196
immolation
  self, 364
implementation
  freestanding, 422
  hosted, 422
implementation-defined, 425, 433, 444, 449, 452, 454, 986, 1039, 1212
implementation-dependent, 1012
implementation-generated, 32
implicitly captured, 90
implicitly-declared default constructor, 252, see also
default constructor
implicit object parameter, 287
implied object argument, 287
  implicit conversion sequences, 288
inclusion
  conditional, 399
  source file, 400
inclusive-or-expression, 118, 1186
incomplete, 114
increment
  bool, 97, 105
increment operator
  overloaded, 309
independent_bits_engine
  generation algorithm, 913
  state, 913
  textual representation, 914
  transition algorithm, 913
indeterminately sequenced, 9
indirection, 104
inheritance, 226, see also multiple inheritance
init-declarator, 176, 1191
init-declarator-list, 176, 1191
initialization, 59, 196
  aggregate, 199
  array, 199
  array of class objects, 202, 264
  automatic, 132
  automatic object, 196
  base class, 264, 265
  character array, 202
  class member, 197
  class object, see also constructor, 199, 263
const, 145, 199
const member, 266
constant, 59
constructor and, 263
copy, 198
default, 196
default constructor and, 263
definition and, 136
direct, 198
dynamic, 59
jump past, 132
local static, 132
member, 264
member object, 265
order of, 59, 227
order of base class, 267
order of member, 267
order of virtual base class, 267
overloaded assignment and, 264
parameter, 94
reference, 182, 202
reference member, 266
run-time, 59
static and thread, 59
static member, 220
static object, 59
static object, 196
union, 202, 222
virtual base class, 274
initializer
  base class, 192
  member, 192
  scope of member, 267
temporary and declarator, 254
initializer, 196, 1192
initializer-clause, 196, 1192
initializer-list, 196, 1192
initializer-list constructor
  seed sequence requirement, 897
injected-class-name, 210
inline, 428
inline
  linkage of, 55
inline function, 140
insert
  unordered associative containers, 733, 734
instantiation
  explicit, 360
  point of, 354
template implicit, 357
instantiation units, 16
int
  bool promotion to, 80
integer representation, 65
integer-literal, 22, 1180
integer-suffix, 22, 1180
integer type, 72
integral type, 72
sizeof, 71
inter-thread happens before, 12
internal linkage, 55
interval boundaries
  piecewise_constant_distribution, 938
  piecewise_linear_distribution, 940
invocation
  macro, 402
isctype
  Regular Expression Traits, 1062
iteration-statement, 128, 131, 1187
Jessie, 256
jump-statement, 131, 1187
key_eq
  unordered associative containers, 732
key_equal
  unordered associative containers, 729
key_type
  unordered associative containers, 729
label, 132
case, 125, 127
default, 125, 127
  scope of, 38, 125
labeled-statement, 125, 1186
lambda-capture, 87, 1183
lambda-declarator, 88, 1184
lambda-expression, 87, 1183
lambda-introducer, 87, 147, 1183
lattice; see DAG
subobject, 227
layout
  bit-field, 222
class object, 215, 227
layout-compatible type, 71
left shift
  undefined, 116
left shift operator, 116
lexical conventions, 15
library
  x C++ Standard, 425, 427, 430
  x C++ standard, 410, 430
C standard, 1214
C standard, 421
Standard C, 411, 423, 1209, 1212
standard C, 418
library clauses, 5
lifetime, 65
limits
  implementation, 2
<limits>, 433
linear_congruential_engine
generation algorithm, 908
  modulus, 908
  state, 908
  textual representation, 908
  transition algorithm, 908
linkage, 31, 55
  external, 55, 423, 425
  implementation-defined object, 169
  internal, 55
linkage specification, 167, 1190
  linkage specification, 167
    extern, 167
      implementation-defined, 167
list
  operator, 21, 306
literal, 22, 85
  base of integer, 23
  char16_t, 24
  char32_t, 24
  character, 24
  decimal, 23
  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, 1180
  literal type, 70
literal-operator-id, 310, 1194
load_factor
  unordered associative containers, 736
  local lambda expression, 89
  local variable, 37
local_iterator, 729
  unordered associative containers, 729
locale, 1060, 1061, 1063, 1071
local class
  friend, 248
  member function in, 217
  scope of, 224
local variable
  destruction of, 131, 132
logical-and-expression, 118, 1186
logical-or-expression, 119, 1186
lognormal_distribution
  probability density function, 932
long
  typedef and, 137
long-long-suffix, 22, 1180
long-suffix, 22, 1180
lookup
  argument-dependent, 46
  member name, 229
  name, 31, 41
  template name, 343
lookup_classname
  Regular Expression Traits, 1106
lookup_classname
  Regular Expression Traits, 1062
lookup_collatename
  Regular Expression Traits, 1062
low-order bit, 5
lowercase, 418
lparen, 398, 1197
lvalue, 74, 1203
lvalue reference, 72, 181
macro
  function-like, 401
  masking, 428
  object-like, 401
main(), 58
  implementation-defined linkage of, 58
  implementation-defined parameters to, 58
  parameters to, 58
  return from, 61
  return from, 59
match_results
  as sequence, 1088
matched, 1060
max
  random number distribution requirement, 904
  uniform random number generator requirement, 898
max_bucket_count
  unordered associative containers, 735
max_load_factor
unordered associative containers, 736
mean
normal_distribution, 931
poisson_distribution, 926
mem-initializer, 264, 1194
mem-initializer-id, 264, 1194
mem-initializer-list, 264, 1194
member
class static, 62
enumerator, 153
static, 219
template and static, 329
member names, 38
member subobject, 6
member-declaration, 214, 1193
member-declarator, 214, 1193
member-declarator-list, 214, 1193
member-specification, 214, 1193
members, 38
member access operator
overloaded, 309
member function
class, 216
const, 218
constructor and, 253
destructor and, 259
friend, 247
inline, 216
local class, 225
nested class, 250
overload resolution and, 287
static, 219, 220
union, 221
volatile, 218
member function call
undefined, 217
member pointer to; see pointer to member, 73
member use
static, 219
memory location, 6
memory model, 5
memory management, see also new, delete
mersenne_twister_engine
generation algorithm, 909
state, 909
textual representation, 910
transition algorithm, 909
message
diagnostic, 2
min
random number distribution requirement, 904
uniform random number generator requirement, 898
modification order, 11
most derived class, 6
most derived object, 6
multi-pass guarantee, 814
multicharacter literal, 24
multiple inheritance, 226, 227
virtual and, 235
multiplicative-expression, 114, 1185
mutable, 137
mutable iterator, 811
name, 20, 31, 86
address of cv-qualified, 104
dependent, 349, 354
elaborated enum, 149
global, 39
length of, 20
macro, 401
overloaded function, 283
overloaded member, 214
point of declaration, 36
predefined macro, 407
qualified, 47
reserved, 424
scope of, 35
unqualified, 42
named-namespace-definition, 154, 1190
namespace, 421, 1214
global, 425
unnamed, 155
namespace-alias, 157, 1190
namespace-alias-definition, 154, 1190
namespace-body, 154, 1190
namespace-definition, 154, 1190
namespace-name, 154, 1190
namespaces, 153
name class, see class name
name hiding, 36, 41, 85, 86, 132
class definition, 212
function, 286
overloading versus, 286
user-defined conversion and, 256
name space
label, 125
narrowing conversion, 209
NDEBUG, 423
negative_binomial_distribution
discrete probability function, 925
nested-name-specifier, 86, 1183

nested class
  local class, 225
  scope of, 223
<new>, 426, 445

new, 62, 106, 108
  array of class objects and, 109
  constructor and, 109
  default constructor and, 109
  exception and, 109
  initialization and, 109
  operator, 426
  scoping and, 107
  storage allocation, 107
  type of, 261
  unspecified constructor and, 109
  unspecified order of evaluation, 109
new-declarator, 107, 1185
new-expression, 107, 1184
new-initializer, 107, 1185
new-line, 398, 1197
new-placement, 107, 1185
new-type-id, 107, 1185

new_handler, 63
  no linkage, 55
noexcept-expression, 111, 1185
noexcept-specification, 392, 1196

non-directive, 398, 1197
non-throwing, 394

non-digit, 20, 1179
nonzero-digit, 22, 1180

noptr-abstract-declarator, 177, 1192
noptr-declarator, 176, 1191
noptr-new-declarator, 107, 1185

normal distributions, 931–936

normal_distribution
  mean, 931
  probability density function, 931
  standard deviation, 931
notation
  syntax, 5
notify_all_at_thread_exit, 1150

NTBS, 419, 1048, 1223, 1224
  static, 419
NTC16S, 419
  static, 419
NTC32S, 420
  static, 420
NTCTS, 413
NTMBS, 419
  static, 419

NTWCS, 420
  static, 420

number
  hex, 25
  octal, 25
numeric_limits, 434
numeric_limits, 72

object, 6, 31
  complete, 6
  definition, 33
  delete, 110
  destructor static, 61
  destructor and placement of, 260
  linkage specification, 169
  local static, 62
  undefined deleted, 64
  unnamed, 253
object representation, 69
object type, 6, 70
object-expression, 84
object class, see also class object
object lifetime, 65
object temporary, see temporary
object type, 70
observable behavior, 8

octal-digit, 22, 1180
octal-escape-sequence, 24, 1181
octal-literal, 22, 1180
opaque-enum-declaration, 151, 1189

operator, 307
  **=, 120
  +=, 105, 120
  -=, 120
  /=, 120
  <<=, 120
  >>=, 120
  %=, 120
  &=, 120
  ^=, 120
  |=, 120
  additive, 114
  address-of, 104
  assignment, 120, 420
  bitwise, 118
  bitwise AND, 118
  bitwise exclusive OR, 118
  bitwise inclusive OR, 118
  cast, 104, 177
  class member access, 95
  comma, 122
conditional expression, 119
operator, 307, 1194
operator
overloaded, 306
operator delete, see also delete, 108, 111, 261
operator new, see also new, 108
operator()
random number distribution requirement, 903, 904
random number engine requirement, 900
uniform random number generator requirement, 898
operator-function-id, 307, 1194
operator<<
random number distribution requirement, 904
random number engine requirement, 901
operator left shift, see left shift operator
operator overloading, see also overloaded operator
operator right shift; right shift operator, 116
operator shift, see left shift operator, right shift operator
operator use
scope resolution, 220
optimization of temporary, see elimination of temporary
ordering
function template partial, 341
order of execution
base class constructor, 253
base class destructor, 259
constructor and static objects, 264
constructor and array, 263
destructor, 259
destructor and array, 259
member constructor, 253
member destructor, 259
original-namespace-definition, 154, 1190
original-namespace-name, 154, 1190
over-aligned type, 76
overflow, 83
undefined, 83
overloaded function
address of, 105, 305
overloaded operator, 306
inheritance of, 307
overloading, 186, 212, 283, 340
example of, 283
overloads
floating point, 895
overload resolution contexts, 287
overrider
final, 233
own, 567
pair
tuple interface to, 492
param
random number distribution requirement, 903
seed sequence requirement, 898
param_type
random number distribution requirement, 903
parameter, 3
reference, 181
scope of, 37
void, 186
parameter-declaration, 186, 1192
parameter-declaration-clause, 185, 1192
parameter-declaration-list, 185, 1192
parameter-type-list, 186
parameterized type, see template parameters
macro, 402
parameters-and-qualifiers, 176, 1191
parameter list
variable, 94, 186
period, 418
phases
translation, 15
piecewise construction, 493
piecewise_constant_distribution
interval boundaries, 938
probability density function, 938
weights, 938
piecewise_linear_distribution
interval boundaries, 940
probability density function, 940
weights at boundaries, 940
placement syntax
new, 108
pm-expression, 113, 1185
POD class, 211
POD struct, 211
POD union, 211
POF, 460
point of declaration, 36
pointer
safely-derived, 65
to traceable object, 64, 431
zero, 81
pointer, integer representation of safely-derived, 65
pointer-literal, 28, 1182
void*, 73
pointer to member, 73, 113
Poisson distributions, 926–931
poisson_distribution
discrete probability function, 926
mean, 926
POSIX, 2
extended regular expressions, 1071
regular expressions, 1071
postfix-expression, 92, 1184
postfix ++ and --
overloading, 309
postfix ++ and --, 97
potential scope, 35
potentially evaluated, 33
pp-number, 20, 1179
pp-tokens, 398, 1197
prefix
L, 24, 27
prefix ++ and --
overloading, 309
prefix ++ and dcr, 105
preprocessing, 397
preprocessing directive, 397
preprocessing-file, 397, 1196
preprocessing-op-or-punc, 21, 1180
preprocessing-token, 18, 1179
preprocessor
macro, 397
primary equivalence class, 1061
primary-expression, 85, 1183
private, 239
probability density function
cauchy_distribution, 934
chi_squared_distribution, 933
exponential_distribution, 927
extreme_value_distribution, 930
fisher_f_distribution, 935
gamma_distribution, 928
lognormal_distribution, 932
normal_distribution, 931
piecewise_constant_distribution, 938
piecewise_linear_distribution, 940
student_t_distribution, 936
uniform_real_distribution, 922
weibull_distribution, 929
program, 55
ill-formed, 2
well-formed, 4
promotion
floating point, 80
integral, 79
protected, 239
protection, see access control, 430
prvalue, 75
pseudo-destructor-name, 95
pseudo-destructor-name, 93, 1184
ptr-abstract-declarator, 177, 1192
ptr-declarator, 176, 1191
ptr-operator, 177, 1191
ptrdiff_t, 115
implementation defined type of, 115
public, 239
punctuators, 21
INDEX 1263
pure-specifier, 214, 1193
q-char, 20, 1179
q-char-sequence, 20, 1179
qualification
  explicit, 47
qualified-id, 86, 1183
qualified-name-space-specifier, 157, 1190
r-char, 26, 1182
r-char-sequence, 26, 1182
random number distribution
  bernoulli_distribution, 923
  binomial_distribution, 923
  chi_squared_distribution, 933
  discrete_distribution, 936
  exponential_distribution, 927
  extreme_value_distribution, 930
  fisher_f_distribution, 935
  gamma_distribution, 928
  geometric_distribution, 924
  lognormal_distribution, 932
  negative_binomial_distribution, 925
  normal_distribution, 931
  piecewise_constant_distribution, 938
  piecewise_linear_distribution, 940
  poisson_distribution, 926
  requirements, 902–905, 905
  student_t_distribution, 936
  uniform_int_distribution, 921
  uniform_real_distribution, 922
random number distributions
  Bernoulli, 923–926
  normal, 931–936
  Poisson, 926–931
  sampling, 936–942
  uniform, 921–922
random number engine
  linear_congruential_engine, 908
  mersenne twister_engine, 909
  requirements, 899–901
  subtract_with_carry_engine, 910
  with predefined parameters, 915–917
random number engine adaptor
  discard_block_engine, 912
  independent_bits_engine, 913
  shuffle_order_engine, 914
  with predefined parameters, 915–917
random number generation, 895–942
  distributions, 920–942
  engines, 907–915
  predefined engines and adaptors, 915–917
  requirements, 896–905
  synopsis, 905–907
  utilities, 918–920
random number generator, see uniform random number generator
random_device
  implementation leeway, 917
raw string literal, 27
raw-string, 26, 1182
reaching scope, 89
ready, 1160
redefinition
typedef, 141
ref-qualifier, 177, 1191
reference, 72
  assignment to, 121
  call by, 94
  lvalue, 72
  null, 182
  rvalue, 72
sizeof, 106
reference-compatible, 203
reference-related, 203
regex_iterator
  end-of-sequence, 1099
regex_token_iterator
  end-of-sequence, 1100
regex_traits
  specializations, 1075
region
  declarative, 31, 35
register, 137
regular expression, 1060–1106
  grammar, 1104
  matched, 1060
  requirements, 1061
Regular Expression Traits, 1104
  char_class_type, 1061
  isctype, 1062
  lookup_classname, 1106
  lookup_classname, 1062
  lookup_collatename, 1062
  requirements, 1061, 1075
transform, 1105
transform, 1062
transform_primary, 1106
transform_primary, 1062
translate, 1105
translate, 1062
translate_nocase, 1105
translate_nocase, 1062

rehash
unordered associative containers, 736

relational-expression, 116, 1185

relaxed pointer safety, 65
release sequence, 11
remainder operator, see modulus operator
replacement
macro, 401
replacement-list, 398, 1197

representation
object, 69
value, 69
requirements, 415

Allocator, 483
Container, 729, 740, 742
not required for unordered associated containers, 728
container, 710, 1088
CopyAssignable, 479
CopyConstructible, 479
DefaultConstructible, 479
Destructible, 479
EqualityComparable, 479
Hash, 482
iterator, 810
LessThanComparable, 479
MoveAssignable, 479
MoveConstructible, 479
NullablePointer, 482
numeric type, 884
random number distribution, 902–905, 905
random number engine, 899–901
Regular Expression Traits, 1061, 1075
seed sequence, 897–898
sequence, 1088
uniform random number generator, 898–899
Unordered Associative Container, 729

reraise, 389
rescanning and replacement, 404
reserved identifier, 21
reset, 567

reset
random number distribution requirement, 903
resolution
argument matching, see overload
function template overload, 385
overload, 286
overloaded function call resolution, see also argument matching, overload
resolution overloading, see overload
scoping ambiguity, 230
template name, 343
template overload, 341

restriction, 427, 428, 430
static member local class, 221
address of bit-field, 223
anonymous union, 222
bit-field, 223
compiler, 252, 253
destructor, 259
extern, 138
local class, 225
overloading, 307
pointer to bit-field, 223
reference, 182
register, 138
static, 138
union, 221

restrictions
operator overloading, 307
result, 1160
result_type
entity characterization based on, 896
result_type
random number distribution requirement, 903
seed sequence requirement, 897
uniform random number generator requirement, 898
rthrow, 389

return, 131
constructor and, 131
reference and, 203
return statement, see also return
return type, 187
overloading and, 283
right shift
implementation defined, 116
right shift operator, 116

rounding, 81
rule
as-if, 7
one-definition, 33

rvalue, 75
lvalue conversion to, 78
lvalue conversion to, 1203
rvalue reference, 72, 181

s-char, 26, 1182
s-char-sequence, 26, 1182
safely-derived pointer, 65
integer representation, 65
sampling distributions, 936–942
scalar type, 70
scope, 31, 35
  anonymous union at namespace, 222
class, 39
destructor and exit from, 131
enumeration, 40
exception declaration, 38
function, 38
global, 39
global namespace, 39
iteration-statement, 128
local, 37
macro definition, 404
namespace, 38
overloading and, 285
potential, 35
selection-statement, 126
scope resolution operator, 48
seed
  random number engine requirement, 900
seed sequence, 897
  requirements, 897–898
selection-statement, 126, 1187
semantics
  class member, 95
sequence
  ambiguous conversion, 298
  implicit conversion, 297
  standard conversion, 77
  statement, 125
sequence constructor
  seed sequence requirement, 897
Sequenced before, 9
sequencing operator, see comma operator
setlocale, 418
shift-expression, 116, 1185
shift operator, see left shift operator, right shift operator
short
typedef and, 137
shuffle_order_engine
generation algorithm, 914
state, 914
textual representation, 915
transition algorithm, 914
side effects, 9, 10
sign, 25, 1181
signature, 3
signed
typedef and, 137
signed integer type, 71
simple call wrapper, 533
simple-declaration, 135, 1188
simple-escape-sequence, 24, 1181
simple-template-id, 319, 1195
simple-type-specifier, 147, 1189
size
  seed sequence requirement, 898
size_t, 106
sizeof
  empty class, 210
smart pointers, 577–593
space
  white, 18
specialization
  class template, 320
class template partial, 335
template, 356
template explicit, 362
special member function, see constructor, destructor,
  inline function, user-defined conversion, virtual function
specification
  template argument, 367
specifications
  x C++ standard library exception, 430
  implementation-defined exception, 431
  Standard C library exception, 430
specifier
  declaration, 137
  explicit, 140
friend, 430
friend, 142
function, 139
inline, 140
missing storage class, 138
static, 138
storage class, 137
typedef, 140
virtual, 140
specifier access, see access specifier
specifier type, see type specifier
stable algorithm, 413
standard
  structure of, 5
standard deviation
  normal_distribution, 931
standard-layout types, 70
standard-layout class, 211
standard-layout struct, 211
standard-layout union, 211
standard integer type, 71
standard signed integer type, 71
standard unsigned integer type, 71
start
  program, 58, 59
startup
  program, 423, 426
state
  discard_block_engine, 912
  independent_bits_engine, 913
  linear_congruential_engine, 908
  mersenne_twister_engine, 909
  object, 413
  shuffle_order_engine, 914
  subtract_with_carry_engine, 910
statement, 125
  continue in for, 129
  break, 131
  compound, 125
  continue, 131
  declaration, 132
  declaration in for, 130
  declaration in switch, 128
  do, 128, 129
  empty, 125
  expression, 125
  for, 128, 129
  goto, 125, 131, 132
  if, 126, 127
  iteration, 128
  jump, 131
  labeled, 125
  null, 125
  selection, 126
  switch, 126, 127, 131
  while, 128
statement, 125, 1186
statement-seq, 126, 1186
static, 137
  destruction of local, 133
  linkage of, 55, 138
  overloading and, 283
static initialization, 59
static storage duration, 62
static_assert, 136
static_assert-declaration, 135, 1188
<stdio.h>, 24, 27
<stdioexcept>, 461
storage-class-specifier, 137, 1188
storage class, 31
storage duration, 61
  automatic, 62
  class member, 65
  dynamic, 62, 107
  local object, 62
  register, 62
storage management, see new, delete
stream
  arbitrary-positional, 411
  repositional, 413
streambuf
  implementation-defined, 973
strict pointer safety, 65
string
  distinct, 27
  null-terminated byte, 419
  null-terminated char16-character, 419
  null-terminated char32-character, 420
  null-terminated character type, 413
  null-terminated multibyte, 419
  null-terminated wide-character, 420
sizeof, 28
type of, 27
string-literal, 26, 1181
string literal, 27
  char16_t, 27
  char32_t, 27
  implementation-defined, 27
  narrow, 27
type of, 27
  undefined change to, 27
  wide, 27
struct
  standard-layout, 211
struct
  class versus, 210
structure, 210
structure tag, see class name
student_t_distribution
  probability density function, 936
sub-expression, 1061
subobject, 6
subobjects, 6
subscripting operator
  overloaded, 309
subsequence rule
  overloading, 302
subtract_with_carry_engine
  carry, 910
  generation algorithm, 911
  state, 910
textual representation, 911
transition algorithm, 910
subtraction
  implementation defined pointer, 115
subtraction operator, 114
suffix
  E, 26
  F, 26
  f, 26
  L, 23, 26
  l, 23, 26
  U, 23
  u, 23
summary
  compatibility with ISO C, 1200
  scope rules, 41
swappable, 481
swappable with, 481
synchronize with, 11
synonym, 157
  type name as, 140
syntax
  class member, 95
target object, 533
template, 315
  definition of, 315
  function, 367
  member function, 328
  primary, 335
template, 315
template alias, 342
template-argument, 319, 1195
template-argument-list, 319, 1195
template-declaration, 315, 1194
template-id, 319, 1195
template-name, 319, 1195
template-parameter, 316, 1194
template-parameter-list, 315, 1194
template name
  linkage of, 315
temporary, 253
  constructor for, 254
  destruction of, 254
  destructor for, 254
  elimination of, 253, 278
  implementation-defined generation of, 253
  order of destruction of, 254
terminate(), 395, 396
termination
  program, 59, 61
terminology
pointer, 73
text-line, 398, 1196
textual representation
  discard_block_engine, 913
  independent_bits_engine, 914
  shuffle_order_engine, 915
  subtract_with_carry_engine, 911
the complete object of, 6
this, 85
  type of, 218
this pointer, see this
thread, 10
thread of execution, 10
thread storage duration, 62
thread, blocked, 411
thread_local, 137
throw, 387
throw-expression, 387, 1195
throwing
  exception, 388
token, 22
token, 19, 1179
traceable pointer object, 64, 431
trailing-return-type, 176, 1191
trailing-type-specifier, 144, 1188
trailing-type-specifier-seq, 145, 1188
traits, 414
transfer ownership, 567
transform
  Regular Expression Traits, 1105
transform
  Regular Expression Traits, 1062
transform_primary
  Regular Expression Traits, 1106
transform_primary
  Regular Expression Traits, 1105
transform_primary
  Regular Expression Traits, 1062
TransformationTrait, 514
transition algorithm
  discard_block_engine, 912
  independent_bits_engine, 913
  linear_congruential_engine, 908
  mersenne_twister_engine, 909
  shuffle_order_engine, 914
  subtract_with_carry_engine, 910
translate
  Regular Expression Traits, 1105
translate
  Regular Expression Traits, 1062
translate_nocase

INDEX 1268
Regular Expression Traits, 1105

translate_nocase

Regular Expression Traits, 1062

translation

separate, 15

translation unit, 15

translation units, 55

translation-unit, 55, 1183

translation unit, 55

name and, 31

trigraph, 15

trivial types, 70

trivially copyable class, 210

trivially copyable types, 70

trivial class, 210

trivial class type, 109

trivial type, 109

truncation, 81

try, 387

try-block, 387, 1195

tuple

and pair, 492

type, 31

arithmetic, 72

array, 72, 186

bitmask, 417, 418

Boolean, 71

char, 71

char16_t, 72

char32_t, 72

character, 71

character container, 411

class and, 210

compound, 72

const, 144

destination, 198

double, 72

dynamic, 2

enumerated, 73, 417

enumeration underlying, 152

example of incomplete, 70

extended integer, 71

extended signed integer, 71

extended unsigned integer, 71

float, 72

floating point, 71

function, 72, 185, 186

fundamental, 71

sizeof, 71

incomplete, 33, 36, 69, 78, 93–97, 99, 104, 106, 111, 121, 226

int, 71

integral, 71

long, 71

long double, 72

long long, 71

multi-level mixed pointer and pointer to member, 79

multi-level pointer to member, 79

object, 6

over-aligned, 76

POD, 70

pointer, 72

polymorphic, 233

short, 71

signed char, 71

signed integer, 71

standard integer, 71

standard signed integer, 71

standard unsigned integer, 71

static, 3

underlying wchar_t, 72

unsigned, 71

unsigned char, 71

unsigned int, 71

unsigned long, 71

unsigned long long, 71

unsigned short, 71

unsigned integer, 71

void, 72

volatile, 144

wchar_t, 72

type-id, 177, 1192

type-id-list, 392, 1196

type-name, 147, 1189

type-parameter, 316, 1195

type-specifier

bool, 147

wchar_t, 147

type-specifier, 144, 1188

type-specifier-seq, 145, 1188

type_info, 98

typedef

function, 187

typedef

overloading and, 284

typedef-name, 140, 1188

typeid, 98

<typeinfo>, 450

typename, 149

typename-specifier, 344, 1195

types
implementation-defined, 417
implementation-defined exception, 431
type checking
argument, 94
type conversion, explicit, see casting
type generator, see template
type name, 177
  nested, 225
  scope of nested, 225
type pun, 102
type specifier
  char, 147
  char16_t, 147
  char32_t, 147
  class, 210
  double, 147
  enum, 149
  float, 147
  int, 147
  long, 147
  short, 147
  signed, 147
  struct, 210
  union, 210
  unsigned, 147
  void, 147
  volatile, 146

_ud-suffix, 29, 1182
unary function, 534
_unary-expression, 104, 1184
_unary-operator, 104, 1184
UnaryTypeTrait, 513
unary operator
  interpretation of, 308
  overloaded, 308
undefined, 413, 424, 425, 427, 630, 947, 949, 951, 952, 954, 955, 959, 963, 989
undefined behavior, 840
underlying type, 72
unexpected(), 396
Unicode required set, 408
uniform distributions, 921–922
uniform random number generator
  requirements, 898–899
uniform_int_distribution
  discrete probability function, 921
uniform_real_distribution
  probability density function, 922
union
  standard-layout, 211
union, 73, 221
  class versus, 210
  anonymous, 222
  global anonymous, 222
unique pointer, 567
unit
  translation, 423–425
universal-character-name, 17, 1178
unnamed-name-space-definition, 154, 1190
unordered associative containers, 728–809
  begin, 735
  bucket, 735
  bucket_count, 735
  bucket_size, 735
  cbegin, 735
  cend, 735
  clear, 734
  complexity, 728
  const_local_iterator, 729
  count, 735
  end, 735
  equal_range, 735
  equality function, 728
  equivalent keys, 728, 729, 800, 806
  erase, 734
  exception safety, 737
  find, 734
  hash function, 728
  hash_function, 732
  hasher, 729
  insert, 733, 734
  iterator invalidation, 737
  iterators, 736
  key_eq, 732
  key_equal, 729
  key_type, 729
  lack of comparison operators, 728
  load_factor, 736
  local_iterator, 729
  max_bucket_count, 735
  max_load_factor, 736
  rehash, 736
  requirements, 728, 729, 737
  unique keys, 728, 729, 796, 803
unordered_map
  element access, 799
  unique keys, 796
unordered_multimap
  equivalent keys, 800
unordered_multiset
  equivalent keys, 806

INDEX 1270
unordered_set
  unique keys, 803
unqualified-id, 85, 1183
unsequenced, 9
unsigned
typedef and, 137
unsigned-suffix, 22, 1180
unsigned integer type, 71
unspecified, 446, 447, 451, 869, 1038, 1218, 1220
unspecified behavior, 952
unwinding
  stack, 390
uppercase, 418, 425
used, 33
user-defined-character-literal, 29, 1182
user-defined-floating-literal, 29, 1182
user-defined-integer-literal, 29, 1182
user-defined-literal, 29, 1182
user-defined-string-literal, 29, 1182
user-provided, 194
Uses-allocator construction, 554
using-declaration, 158
using-declaration, 158, 1190
using-directive, 164
using-directive, 164, 1190
valid, 35
value, 69
call by, 94
  null member pointer, 81
  null pointer, 81
  undefined unrepresentable integral, 81
value category, 75
value representation, 69
value-initialization, 197
ValueSwappable, 481
variable, 31
  indeterminate uninitialized, 196
virtual base class, 228
virtual function, 233
  pure, 237
virtual function call, 236
  constructor and, 270
  destructor and, 270
  undefined pure, 238
visibility, 41
visible, 41
visible sequence of side effects, 13
visible side effect, 13
void*
type, 73

void&, 181
volatile, 73
creator and, 219, 252
destructor and, 219, 259
implementation-defined, 146
overloading and, 285
wchar_t, 24, 27, 420, 650
  implementation-defined, 72
weak result type, 533
weibull_distribution
  probability density function, 929
weights
discrete_distribution, 937
piecewise_constant_distribution, 938
weights at boundaries
  piecewise_linear_distribution, 940
white space, 19
wide-character, 24
X(X&), see copy constructor
xvalue, 74
zero
  undefined division by, 83, 114
  undefined modulus, 83
zero-initialization, 196
Index of Grammar Productions

The first page number for each entry is the page in the general text where the grammar production is defined. The second page number is the corresponding page in the Grammar summary (Annex A).

abstract-declarator, 177, 1192
access-specifier, 226, 1194
additive-expression, 114, 1185
alias-declaration, 135, 1187
and-expression, 118, 1185
asm-definition, 166, 1190
assignment-expression, 121, 1186
assignment-operator, 121, 1186
attribute, 170, 1190
attribute-argument-clause, 170, 1191
attribute-declaration, 135, 1188
attribute-list, 170, 1190
attribute-namespace, 170, 1191
attribute-scoped-token, 170, 1191
attribute-specifier, 170, 1190
attribute-token, 170, 1190
balanced-token, 170, 1191
base-clause, 226, 1193
base-specifier, 226, 1193
base-specifier-list, 226, 1193
base-type-specifier, 226, 1194
block-declaration, 135, 1187
boolean-literal, 28, 1182
brace-or-equal-initializer, 196, 1192
braced-init-list, 196, 1192
c-char, 24, 1181
c-char-sequence, 24, 1181
capture, 88, 1184
capture-default, 87, 1183
capture-list, 88, 1183
cast-expression, 112, 1185
character-literal, 23, 1181
class-head, 210, 1193
class-key, 210, 1193
class-name, 210, 1193
class-or-decltype, 226, 1193
class-specifier, 210, 1193
compound-statement, 125, 1186
condition, 126, 1187
conditional-expression, 119, 1186
constant-expression, 122, 1186
control-line, 398, 1196
conversion-declarator, 257, 1194
conversion-function-id, 257, 1194
conversion-type-id, 257, 1194
cctor-initializer, 264, 1194
cv-qualifier, 177, 1191
cv-qualifier-seq, 177, 1191
d-char, 26, 1182
d-char-sequence, 26, 1182
decimal-literal, 22, 1180
dcl-specifier, 137, 1188
dcl-specifier-seq, 137, 1188
declaration, 135, 1187
declaration-seq, 135, 1187
declaration-statement, 132, 1187
declarator, 176, 1191
declspec-id, 177, 1192
dcltype-specifier, 147, 1189
delete-expression, 110, 1185
digit, 20, 1179
digit-sequence, 25, 1181
dynamic-exception-specification, 392, 1196
elaborated-type-specifier, 149, 1189
eif-group, 397, 1196
eif-groups, 397, 1196
else-group, 397, 1196
empty-declaration, 135, 1188
encoding-prefix, 26, 1181
endif-line, 397, 1196
enum-base, 151, 1189
enum-head, 150, 1189
enum-key, 151, 1189
enum-name, 150, 1189
enum-specifier, 150, 1189
enumerator, 151, 1190
enumerator-definition, 151, 1189
enumerator-list, 151, 1189
equality-expression, 117, 1185
escape-sequence, 24, 1181
exception-declaration, 387, 1195
exception-specification, 392, 1195
exclusive-or-expression, 118, 1186
explicit-instantiation, 360, 1195
explicit-specialization, 362, 1195
exponent-part, 25, 1181
expression, 122, 1186
expression-list, 92, 1184
expression-statement, 125, 1186
extension-namespace-definition, 154, 1190
floating-literal, 25, 1181
floating-suffix, 26, 1181
for-init-statement, 128, 1187
for-range-declaration, 128, 1187
fractional-constant, 25, 1181
function-definition, 192, 1192
function-specifier, 139, 1188
function-try-block, 387, 1195
group, 397, 1196
group-part, 397, 1196
h-char, 19, 1179
h-char-sequence, 19, 1179
handler, 387, 1195
handler-seq, 387, 1195
header-name, 19, 1179
hex-quad, 17, 1178
hexadecimal-digit, 22, 1180
hexadecimal-escape-sequence, 24, 1181
hexadecimal-literal, 22, 1180
id-expression, 85, 1183
identifier, 20, 1179
identifier-list, 398, 1197
identifier-nondigit, 20, 1179
if-group, 397, 1196
if-section, 397, 1196
inclusive-or-expression, 118, 1186
init-declarator, 176, 1191
init-declarator-list, 176, 1191
initializer, 196, 1192
initializer-clause, 196, 1192
initializer-list, 196, 1192
integer-literal, 22, 1180
integer-suffix, 22, 1180
iteration-statement, 128, 1187
jump-statement, 131, 1187
labeled-statement, 125, 1186
lambda-capture, 87, 1183
lambda-declarator, 88, 1184
lambda-expression, 87, 1183
lambda-introducer, 87, 1183
linkage-specification, 167, 1190
literal, 22, 1180
literal-operator-id, 310, 1194
logical-and-expression, 118, 1186
logical-or-expression, 119, 1186
long-long-suffix, 22, 1180
long-suffix, 22, 1180
lparen, 398, 1197
mem-initializer, 264, 1194
mem-initializer-id, 264, 1194
mem-initializer-list, 264, 1194
member-declaration, 214, 1193
member-declarator, 214, 1193
member-declarator-list, 214, 1193
member-specification, 214, 1193
multiplicative-expression, 114, 1185
named-namespace-definition, 154, 1190
namespace-alias, 157, 1190
namespace-alias-definition, 157, 1190
namespace-body, 154, 1190
namespace-definition, 154, 1190
namespace-name, 154, 1190
nested-name-specifier, 86, 1183
new-declarator, 107, 1185
new-expression, 107, 1184
new-initializer, 107, 1185
new-line, 398, 1197
new-placement, 107, 1185
new-type-id, 107, 1185
noexcept-expression, 111, 1185
noexcept-specification, 392, 1196
non-directive, 398, 1197
nondigit, 20, 1179
nonzero-digit, 22, 1180
noptr-abstract-declarator, 177, 1192
noptr-declarator, 176, 1191
noptr-new-declarator, 107, 1185
octal-digit, 22, 1180
octal-escape-sequence, 24, 1181
octal-literal, 22, 1180
opaque-enum-declaration, 151, 1189
operator, 307, 1194
operator-function-id, 307, 1194
original-namespace-definition, 154, 1190
original-namespace-name, 154, 1190
# Index of Library Names

<table>
<thead>
<tr>
<th>Name</th>
<th>Line Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Exit</td>
<td>443</td>
</tr>
<tr>
<td>_1</td>
<td>541</td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>cauchy_distribution</td>
<td>934</td>
</tr>
<tr>
<td>extreme_value_distribution</td>
<td>930</td>
</tr>
<tr>
<td>uniform_int_distribution</td>
<td>921</td>
</tr>
<tr>
<td>uniform_real_distribution</td>
<td>922</td>
</tr>
<tr>
<td>weibull_distribution</td>
<td>930</td>
</tr>
<tr>
<td>abort</td>
<td>61, 131, 422, 443, 450, 455</td>
</tr>
<tr>
<td>abs</td>
<td>955, 968</td>
</tr>
<tr>
<td>complex</td>
<td>893</td>
</tr>
<tr>
<td>accumulate</td>
<td>965</td>
</tr>
<tr>
<td>acos</td>
<td>955, 968</td>
</tr>
<tr>
<td>complex</td>
<td>894</td>
</tr>
<tr>
<td>acosh</td>
<td>968</td>
</tr>
<tr>
<td>complex</td>
<td>894</td>
</tr>
<tr>
<td>address</td>
<td></td>
</tr>
<tr>
<td>allocator</td>
<td>558, 559</td>
</tr>
<tr>
<td>addressof</td>
<td>566</td>
</tr>
<tr>
<td>adjacent_difference</td>
<td>966</td>
</tr>
<tr>
<td>adjacent_find</td>
<td>857</td>
</tr>
<tr>
<td>advance</td>
<td>821</td>
</tr>
<tr>
<td>&lt;algorithm&gt;</td>
<td>844</td>
</tr>
<tr>
<td>align</td>
<td>593</td>
</tr>
<tr>
<td>all</td>
<td></td>
</tr>
<tr>
<td>bitset</td>
<td>509</td>
</tr>
<tr>
<td>all_of</td>
<td>854</td>
</tr>
<tr>
<td>allocate</td>
<td></td>
</tr>
<tr>
<td>allocator</td>
<td>559</td>
</tr>
<tr>
<td>allocator_traits</td>
<td>557</td>
</tr>
<tr>
<td>scoped_allocator_adaptor</td>
<td>563</td>
</tr>
<tr>
<td>allocator</td>
<td>1092</td>
</tr>
<tr>
<td>allocator</td>
<td>558</td>
</tr>
<tr>
<td>address</td>
<td>558, 559</td>
</tr>
<tr>
<td>allocate</td>
<td>559</td>
</tr>
<tr>
<td>constructor</td>
<td>559</td>
</tr>
<tr>
<td>deallocate</td>
<td>559</td>
</tr>
<tr>
<td>destructor</td>
<td>559</td>
</tr>
<tr>
<td>max_size</td>
<td>559</td>
</tr>
<tr>
<td>operator!=</td>
<td>559</td>
</tr>
<tr>
<td>operator==</td>
<td>559</td>
</tr>
<tr>
<td>allocator_arg</td>
<td>554</td>
</tr>
<tr>
<td>allocator_arg_t</td>
<td>554</td>
</tr>
<tr>
<td>allocator_traits</td>
<td>555</td>
</tr>
<tr>
<td>allocate</td>
<td>557</td>
</tr>
<tr>
<td>const_pointer</td>
<td>556</td>
</tr>
<tr>
<td>const_void_pointer</td>
<td>556</td>
</tr>
<tr>
<td>constructor</td>
<td>557</td>
</tr>
<tr>
<td>deallocate</td>
<td>557</td>
</tr>
<tr>
<td>destructor</td>
<td>557</td>
</tr>
<tr>
<td>difference_type</td>
<td>557</td>
</tr>
<tr>
<td>max_size</td>
<td>557</td>
</tr>
<tr>
<td>pointer</td>
<td>556</td>
</tr>
<tr>
<td>propagate_on_container_copy_assignment</td>
<td>557</td>
</tr>
<tr>
<td>propagate_on_container_move_assignment</td>
<td>557</td>
</tr>
<tr>
<td>propagate_on_container_swap</td>
<td>557</td>
</tr>
<tr>
<td>rebind_alloc</td>
<td>557</td>
</tr>
<tr>
<td>select_on_container_copy_construction</td>
<td>557</td>
</tr>
<tr>
<td>size_type</td>
<td>557</td>
</tr>
<tr>
<td>void_pointer</td>
<td>556</td>
</tr>
<tr>
<td>alpha</td>
<td></td>
</tr>
<tr>
<td>gamma_distribution</td>
<td>929</td>
</tr>
<tr>
<td>always_noconv</td>
<td></td>
</tr>
<tr>
<td>codecvt</td>
<td>675</td>
</tr>
<tr>
<td>any</td>
<td></td>
</tr>
<tr>
<td>bitset</td>
<td>509</td>
</tr>
<tr>
<td>any_of</td>
<td>855</td>
</tr>
<tr>
<td>append</td>
<td></td>
</tr>
<tr>
<td>basic_string</td>
<td>631</td>
</tr>
<tr>
<td>basic_string</td>
<td>631, 632</td>
</tr>
<tr>
<td>apply</td>
<td></td>
</tr>
<tr>
<td>valarray</td>
<td>953</td>
</tr>
<tr>
<td>arg</td>
<td>895</td>
</tr>
<tr>
<td>complex</td>
<td>893</td>
</tr>
<tr>
<td>&lt;array&gt;</td>
<td>737</td>
</tr>
<tr>
<td>array</td>
<td>740, 742</td>
</tr>
<tr>
<td>begin</td>
<td>740</td>
</tr>
<tr>
<td>data</td>
<td>742</td>
</tr>
<tr>
<td>end</td>
<td>740</td>
</tr>
<tr>
<td>fill</td>
<td>742</td>
</tr>
<tr>
<td>get</td>
<td>743</td>
</tr>
<tr>
<td>max_size</td>
<td>740</td>
</tr>
<tr>
<td>size</td>
<td>740, 742</td>
</tr>
<tr>
<td>swap</td>
<td>742</td>
</tr>
<tr>
<td>asin</td>
<td>955, 968</td>
</tr>
<tr>
<td>complex</td>
<td>894</td>
</tr>
<tr>
<td>asinh</td>
<td>968</td>
</tr>
<tr>
<td>complex</td>
<td>894</td>
</tr>
<tr>
<td>&lt;assert.h&gt;</td>
<td>423</td>
</tr>
</tbody>
</table>
assign
  basic_string, 632
deque, 746
list, 757
basic_regex, 1080, 1081
basic_string, 633
deque, 746
error_code, 472
error_condition, 474
forward_list, 751
function, 548
list, 758
vector, 772, 773
async, 1172
at
  basic_string, 630
map, 783
unordered_map, 799
at_quick_exit, 444
atan, 955, 968
complex, 894
atan2, 955, 968
atanh, 968
complex, 894
atexit, 61, 422, 443, 444
atomic type
  atomic_compare_exchange_strong, 1123
  atomic_compare_exchange_strong_explicit, 1123
  atomic_compare_exchange_weak, 1123
  atomic_compare_exchange_weak_explicit, 1123
  atomic_exchange, 1123
  atomic_exchange_explicit, 1123
  atomic_fetch_, 1125
  atomic_flag
    clear, 1127
    atomic_flag_clear, 1127
    atomic_flag_clear_explicit, 1127
    atomic_flag_test_and_set, 1127
    atomic_flag_test_and_set_explicit, 1127
  atomic_future
    constructor, 1170
destructor, 1171
get, 1171
operator=, 1171
valid, 1171
wait, 1171
wait_for, 1172
wait_until, 1172
atomic_is_lock_free, 1122
atomic_load, 1123
atomic_load_explicit, 1123
atomic_store, 1123
atomic_store_explicit, 1123
compare_exchange_strong, 1123
compare_exchange_strong_explicit, 1123
compare_exchange_weak, 1123
compare_exchange_weak_explicit, 1123
constructor, 1122
destructor, 1123
get, 1125
load, 1123
operator @=, 1125
operator C, 1123
operator++ , 1125, 1126
operator--, 1126
operator=, 1123
store, 1123
atomic_compare_exchange_strong
  atomic type, 1123
  shared_ptr, 591
atomic_compare_exchange_strong_explicit
  atomic type, 1123
  shared_ptr, 591
atomic_compare_exchange_weak
  atomic type, 1123
  shared_ptr, 591
atomic_compare_exchange_weak_explicit
  atomic type, 1123
  shared_ptr, 591
atomic_exchange
  atomic type, 1123
  shared_ptr, 591
atomic_exchange_explicit
  atomic type, 1123
  shared_ptr, 591
atomic_fetch_
  atomic type, 1125
atomic_flag
  clear, 1127
atomic_flag_clear, 1127
atomic_flag_clear_explicit, 1127
atomic_flag_test_and_set, 1127
atomic_flag_test_and_set_explicit, 1127
atomic_future
  constructor, 1170
destructor, 1171
get, 1171
operator=, 1171
valid, 1171
wait, 1171
wait_for, 1172
wait_until, 1172
atomic_is_lock_free, 1122
atomic_load, 1123
atomic_load_explicit, 1123
atomic_signal_fence, 1128
atomic_store
  atomic type, 1123
  shared_ptr, 590
atomic_store_explicit
  atomic type, 1123
  shared_ptr, 590
atomic_type
  shared_ptr, 590
INDEX OF LIBRARY NAMES 1276
shared_ptr, 590
atomic_thread_fence, 1127
auto_ptr, 580, 1227
    auto_ptr, 1228
    auto_ptr_ref, 1229
    constructor, 1228, 1229
destructor, 1228
    operator=, 1228
auto_ptr_ref
    auto_ptr, 1229
    operator auto_ptr, 1229
    operator=, 1229
b
    cauchy_distribution, 934
    extreme_value_distribution, 931
    uniform_int_distribution, 921
    uniform_real_distribution, 922
    weibull_distribution, 930
back
    basic_string, 631
    back_insert_iterator, 828
        back_insert_iterator, 828
    back_inserter, 829
bad
    basic_ios, 994
    bad_alloc, 109, 446, 450
        bad_alloc, 449
        bad_alloc::what
            implementation-defined, 449
    bad_array_new_length
        bad_array_new_length, 450
    bad_cast, 98, 451
        bad_cast, 451, 452
    bad_cast::what
        implementation-defined, 452
    bad_exception, 454
    bad_exception::what
        implementation-defined, 454
    bad_function_call, 544
        bad_function_call, 545
    bad_typeid, 99, 452
        bad_typeid, 452
    bad_weak_ptr, 577
        bad_weak_ptr, 577
        what, 577
base
    move_iterator, 834
    reverse_iterator, 824
    basic_filebuf, 973, 1045
    basic_filebuf<char>, 1045
    basic_filebuf<wchar_t>, 1045
    basic_fstream, 973, 1056
        basic_fstream, 1057
            constructor, 1057
            operator=, 1058
        swap, 1058
    basic_ifstream, 973, 1052
        basic_ifstream, 1053
            constructor, 1053
            operator=, 1053
        swap, 1054
    basic_ifstream<char>, 1045
    basic_ifstream<wchar_t>, 1045
    basic_ios, 973, 989
        basic_ios, 990
            constructor, 990
destructor, 990
            exceptions, 994
            fill, 992
            init, 990
            move, 992
            rdbuf, 991
            set_rdbuf, 993
            swap, 993
tie, 991
    basic_ios<char>, 978
    basic_ios<wchar_t>, 978
    basic_iostream, 1020
        basic_iostream, 1020
            constructor, 1020, 1021
destructor, 1021
            operator=, 1021
        swap, 1021
    basic_istream, 973, 1008
        basic_istream, 1010
            constructor, 1010
destructor, 1011, 1012
            get, 1016, 1017, 1019
            operator<<, 1021
            operator=, 1011
            seekg, 1019
            swap, 1011
    basic_istream<char>, 1007
    basic_istream<wchar_t>, 1007
    basic_istreambuf_iterator, 973
INDEX OF LIBRARY NAMES

1277
INDEX OF LIBRARY NAMES 1279

operator=, 1044
str, 1045
swap, 1044
before
type_info, 451
before_begin
forward_list, 751
begin
array, 740
basic_string, 629
initializer_list, 458
match_results, 1091
valarray, 964
begin(C&), 843
begin(const pair&), 494
begin(const tuple&), 504
begin(initlist<E>), 459
begin(T (&)[N]), 843
bernoulli_distribution, 923
constructor, 923
p, 923
beta
gamma_distribution, 929
bidirectional_iterator_tag, 820
binary_function, 534, 535, 544
binary_negate, 538
binary_search, 872
bind, 539–541
bind1st, 1226
bind2nd, 1226
binder1st, 1225
binder2nd, 1226
binomial_distribution, 923
constructor, 924
p, 924
t, 924
bit_and, 538
bit_or, 538
bit_xor, 538
<bitset>, 504
bitset, 504
bitset, 506, 507
flip, 508
operator[], 509
reset, 508
set, 507
boolalpha, 994
byte_string
wstring_convert, 664
c_str

basic_string, 638
cacos
complex, 894
cacosh
complex, 894
call_once, 1149
calloc, 594, 1213
capacity
basic_string, 630
vector, 773
casin
complex, 894
casinh
complex, 894
<cassert>, 423
catan
complex, 894
catanh
complex, 894
category
error_code, 472
error_condition, 475
locale, 657
cauhcy_distribution, 934
a, 934
b, 934
constructor, 934
cbefore_begin
forward_list, 751
cbegin
basic_string, 629
cbret, 968
<complex>, 895
cend
basic_string, 629
cerr, 976
<cerrno>, 425
<cfenv>, 885
cchar_class_type
regex_traits, 1075
cchar_traits
cchar_type, 613
int_type, 613
off_type, 613
pos_type, 613
state_type, 613
cchar_type
cchar_traits, 613
chi_squared_distribution, 933
constructor, 933
n, 933
chrono, 594
<cin>, 976
classic
locale, 662
classic_table
cctype<char>, 673
clear
atomic_flag, 1127
basic_ios, 993
error_code, 472
forward_list, 752
<climits>, 1219
<locale>, 418, 1212
clog, 977
close
basic_filebuf, 1048, 1058
basic_ifstream, 1054
basic_ofstream, 1056
messages, 704
code
future_error, 1160
system_error, 477
codecvt, 674
codecvt_byname, 678
collate, 689
collate_byname, 691
combine
locale, 661
common_type, 598, 602
ccompare
atomic type, 1123
ccompare_exchange_strong
atomic type, 1123
ccompare_exchange_strong_explicit
atomic type, 1123
ccompare_exchange_weak
atomic type, 1123
ccompare_exchange_weak_explicit
atomic type, 1123
<complex>, 886
complex, 888
copy, 859
copy_backward, 860
copy_n, 859
copyfmt
basic_ios, 992
copiesign, 968
cos, 955, 968
cosh, 955, 968
count, 857
count_if, 857
cout, 976
<system_error>, 477
<condition_variable>, 1150
condition_variable
constructor, 1151
destructor, 1151
notify_all, 1152
notify_one, 1152
wait, 1152
wait_until, 1153
condition_variable_any
constructor, 1155
destructor, 1155
notify_all, 1156
notify_one, 1156
wait, 1156
wait_for, 1157
wait_until, 1156, 1157
real, 890
condition
wait, 1152, 1156
wait_for, 1153, 1154
wait_until, 1154
<condition_variable>, 1150
language reference
INDEX OF LIBRARY NAMES

...
do_get
  messages, 704
  money_get, 698
  num_get, 680, 682
  time_get, 694
do_get_date
  time_get, 694
do_get_mondayname
  time_get, 694
do_get_time
  time_get, 693
do_get_weekday
  time_get, 694
do_get_year
  time_get, 694
do_grouping
  moneypunct, 702
  numpunct, 689
do_hash
  collate, 690
do_in
  codecvt, 676
do_is
  ctype, 669
do_length
  codecvt, 677
do_max_length
  codecvt, 678
do_narrow, 673
  ctype, 670
do_neg_format
  moneypunct, 702
do_negative_sign
  moneypunct, 702
do_open
  messages, 704
do_out
  codecvt, 676
do_pos_format
  moneypunct, 702
do_positive_sign
  moneypunct, 702
do_put
  money_put, 700
  num_put, 684, 687
  time_put, 696
do_scan_is
  ctype, 670
do_scan_not
  ctype, 670
do_thousands_sep
  moneypunct, 702
  numpunct, 689
  do_tolower
  ctype, 670
  do_toupper
  ctype, 670
  do_transform
    collate, 690
do_truename
    numpunct, 689
do_unshift
    codecvt, 677
do_widen, 673
    ctype, 670
domain_error, 462
  domain_error, 462
duration
  count, 600
  duration, 600
  max, 601
  min, 601
  operator!=, 603
  operator*, 602
  operator**, 601
  operator+, 600, 605
  operator++, 600, 601
  operator+=, 601
  operator-, 600, 606
  operator=, 601
  operator-=, 601
  operator/., 602
  operator/=, 601
  operator<, 603
  operator<=, 603
  operator==, 603
  operator>=, 603
  operator%?, 602, 603
  operator%=?, 601
  zero?, 601
duration_cast, 603
duration_values, 597
  max, 598
  min, 598
  zero, 597
dynamic_pointer_cast
  shared_ptr, 584
eback
  basic_streambuf, 1003
eptr
  basic_streambuf, 1003
element_type
  pointer_traits, 555
emplace
  priority_queue, 767
emplace_after
  forward_list, 752
emplace_front
  forward_list, 751
empty
  basic_string, 630
match_results, 1090
enable_shared_from_this, 588
destructor, 589
enable_shared_from_this, 589
operator=, 589
shared_from_this, 589
encoding
  codecvt, 675
end
  array, 740
  basic_string, 629
  initializer_list, 458
  match_results, 1091
valarray, 964
end(C&), 843
end(const pair&), 495
end(const tuple&), 504
end(initializer_list<E>), 459
end(T (&)[N]), 843
endl, 1027, 1029
ends, 1029
entropy
  random_device, 917
eof
  basic_ios, 993
epptr
  basic_streambuf, 1003
epsilon
  numeric_limits, 436
eq
  char_traits, 639–641
equal, 858
  istreambuf_iterator, 841
equal_range, 872
equal_to, 537
equivalent
  error_category, 469, 470
erase
  basic_string, 635
deque, 747
list, 759
basic_string, 635
vector, 774
erase_after
  forward_list, 752
erased
  forward_list, 752
erf, 968
erfc, 968
error
  clear, 474
error_category
  default_error_condition, 469, 470
equivalent, 469, 470
message, 469
name, 469, 470
operator!=, 469
operator<, 470
operator==, 469
error_code
  assign, 472
category, 472
clear, 472
default_error_condition, 472
error_code, 471, 472
message, 472
operator bool, 472
operator!=, 476
operator<, 473
operator<<, 473
operator==, 472
operator==, 475
value, 472
error_condition
  assign, 474
category, 475
default_error_condition, 474
message, 475
operator bool, 475
operator!=, 476
operator<, 475
operator<<, 474
operator==, 475
value, 475
error_type, 1072–1074
exception
  bad_function_call, 544
  bad_weak_ptr, 577
<exception>, 452
exception
category, 453
destructor, 454
<table>
<thead>
<tr>
<th>Library Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>exception_ptr</td>
<td>456</td>
</tr>
<tr>
<td>exceptions</td>
<td>basic_ios, 994</td>
</tr>
<tr>
<td>exchange</td>
<td>atomic type, 1123</td>
</tr>
<tr>
<td>exit</td>
<td>59, 61, 131, 422, 444, 450</td>
</tr>
<tr>
<td>exp</td>
<td>955, 968</td>
</tr>
<tr>
<td>complex</td>
<td>894</td>
</tr>
<tr>
<td>exp2</td>
<td>968</td>
</tr>
<tr>
<td>expired</td>
<td>weak_ptr, 587</td>
</tr>
<tr>
<td>expml</td>
<td>968</td>
</tr>
<tr>
<td>exponential_distribution</td>
<td>927</td>
</tr>
<tr>
<td>constructor</td>
<td>928</td>
</tr>
<tr>
<td>lambda</td>
<td>928</td>
</tr>
<tr>
<td>extreme_value_distribution</td>
<td>930</td>
</tr>
<tr>
<td>a</td>
<td>930</td>
</tr>
<tr>
<td>b</td>
<td>931</td>
</tr>
<tr>
<td>constructor</td>
<td>930</td>
</tr>
<tr>
<td>facet</td>
<td>locale, 658</td>
</tr>
<tr>
<td>fail</td>
<td>basic_ios, 994</td>
</tr>
<tr>
<td>failed</td>
<td>ostreambuf_iterator, 843</td>
</tr>
<tr>
<td>failure</td>
<td>ios_base::failure, 981, 982</td>
</tr>
<tr>
<td>falsename</td>
<td>numpunct, 688</td>
</tr>
<tr>
<td>fclose</td>
<td>1049</td>
</tr>
<tr>
<td>fdim</td>
<td>968</td>
</tr>
<tr>
<td>FE_ALL_EXCEPT</td>
<td>885</td>
</tr>
<tr>
<td>FE_DFL_ENV</td>
<td>885</td>
</tr>
<tr>
<td>FE_DIVBYZERO</td>
<td>885</td>
</tr>
<tr>
<td>FE_DOWNWARD</td>
<td>885</td>
</tr>
<tr>
<td>FE_INEXACT</td>
<td>885</td>
</tr>
<tr>
<td>FE_INTEGRAL</td>
<td>885</td>
</tr>
<tr>
<td>FE_OVERFLOW</td>
<td>885</td>
</tr>
<tr>
<td>FE_TONEAREST</td>
<td>885</td>
</tr>
<tr>
<td>FE_TOWARDZERO</td>
<td>885</td>
</tr>
<tr>
<td>FE_UNDERFLOW</td>
<td>885</td>
</tr>
<tr>
<td>FECLEAVE</td>
<td>885</td>
</tr>
<tr>
<td>fesetexceptflag</td>
<td>885</td>
</tr>
<tr>
<td>fesetround</td>
<td>885</td>
</tr>
<tr>
<td>fetch</td>
<td>atomic type, 1125</td>
</tr>
<tr>
<td>fetestexcept</td>
<td>885</td>
</tr>
<tr>
<td>feupdateenv</td>
<td>885</td>
</tr>
<tr>
<td>fexcept_t</td>
<td>885</td>
</tr>
<tr>
<td>filebuf</td>
<td>973, 1045</td>
</tr>
<tr>
<td>fill</td>
<td>862</td>
</tr>
<tr>
<td>array</td>
<td>742</td>
</tr>
<tr>
<td>basic_ios</td>
<td>992</td>
</tr>
<tr>
<td>gslice_array</td>
<td>961</td>
</tr>
<tr>
<td>indirect_array</td>
<td>963</td>
</tr>
<tr>
<td>mask_array</td>
<td>962</td>
</tr>
<tr>
<td>slice_array</td>
<td>958</td>
</tr>
<tr>
<td>fill_n</td>
<td>862</td>
</tr>
<tr>
<td>find</td>
<td>855</td>
</tr>
<tr>
<td>basic_string</td>
<td>638, 639</td>
</tr>
<tr>
<td>map</td>
<td>783</td>
</tr>
<tr>
<td>multimap</td>
<td>787</td>
</tr>
<tr>
<td>find_end</td>
<td>856</td>
</tr>
<tr>
<td>find_first_not_of</td>
<td>basic_string, 640</td>
</tr>
<tr>
<td>find_first_of</td>
<td>856</td>
</tr>
<tr>
<td>basic_string</td>
<td>639</td>
</tr>
<tr>
<td>basic_string</td>
<td>640</td>
</tr>
<tr>
<td>find_if</td>
<td>855</td>
</tr>
<tr>
<td>find_if_not</td>
<td>855</td>
</tr>
<tr>
<td>find_last_not_of</td>
<td>basic_string, 641</td>
</tr>
<tr>
<td>find_last_of</td>
<td>855</td>
</tr>
<tr>
<td>basic_string</td>
<td>641</td>
</tr>
<tr>
<td>basic_string</td>
<td>640</td>
</tr>
<tr>
<td>fisher_f_distribution</td>
<td>935</td>
</tr>
<tr>
<td>constructor</td>
<td>935</td>
</tr>
<tr>
<td>m</td>
<td>935</td>
</tr>
<tr>
<td>n</td>
<td>935</td>
</tr>
<tr>
<td>fixed</td>
<td>996</td>
</tr>
<tr>
<td>flag_type</td>
<td>basic_regex, 1081</td>
</tr>
<tr>
<td>flags</td>
<td>ios_base, 667, 985</td>
</tr>
<tr>
<td>flip</td>
<td>bitset, 508</td>
</tr>
<tr>
<td>bitset</td>
<td>508</td>
</tr>
<tr>
<td>vector&lt;bool&gt;</td>
<td>777</td>
</tr>
<tr>
<td>float_denorm_style</td>
<td>numeric_limits, 437</td>
</tr>
</tbody>
</table>
float_round_style, 439
floor, 968
flush, 985, 1011, 1024, 1025, 1030
basic_ostream, 1029
fma, 968
fmax, 968
fmin, 968
fmtflags
ios, 1030
ios_base, 982, 985
fopen, 1048
for_each, 855
format
match_results, 1091, 1092
format_default, 1070
format_default, 1072
format_first_only, 1070, 1097
format_first_only, 1072
format_no_copy, 1070, 1097
format_no_copy, 1072
format_sed, 1070
format_sed, 1072
forward, 490
forward_iterator_tag, 820
<forward_list>, 738
forward_list
assign, 751
before_begin, 751
cbefore_begin, 751
clear, 752
demplace_after, 752
demplace_front, 751
derase_after, 752
derased, 752
dforward_list, 750
dfront, 751
dinsert_after, 751, 752
derange, 754
dpop, 751
dpush_front, 751
dremove, 753
dremove_if, 753
dresize, 752
dreverse, 754
dsort, 754
dsplice_after, 752, 753
dswap, 754
dunique, 753
dfpclassify, 970
dfpos, 987, 988
dstate, 988
dfree, 594
dfreeze
dostrstream, 1224
dstrstream, 1225
strstreambuf, 1219
dfexp, 968
dfrom_bytes
wstring_convert, 664
dfrom_time_t, 607
dfront
basic_string, 631
dforward_list, 751
dfront_insert_iterator, 829
dfront_insert_iterator, 829
dfront_inserter, 830
<fstream>, 1045
fstream, 973
dfunction
assign, 548
dbool conversion, 548
destructor, 548
dfunction, 547
doperator!, 549
doperator(), 548
doperator=, 547, 548
doperator==, 549
dswap, 548, 549
target, 549
target_type, 549
<functional>, 529
dfuture
dconstructor, 1165
dget, 1165
doperator=, 1165
dvalid, 1166
dwait, 1166
dwait_for, 1166
dwait_until, 1166
dfuture_category, 1159
dfuture_errc
dmake_error_code, 1160
dmake_error_condition, 1160
dfuture_error
dcode, 1160
what, 1160
gamma_distribution, 928
dalpha, 929
dbeta, 929
INDEX OF LIBRARY NAMES
INDEX OF LIBRARY NAMES

basic_istream, 1018
ilogb, 968
imag, 895
  complex, 890, 893
imbue, 1076
  basic_filebuf, 1052
  basic_ios, 991
  basic_regex, 1081
  basic_streambuf, 1003
ios_base, 986
in
  codecvt, 675
in_avail
  basic_streambuf, 1001
includes, 874
indirect_bits_engine, 913
indirect_array, 962
operator[], 963
infinity
  numeric_limits, 437
Init
  ios_base::Init, 984
init
  basic_ios, 990, 1010, 1023
initializer_list
  begin, 458
  end, 458
  initializer_list, 458
size, 459
inner_allocator
  scoped_allocator_adaptor, 562
inner_allocator_type
  scoped_allocator_adaptor, 561
inner_product, 965
inplace_merge, 873
input_iterator_tag, 820
emplace
deque, 747
insert
  basic_string, 634
deque, 747
list, 758
basic_string, 634, 635
map, 783
multimap, 787
vector, 774
push_back
deque, 747
push_front
deque, 747
insert_after
forward_list, 751, 752
insert_iterator, 830
  insert_iterator, 831
inserter, 831
int16_t, 442
int32_t, 442
int64_t, 442
int8_t, 442
int_fast16_t, 442
int_fast32_t, 442
int_fast64_t, 442
int_fast8_t, 442
int_least16_t, 442
int_least32_t, 442
int_least64_t, 442
int_least8_t, 442
int_type
  char_traits, 613
  wstring_convert, 665
internal, 995
intervals
  piecewise_constant_distribution, 940
  piecewise_linear_distribution, 942
intmax_t, 442
intptr_t, 442
invalid_argument, 462, 506
  invalid_argument, 462, 463
INVOKO, 533
<iomanip>, 1008
<iostream>, 977
ios, 973, 978
ios_base, 979
  destructor, 988
  fmtflags, 985
  ios_base, 987
  iostate, 983
  precision, 985
  setf, 985
  streamsize, 985
ios_base::failure, 981
ios_base::Init, 984
  destructor, 985
<iowsfw>, 973
iostate
  ios_base, 983
<iostream>, 975
iostream_category, 997
iota, 967
is
cctype, 669
cctype<char>, 672
is_bind_expression, 539
is_bounded
    numeric_limits, 438
is_exact
    numeric_limits, 436
is_heap, 878
is_heap_until, 878
is_iec559
    numeric_limits, 438
is_integer
    numeric_limits, 436
is_modulo
    numeric_limits, 438
is_open
    basic_filebuf, 1048, 1058
    basic_ifstream, 1054
    basic_ofstream, 1056
is_partitioned, 866
is_permutation, 858
is_signed
    numeric_limits, 436
is_sorted
    870
is_sorted_until, 870
isalnum, 662
isalpha, 662
iscntrl, 662
iscctype
    regex_traits, 1076
    Regular Expression Traits, 1105
isdigit, 662
isfinite, 970
isgraph, 662
isgreater, 970
isgreaterequal, 970
isinf, 970
isless, 970
islessequal, 970
islessgreater, 970
islower, 662
isnan, 970
isnormal, 970
<iso646.h>, 1212
isprint, 662
ispunct, 662
isspace, 662
<istream>, 1007
istream, 973, 1007
istream_iterator, 836
    constructor, 837
destructor, 837
operator!», 838
operator*, 838
operator++, 838
operator->, 838
operator==, 838
istreambuf_iterator, 840
    constructor, 841
    operator++, 841
istringstream, 973, 1034
istrstream, 1222
    constructor, 1223
    istrstream, 1222
isunordered, 970
isupper, 662
isxdigit, 662
iter_swap, 861
<iterator>, 815
iword
    ios_base, 987
join
    thread, 1134
joinable
    thread, 1134
kill_dependency, 1112
knuth_b, 916
lambda
    exponential_distribution, 928
left, 995
length
    char_traits, 627, 628, 631–634, 637, 639–641, 643–645
    basic_string, 629
codecv, 675
match_results, 1091
regex_traits, 1075
sub_match, 1082
valarray, 952
length_error, 463, 621
length_error, 463
less, 537
less_equal, 537
lexicographical_compare, 880
lgamma, 968
<limits>, 433
linear_congruential_engine, 908
    constructor, 908
<list>, 738
list, 754
assign, 758
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>list, 757</td>
<td></td>
</tr>
<tr>
<td>splice, 759</td>
<td></td>
</tr>
<tr>
<td>swap, 761</td>
<td></td>
</tr>
<tr>
<td>lrint, 968</td>
<td></td>
</tr>
<tr>
<td>lround, 968</td>
<td></td>
</tr>
<tr>
<td>load</td>
<td></td>
</tr>
<tr>
<td>atomic type, 1123</td>
<td></td>
</tr>
<tr>
<td>&lt;locale&gt;, 654</td>
<td></td>
</tr>
<tr>
<td>locale, 1076, 1081, 1104</td>
<td></td>
</tr>
<tr>
<td>destructor, 661</td>
<td></td>
</tr>
<tr>
<td>locale, 660, 661</td>
<td></td>
</tr>
<tr>
<td>operator=, 660</td>
<td></td>
</tr>
<tr>
<td>lock, 1148</td>
<td></td>
</tr>
<tr>
<td>unique_lock, 1146</td>
<td></td>
</tr>
<tr>
<td>weak_ptr, 587</td>
<td></td>
</tr>
<tr>
<td>lock_guard</td>
<td></td>
</tr>
<tr>
<td>constructor, 1143</td>
<td></td>
</tr>
<tr>
<td>destructor, 1143</td>
<td></td>
</tr>
<tr>
<td>log, 955, 968</td>
<td></td>
</tr>
<tr>
<td>complex, 894</td>
<td></td>
</tr>
<tr>
<td>log10, 955, 968</td>
<td></td>
</tr>
<tr>
<td>complex, 894</td>
<td></td>
</tr>
<tr>
<td>log1p, 968</td>
<td></td>
</tr>
<tr>
<td>log2, 968</td>
<td></td>
</tr>
<tr>
<td>logb, 968</td>
<td></td>
</tr>
<tr>
<td>logic_error, 461</td>
<td></td>
</tr>
<tr>
<td>logic_error, 462</td>
<td></td>
</tr>
<tr>
<td>logical_and, 537</td>
<td></td>
</tr>
<tr>
<td>logical_not, 538</td>
<td></td>
</tr>
<tr>
<td>logical_or, 538</td>
<td></td>
</tr>
<tr>
<td>lognormal_distribution, 932</td>
<td></td>
</tr>
<tr>
<td>constructor, 932</td>
<td></td>
</tr>
<tr>
<td>m, 932</td>
<td></td>
</tr>
<tr>
<td>s, 933</td>
<td></td>
</tr>
<tr>
<td>longjmp, 460</td>
<td></td>
</tr>
<tr>
<td>lookup_classname</td>
<td></td>
</tr>
<tr>
<td>regex_traits, 1075</td>
<td></td>
</tr>
<tr>
<td>Regular Expression Traits, 1105</td>
<td></td>
</tr>
<tr>
<td>lookup_collatename</td>
<td></td>
</tr>
<tr>
<td>regex_traits, 1075</td>
<td></td>
</tr>
<tr>
<td>Regular Expression Traits, 1105</td>
<td></td>
</tr>
<tr>
<td>lower_bound, 871</td>
<td></td>
</tr>
<tr>
<td>lowest</td>
<td></td>
</tr>
<tr>
<td>numeric_limits, 435</td>
<td></td>
</tr>
<tr>
<td>lrint, 968</td>
<td></td>
</tr>
<tr>
<td>lround, 968</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td></td>
</tr>
<tr>
<td>fisher_f_distribution, 935</td>
<td></td>
</tr>
<tr>
<td>lognormal_distribution, 932</td>
<td></td>
</tr>
<tr>
<td>make_error_code, 473, 997</td>
<td></td>
</tr>
<tr>
<td>future_errc, 1160</td>
<td></td>
</tr>
<tr>
<td>make_error_condition, 475, 997</td>
<td></td>
</tr>
<tr>
<td>future_errc, 1160</td>
<td></td>
</tr>
<tr>
<td>make_exception_ptr, 456</td>
<td></td>
</tr>
<tr>
<td>make_heap, 877</td>
<td></td>
</tr>
<tr>
<td>make_move_iterator, 836</td>
<td></td>
</tr>
<tr>
<td>make_pair, 494</td>
<td></td>
</tr>
<tr>
<td>make_ready_at_thread_exit</td>
<td></td>
</tr>
<tr>
<td>packaged_task, 1177</td>
<td></td>
</tr>
<tr>
<td>make_tuple, 500</td>
<td></td>
</tr>
<tr>
<td>malloc, 594, 1213</td>
<td></td>
</tr>
<tr>
<td>&lt;map&gt;, 777</td>
<td></td>
</tr>
<tr>
<td>map, 779</td>
<td></td>
</tr>
<tr>
<td>constructor, 782</td>
<td></td>
</tr>
<tr>
<td>find, 783</td>
<td></td>
</tr>
<tr>
<td>insert, 783</td>
<td></td>
</tr>
<tr>
<td>map, 782</td>
<td></td>
</tr>
<tr>
<td>operator&lt;, 782</td>
<td></td>
</tr>
<tr>
<td>operator==, 782</td>
<td></td>
</tr>
<tr>
<td>swap, 784</td>
<td></td>
</tr>
<tr>
<td>mark_count</td>
<td></td>
</tr>
<tr>
<td>basic_regex, 1081</td>
<td></td>
</tr>
<tr>
<td>mask_array, 961</td>
<td></td>
</tr>
<tr>
<td>operator[], 961</td>
<td></td>
</tr>
<tr>
<td>match_any, 1070</td>
<td></td>
</tr>
<tr>
<td>match_any, 1072</td>
<td></td>
</tr>
<tr>
<td>match_continuous, 1070, 1100</td>
<td></td>
</tr>
<tr>
<td>match_continuous, 1072</td>
<td></td>
</tr>
<tr>
<td>match_default, 1070</td>
<td></td>
</tr>
<tr>
<td>match_flag_type, 1070, 1071, 1105</td>
<td></td>
</tr>
<tr>
<td>match_not_bol, 1070</td>
<td></td>
</tr>
<tr>
<td>match_not_bol, 1072</td>
<td></td>
</tr>
<tr>
<td>match_not_bow, 1070</td>
<td></td>
</tr>
<tr>
<td>match_not_bow, 1072</td>
<td></td>
</tr>
<tr>
<td>match_not_eol, 1070</td>
<td></td>
</tr>
<tr>
<td>match_not_eol, 1070</td>
<td></td>
</tr>
<tr>
<td>match_not_eow, 1070</td>
<td></td>
</tr>
<tr>
<td>match_not_eow, 1072</td>
<td></td>
</tr>
<tr>
<td>match_not_null, 1070, 1100</td>
<td></td>
</tr>
<tr>
<td>match_not_null, 1072</td>
<td></td>
</tr>
<tr>
<td>match_prev_avail, 1070, 1100</td>
<td></td>
</tr>
<tr>
<td>match_prev_avail, 1072</td>
<td></td>
</tr>
<tr>
<td>match_results, 1088, 1098, 1100</td>
<td></td>
</tr>
<tr>
<td>begin, 1091</td>
<td></td>
</tr>
<tr>
<td>empty, 1090</td>
<td></td>
</tr>
<tr>
<td>end, 1091</td>
<td></td>
</tr>
<tr>
<td>format, 1091, 1092</td>
<td></td>
</tr>
<tr>
<td>get_allocator, 1092</td>
<td></td>
</tr>
<tr>
<td>length, 1091</td>
<td></td>
</tr>
<tr>
<td>match_results, 1089, 1090</td>
<td></td>
</tr>
<tr>
<td>matched, 1088</td>
<td></td>
</tr>
</tbody>
</table>
max_size, 1090
operator!=, 1093
operator=, 1090
operator==, 1092
operator[], 1091
position, 1091
prefix, 1091
size, 1090
str, 1091
suffix, 1091
swap, 1092
max, 878, 879
duration, 601
duration_values, 598
numeric_limits, 435
time_point, 605
valarray, 952
max_digits10
numeric_limits, 435
max_element, 880
max_exponent
numeric_limits, 436
max_exponent10
numeric_limits, 437
max_length
codecvt, 676
max_size
basic_string, 629
allocator, 559
allocator_traits, 557
array, 740
match_results, 1090
scoped_allocator_adaptor, 563
mean
normal_distribution, 931
poisson_distribution, 927
student_t_distribution, 936
mem_fn, 544
mem_fun, 542, 543
mem_fun1_ref_t, 542
mem_fun1_t, 542
mem_fun_ref, 543
mem_fun_ref_t, 542
mem_fun_t, 542
memchr, 650
<memory>, 550
merge, 873
list, 760
forward_list, 754
mersenne_twister_engine, 909
constructor, 910
message
error_category, 469
error_code, 472
error_condition, 475
messages, 703
messages_byname, 704
min, 878
duration, 601
duration_values, 598
numeric_limits, 435
time_point, 605
valarray, 952
min_element, 880
min_exponent
numeric_limits, 436
min_exponent10
numeric_limits, 436
minmax, 879
minmax_element, 880
minstd_rand, 915
minstd_rand0, 915
minus, 536
mismatch, 857
mod, 968
modf, 968
modulus, 536
money_get, 697
money_put, 699
moneypunct, 700
moneypunct_byname, 703
move, 490
basic_ios, 992
move move, 860
move_backward, 860
move_iterator, 832
base, 834
constructor, 833
move_iterator, 833
operator!=, 835
operator*, 834
operator+, 835, 836
operator++, 834
operator+=, 835
operator-, 835, 836
operator-=, 835
operator->, 834
operator<<, 834
operator<<=, 835
operator>=, 835
operator==, 833
operator==, 835
operator==, 835
INDEX OF LIBRARY NAMES
INDEX OF LIBRARY NAMES

operator>, 835
operator>=, 836
operator[], 835
mt19937, 915
mt19937_64, 916
multimap, 784
find, 787
insert, 787
multimap, 787
operator<, 787
operator==, 787
swap, 788
multiplies, 536
multiset, 791
multiset, 793, 794
operator<, 793
operator==, 793
swap, 794
<mutex>, 1136
mutex
    unique_lock, 1147
n
    chi_squared_distribution, 933
    fisher_f_distribution, 935
name
    type_info, 451
    error_category, 469, 470
    locale, 661
    type_index, 609
nan, 968
narrow
    basic_ios, 991
    ctype, 669
    ctype<char>, 673
NDEBUG, 423
nearbyint, 968
negate, 536
negative_binomial_distribution, 925
    constructor, 926
    p, 926
t, 926
nested_exception
    nested_exception, 457
    nested_ptr, 457
    rethrow_if_nested, 457
    returned, 457
    throw_with_nested, 457
nested_ptr
    nested_exception, 457
    <new>, 426, 445
new
    operator, 445, 448
    operator, 426, 446–449, 594
    new_handler, 450
next, 822
next_permutation, 881
nextafter, 968
nexttoward, 968
noboolalpha, 994
none
    bitset, 509
    none_of, 855
norm, 895
    complex, 893
    normal_distribution, 931
        constructor, 931
        mean, 931
        stddev, 932
noshowbase, 994
noshowpoint, 994
noshowpos, 995
noskipws, 995
not1, 538
not2, 539
not_equal_to, 537
notify_all
    condition_variable, 1152
    condition_variable_any, 1156
notify_one
    condition_variable, 1152
    condition_variable_any, 1156
nounitbuf, 995
nouppercase, 995
nth_element, 870
NULL, 433
num_get, 678
    do_get, 682
num_put, 683
    do_put, 687
<numeric>, 964
numeric_limits, 434
numeric_limits
    denorm_min, 438
digits, 435
digits10, 435
epsilon, 436
float_denorm_style, 437
has_denorm_loss, 437
has_infinity, 437
has_qaudt_NaN, 437
has_signaling_NaN, 437
infinity, 437
is_bounded, 438
is_exact, 436
is_iec559, 438
is_integer, 436
is_modulo, 438
is_signed, 436
lowest, 435
max, 435
max_digits10, 435
max_exponent, 436
max_exponent10, 437
min, 435
min_exponent, 436
min_exponent10, 436
quiet_NaN, 437
radix, 436
round_error, 436
round_style, 439
signaling_NaN, 438
tinyness_before, 438
traps, 438
numpunct, 687
numpunct_byname, 689
oct, 996
off_type
  char_traits, 613
offsetof, 1213
ofstream, 973, 1045
once_flag, 1148
open
  basic_filebuf, 1048, 1058
  basic_ifstream, 1054
  basic_ofstream, 1056
  messages, 704
openmode
  ios_base, 983
operator @=
  atomic type, 1125
operator auto_ptr
  auto_ptr_ref, 1229
operator basic_string
  sub_match, 1083
operator bool
  basic_istream, 1012
  basic_ios, 993
  basic_ostream, 1025
  error_code, 472
  error_condition, 475
  packaged_task, 1176
shared_ptr, 583
unique_lock, 1147
unique_ptr, 574
operator C
  atomic type, 1123
operator T&
  reference_wrapper, 535
operator!
  basic_ios, 993
  valarray, 951
operator!=, 489
  basic_string, 645
  pair, 493
  type_info, 451
  allocator, 559
  basic_string, 645
  bitset, 509
  complex, 892
  duration, 603
  error_category, 469
  error_code, 476
  error_condition, 476
  function, 549
  istream_iterator, 838
  istreambuf_iterator, 842
  locale, 661
  match_results, 1093
  move_iterator, 835
  queue, 763
  regex_iterator, 1099
  regex_token_iterator, 1103
  reverse_iterator, 826
  stack, 769
  sub_match, 1083–1087
  thread::id, 1132
  time_point, 606
  tuple, 503
  type_index, 609
  unique_ptr, 576
  valarray, 954, 955
operator()
  default_delete, 568, 569
  function, 548
  locale, 661
  packaged_task, 1176
  random_device, 918
  reference_wrapper, 535
operator*
  auto_ptr, 1229
  back_insert_iterator, 828
  complex, 892
<table>
<thead>
<tr>
<th>Operator</th>
<th>Library Names</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>operator*=</code></td>
<td><code>basic_string</code>, <code>complex</code>, <code>duration</code>, <code>gslice_array</code>, <code>indirect_array</code>, <code>mask_array</code>, <code>move_iterator</code>, <code>reverse_iterator</code>, <code>slice_array</code>, <code>time_point</code>, <code>valarray</code></td>
</tr>
<tr>
<td><code>operator+=</code></td>
<td><code>basic_string</code>, <code>complex</code>, <code>duration</code>, <code>gslice_array</code>, <code>indirect_array</code>, <code>mask_array</code>, <code>move_iterator</code>, <code>reverse_iterator</code>, <code>slice_array</code>, <code>time_point</code>, <code>valarray</code></td>
</tr>
<tr>
<td><code>operator-=</code></td>
<td><code>basic_string</code>, <code>complex</code>, <code>duration</code>, <code>gslice_array</code>, <code>indirect_array</code>, <code>mask_array</code>, <code>move_iterator</code>, <code>reverse_iterator</code>, <code>slice_array</code>, <code>time_point</code>, <code>valarray</code></td>
</tr>
<tr>
<td><code>operator-&gt;</code></td>
<td><code>auto_ptr</code>, <code>basic_string</code>, <code>complex</code>, <code>duration</code>, <code>gslice_array</code>, <code>indirect_array</code>, <code>mask_array</code>, <code>move_iterator</code>, <code>reverse_iterator</code>, <code>slice_array</code>, <code>time_point</code>, <code>valarray</code></td>
</tr>
<tr>
<td><code>operator++</code></td>
<td><code>atomic type</code>, <code>back_insert_iterator</code>, <code>complex</code>, <code>duration</code>, <code>front_insert_iterator</code>, <code>insert_iterator</code>, <code>istream_iterator</code>, <code>istreambuf_iterator</code>, <code>move_iterator</code>, <code>ostream_iterator</code>, <code>ostreambuf_iterator</code>, <code>raw_storage_iterator</code>, <code>regex_iterator</code>, <code>regex_token_iterator</code>, <code>reverse_iterator</code>, <code>shared_ptr</code>, <code>unique_ptr</code>, <code>valarray</code></td>
</tr>
<tr>
<td><code>operator++</code></td>
<td><code>atomic type</code>, <code>back_insert_iterator</code>, <code>complex</code>, <code>duration</code>, <code>front_insert_iterator</code>, <code>insert_iterator</code>, <code>istream_iterator</code>, <code>istreambuf_iterator</code>, <code>move_iterator</code>, <code>ostream_iterator</code>, <code>ostreambuf_iterator</code>, <code>raw_storage_iterator</code>, <code>regex_iterator</code>, <code>regex_token_iterator</code>, <code>reverse_iterator</code>, <code>shared_ptr</code>, <code>unique_ptr</code>, <code>valarray</code></td>
</tr>
</tbody>
</table>

INDEX OF LIBRARY NAMES

1293
slice_array, 957
valarray, 951
operator<
  basic_string, 645
  pair, 493
  basic_string, 645
duration, 603
error_category, 470
error_code, 473
error_condition, 475
move_iterator, 835
queue, 763
reverse_iterator, 826
shared_ptr, 584
stack, 769
sub_match, 1083–1087
thread::id, 1132
time_point, 606
tuple, 503
type_index, 609
unique_ptr, 576
valarray, 954, 955
operator<<
  shared_ptr, 584
  sub_match, 1087
operator<<=
  basic_string, 645
  pair, 493
  basic_string, 645
duration, 603
move_iterator, 835
queue, 763
reverse_iterator, 826
shared_ptr, 584
stack, 769
sub_match, 1083–1087
thread::id, 1132
time_point, 606
tuple, 503
type_index, 609
unique_ptr, 576
valarray, 954, 955
operator<<=
  basic_string, 645
complex, 507
operator<<=, 489
  basic_string, 645
  pair, 493
  basic_string, 645
duration, 603
move_iterator, 835
queue, 763
reverse_iterator, 827
shared_ptr, 576
stack, 769
sub_match, 1083–1087
thread::id, 1132
time_point, 606
tuple, 503
type_index, 609
valarray, 954, 955
operator<<
  basic_istream, 1021
  basic_ostringstream, 1042
  basic_regex, 1080
  basic_string, 627
  reverse_iterator, 824
atomic_type, 1123
atomic_future, 1171
auto_ptr, 1228
auto_ptr_ref, 1229
back_insert_iterator, 828
basic_filebuf, 1048
basic_ifstream, 1053
basic_istream, 1021
basic_iostream, 1021
basic_regex, 1080
basic_stringbuf, 1036
basic_stringstream, 1044
enable_shared_from_this, 589
error_code, 472
error_condition, 474
future, 1165
future::value, 1165
gslice_array, 960
indirect_array, 963
mask_array, 962
move_iterator, 835
move_iterator, 833
narrow_int, 1203
ostream_iterator, 839
ostreambuf_iterator, 843
sub_match, 1087
valarray, 951
valarray, 957
valarray, 954
valarray, 954
valarray, 955
unique_ptr, 576
unordered_map, 799
unordered_multimap, 803
valarray, 948–950
operator%, 602, 603
valarray, 953, 954
operator%= duration, 601
gslice_array, 960
indirect_array, 963
mask_array, 962
slice_array, 957
valarray, 951
operator& bitset, 510
valarray, 953, 954
operator&= bitset, 507
gslice_array, 960
indirect_array, 963
mask_array, 962
slice_array, 957
valarray, 951
operator&& valarray, 953–955
operator~ bitset, 510
valarray, 953, 954
operator-= bitset, 507
gslice_array, 960
indirect_array, 963
mask_array, 962
slice_array, 957
valarray, 951
operator~ bitset, 508
valarray, 951
operator| bitset, 510
valarray, 953, 954
operator|= bitset, 507
gslice_array, 960
indirect_array, 963
mask_array, 962
slice_array, 957
valarray, 951
operator|| valarray, 953–955
<ostream>, 1007
ostream, 973, 1008
ostream_iterator, 838
constructor, 839
destructor, 839
operator*, 839
operator++, 839
operator=, 839
ostreambuf_iterator, 842
constructor, 842, 843
ostreamstream, 973, 1034
ostreamstream, 1223
constructor, 1223
ostrstream, 1223
out
codecvt, 675
out_of_range, 463, 506, 508, 509, 621
out_of_range, 463
outer_allocator
scoped_allocator_adaptor, 562, 563
output_iterator_tag, 820
overflow
basic_filebuf, 1050
basic_streambuf, 1006
basic_stringbuf, 1038
strstreambuf, 1220
overflow_error, 464, 465, 506, 508
overflow_error, 465
owner_before
shared_ptr, 583, 587
owns_lock
unique_lock, 1147
P
beroulli_distribution, 923
binomial_distribution, 924
gemetric_distribution, 925
negative_binomial_distribution, 926
pack_arguments, 501
packaged_task
constructor, 1175
destructor, 1175
get_future, 1176
make_ready_at_thread_exit, 1177
operator bool, 1176
operator(), 1176
operator=, 1175
reset, 1177
swap, 1176, 1177
pair, 492, 498, 500
get, 494
INDEX OF LIBRARY NAMES
INDEX OF LIBRARY NAMES

pubimbue
  basic_streambuf, 1000
pubseekoff
  basic_streambuf, 1001
pubseekpos
  basic_streambuf, 1001
pubsetbuf
  basic_streambuf, 1001
pubsync
  basic_streambuf, 1001
push
  priority_queue, 766
push_back
  basic_string, 632
push_front
  forward_list, 751
push_heap, 876
put
  basic_ostream, 1029
  money_put, 699
  num_put, 684
  time_put, 696
put_money, 1032
put_time, 1033
putback
  basic_istream, 1019
pword
  ios_base, 987
<queue>, 739
queue, 761
swap, 764
quick_exit, 444
quiet_NaN
  numeric_limits, 437
radix
  numeric_limits, 436
<random>, 905–907
random_access_iterator_tag, 820
random_device, 917
  constructor, 917
  entropy, 917
  operator(), 918
random_shuffle, 866
range_error
  range_error, 464
  range_error, 464
ranlux24, 916
ranlux24_base, 916
ranlux48, 916
ranlux48_base, 916
ratio, 511
ratio_add, 512
ratio_divide, 512
ratio_equal, 512
ratio_greater, 513
ratio_greater_equal, 513
ratio_less, 513
ratio_less_equal, 513
ratio_multiply, 512
ratio_not_equal, 513
ratio_subtract, 512
raw_storage_iterator
  operator*, 565
  operator++, 565
  operator=, 565
  raw_storage_iterator, 565
rbegin
  basic_string, 629
rdbuf
  basic_filebuf, 1058
  basic_ifstream, 1054
  basic_istream, 991
  basic_istringstream, 1041
  basic_ifstream, 1056
  basic_ofstream, 1043
  basic_stringstream, 1045
  istream, 1223
  ostream, 1224
  strstream, 1225
  wbuffer_convert, 667
rdstate
  basic_istream, 1018
read
  basic_istream, 1018
readsome
  basic_istream, 1018
real, 895
complex, 890, 893
realloc, 594, 1213
rebind
  pointer_traits, 555
rebind_alloc
  allocator_traits, 557
ref
  reference_wrapper, 536
reference_wrapper, 534
cref, 536
get, 535
operator T&, 535
operator(), 535
operator=, 535
ref, 536
  reference_wrapper, 535
<regex>, 1063
regex, 1063
regex_constants, 1069
  error_type, 1072–1074
  match_flag_type, 1070
  syntax_option_type, 1070
regex_error, 1074, 1076, 1105
  constructor, 1074
regex_iterator, 1098
  increment, 1099
  operator!=, 1099
  operator*, 1099
  operator++, 1099, 1100
  operator->, 1099
  operator==, 1099
regex_token_iterator, 1099
regex_match, 1093, 1094
regex_replace, 1096, 1097
regex_search, 1095, 1096
regex_token_iterator, 1100
  end-of-sequence, 1101
  operator!=, 1103
  operator*, 1103
  operator++, 1103
  operator->, 1103
  operator==, 1101, 1103
regex_token_iterator, 1102
regex_traits, 1074
  char_class_type, 1075
  isctype, 1076
  length, 1075
  lookup_classname, 1075
  lookup_collatename, 1075
  transform, 1075
  transform_primary, 1075
  translate, 1075
  translate_nocase, 1075
  value, 1076
register_callback
ios_base, 987
Regular Expression Traits
  isctype, 1105
  lookup_classname, 1105
  lookup_collatename, 1105
  transform_primary, 1105
rel_ops, 488
release
  auto_ptr, 1229
  unique_lock, 1147
unique_ptr, 574
remainder, 968
remove, 863
  list, 760
  forward_list, 753
remove_copy, 863
remove_copy_if, 863
remove_if, 863
  forward_list, 753
remquo, 968
rend
  basic_string, 629
rep
  system_clock, 607
replace, 862
  basic_string, 636
  basic_string, 636, 637
replace_copy, 862
replace_copy_if, 862
replace_if, 862
reserve
  basic_string, 630
  vector, 773
reset
  auto_ptr, 1229
  bitset, 508
  packaged_task, 1177
  shared_ptr, 582
  unique_ptr, 574, 576
  weak_ptr, 587
resetiosflags, 1030
resize
  basic_string, 629
  deque, 746
  list, 758
  basic_string, 630
  forward_list, 752
  valarray, 953
  vector, 773
rethrow_exception, 456
rethrow_if_nested
  nested_exception, 457
returned
  nested_exception, 457
reverse, 865
  list, 761
  forward_list, 754
reverse_copy, 865
reverse_iterator, 822
  reverse_iterator, 824
  base, 824

INDEX OF LIBRARY NAMES
INDEX OF LIBRARY NAMES

constructor, 824
operator++, 825
operator--, 825
rfind
   basic_string, 639
   basic_string, 639
right, 996
rint, 968
rotate, 865
rotate_copy, 865
round, 968
round_error
   numeric_limits, 436
round_style
   numeric_limits, 439
runtime_error, 464
   runtime_error, 464
s
   lognormal_distribution, 933
sbumpc
   basic_streambuf, 1001
scalbn, 968
scalbn, 968
scan_is
   ctype, 669
   ctype<char>, 673
scan_not
   ctype, 669
   ctype<char>, 673
scientific, 996
scoped_allocator_adaptor
   allocate, 563
   constructor, 562, 563
deallocate, 563
destructor, 564
inner_allocator, 562
inner_allocator_type, 561
max_size, 563
outer_allocator, 562, 563
propagate_on_container_copy_assignment, 561
propagate_on_container_move_assignment, 561
propagate_on_container_swap, 562
search, 858
search_n, 859
seed_seq
   constructor, 918, 919
generate, 919
param, 920
size, 919
seekdir
ios_base, 984
seekg
   basic_istream, 1019
seekoff
   basic_filebuf, 1051
   basic_streambuf, 1004
   basic_stringbuf, 1038
   stringstream, 1221
seekp
   basic_ostream, 1025
seekpos
   basic_filebuf, 1051
   basic_streambuf, 1004
   basic_stringbuf, 1039
   stringstream, 1222
select_on_container_copy_construction
   allocator_traits, 557
sentry
   basic_istream, 1011
   basic_ostream, 1024
   constructor, 1011, 1024
<set>, 778
set, 788
   bitset, 507
   operator<, 790
   operator==, 790
   set, 790
   swap, 791
set_difference, 875
set_exception
   promise, 1163
set_exception_at_thread_exit
   promise, 1164
set_intersection, 874
set_new_handler, 426, 450
set_rdbuf
   basic_ios, 993
set_symmetric_difference, 876
set_terminate, 426, 455
set_unexpected, 426, 455
set_union, 874
set_value
   promise, 1162
set_value_at_thread_exit
   promise, 1163
setbase, 1031
setbuf
   basic_filebuf, 1051
   basic_streambuf, 1004, 1039
   stringstream, 1222
   stringstream, 1222
<table>
<thead>
<tr>
<th>Set Function</th>
<th>Library Name</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>setf</td>
<td>ios_base, 1031</td>
<td>985</td>
</tr>
<tr>
<td>setfill</td>
<td>basic_streambuf, 1003</td>
<td>1219</td>
</tr>
<tr>
<td>setg</td>
<td>strstreambuf, 1003</td>
<td>2125</td>
</tr>
<tr>
<td>setiosflags</td>
<td>basic_streambuf, 1003</td>
<td>1030</td>
</tr>
<tr>
<td>setjmp</td>
<td>basic_streambuf, 1003</td>
<td>425</td>
</tr>
<tr>
<td>setlocale</td>
<td>basic_streambuf, 1003</td>
<td>418</td>
</tr>
<tr>
<td>setp</td>
<td>basic_streambuf, 1003</td>
<td>1219</td>
</tr>
<tr>
<td>setprecision</td>
<td>basic_streambuf, 1003</td>
<td>1031</td>
</tr>
<tr>
<td>setstate</td>
<td>basic_streambuf, 1003</td>
<td>993</td>
</tr>
<tr>
<td>setw</td>
<td>basic_streambuf, 1003</td>
<td>1032</td>
</tr>
<tr>
<td>sgetc</td>
<td>basic_streambuf, 1001</td>
<td>1003</td>
</tr>
<tr>
<td>sgetn</td>
<td>basic_streambuf, 1001</td>
<td>1219</td>
</tr>
<tr>
<td>shared_from_this</td>
<td>enable_shared_from_this, 589</td>
<td>589</td>
</tr>
<tr>
<td>shared_future</td>
<td>constructor, 1167, 1168</td>
<td>1167</td>
</tr>
<tr>
<td></td>
<td>destructor, 1168</td>
<td>1168</td>
</tr>
<tr>
<td>get</td>
<td>operator=, 1168</td>
<td>1168</td>
</tr>
<tr>
<td></td>
<td>valid, 1169</td>
<td>1169</td>
</tr>
<tr>
<td></td>
<td>wait, 1169</td>
<td>1169</td>
</tr>
<tr>
<td></td>
<td>wait_for, 1169</td>
<td>1169</td>
</tr>
<tr>
<td></td>
<td>wait_until, 1169</td>
<td>1169</td>
</tr>
<tr>
<td>shared_ptr</td>
<td>577, 589</td>
<td>1167</td>
</tr>
<tr>
<td>~shared_ptr</td>
<td>581</td>
<td>581</td>
</tr>
<tr>
<td>atomic_compare_exchange_strong</td>
<td>591</td>
<td>591</td>
</tr>
<tr>
<td>atomic_compare_exchange_strong_explicit</td>
<td>591</td>
<td>591</td>
</tr>
<tr>
<td>atomic_compare_exchange_weak</td>
<td>591</td>
<td>591</td>
</tr>
<tr>
<td>atomic_compare_exchange_weak_explicit</td>
<td>591</td>
<td>591</td>
</tr>
<tr>
<td>atomic_exchange</td>
<td>591</td>
<td>591</td>
</tr>
<tr>
<td>atomic_exchange_explicit</td>
<td>591</td>
<td>591</td>
</tr>
<tr>
<td>atomic_is_lock_free</td>
<td>590</td>
<td>590</td>
</tr>
<tr>
<td>atomic_load</td>
<td>590</td>
<td>590</td>
</tr>
<tr>
<td>atomic_load_explicit</td>
<td>590</td>
<td>590</td>
</tr>
<tr>
<td>atomic_store</td>
<td>590</td>
<td>590</td>
</tr>
<tr>
<td>atomic_store_explicit</td>
<td>590</td>
<td>590</td>
</tr>
<tr>
<td>const_pointer_cast</td>
<td>585</td>
<td>585</td>
</tr>
<tr>
<td>dynamic_pointer_cast</td>
<td>584</td>
<td>584</td>
</tr>
<tr>
<td>get</td>
<td>582</td>
<td>582</td>
</tr>
<tr>
<td>get_deleter</td>
<td>585</td>
<td>585</td>
</tr>
<tr>
<td>operator bool</td>
<td>583</td>
<td>583</td>
</tr>
<tr>
<td>operator*</td>
<td>582</td>
<td>582</td>
</tr>
<tr>
<td>operator-&gt;</td>
<td>582</td>
<td>582</td>
</tr>
<tr>
<td>operator&lt;</td>
<td>584</td>
<td>584</td>
</tr>
<tr>
<td>operator&lt;&lt;</td>
<td>basic_string, 630</td>
<td>630</td>
</tr>
<tr>
<td></td>
<td>deque, 746</td>
<td>746</td>
</tr>
<tr>
<td></td>
<td>vector, 773</td>
<td>773</td>
</tr>
<tr>
<td>shuffle</td>
<td>866</td>
<td>866</td>
</tr>
<tr>
<td>shuffle_order_engine</td>
<td>915</td>
<td>915</td>
</tr>
<tr>
<td>signbit</td>
<td>970</td>
<td>970</td>
</tr>
<tr>
<td>sin</td>
<td>955, 968</td>
<td>955, 968</td>
</tr>
<tr>
<td>sinh</td>
<td>955, 968</td>
<td>955, 968</td>
</tr>
<tr>
<td>complex</td>
<td>894</td>
<td>894</td>
</tr>
<tr>
<td>size_t</td>
<td>106</td>
<td>106</td>
</tr>
<tr>
<td>size_type</td>
<td>557</td>
<td>557</td>
</tr>
<tr>
<td>allocator_traits</td>
<td>1136</td>
<td>1136</td>
</tr>
<tr>
<td>skipws</td>
<td>995</td>
<td>995</td>
</tr>
<tr>
<td>sleep_for</td>
<td>this_thread, 1136</td>
<td>1136</td>
</tr>
<tr>
<td>sleep_until</td>
<td>this_thread, 1136</td>
<td>1136</td>
</tr>
<tr>
<td>slice</td>
<td>956</td>
<td>956</td>
</tr>
</tbody>
</table>

INDEX OF LIBRARY NAMES 1301
slice, 956
slice_array, 957
snextc
  basic_streambuf, 1001
sort, 868
  list, 761
  forward_list, 754
sort_heap, 877
splice
  list, 759
  list, 759
splice_after
  forward_list, 752, 753
sputbackc
  basic_streambuf, 1002
sputc
  basic_streambuf, 1002
spuwn
  basic_streambuf, 1002
sqrt, 955, 968
  complex, 895
<string>, 1034
stable_partition, 867
stable_sort, 869
<stack>, 739
stack, 767
  swap, 769
start
  gslice, 959
  slice, 956
state
  fpos, 988
  wbuffer_convert, 667
  wstring_convert, 665
state_type
  char_traits, 613
  wbuffer_convert, 667
  wstring_convert, 665
static_pointer_cast
  shared_ptr, 584
stddev
  normal_distribution, 932
<stdexcept>, 461
<stdlib.h>, 1214
stod, 649
stof, 648, 649
stoi, 648, 649
stol, 648, 649
stold, 648, 649
stoll, 648, 649
store
  atomic type, 1123
stoul, 648, 649
stoull, 648, 649
str
  basic_istringstream, 1041
  basic_ostringstream, 1043
  basic_stringbuf, 1037
  basic_stringstream, 1045
  istream, 1223
  match_results, 1091
  ostringstream, 1224
  stringstream, 1225
  streambuf, 1219
  sub_match, 1083
strchr, 650
<string>, 997
streambuf, 973, 997
streamoff, 978, 988, 1215
streamsize, 978
  ios_base, 985
strftime, 696
stride
  gslice, 959
  slice, 956
<string>, 617
stringbuf, 973, 1034
stringstream, 973
strlen, 1218, 1219, 1224
strpbrk, 650
strstr, 650
strstream, 650
strstream
  destructor, 1225
  stringstream, 1225
strstreambuf, 1216, 1218
  stringstream, 1218
  destructor, 1219
  setg, 1219
student_t_distribution, 936
  constructor, 936
  mean, 936
sub_match, 1082
  compare, 1083
  length, 1082
  operator basic_string, 1083
  operator! =, 1083–1087
  operator<, 1083–1087
  operator<<, 1087
  operator<=, 1083–1087
  operator==, 1083–1087
  operator>, 1083–1087
  operator>=, 1083–1087
INDEX OF LIBRARY NAMES
## INDEX OF LIBRARY NAMES

<table>
<thead>
<tr>
<th>Library Name</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator&gt;=</td>
<td>1083–1087</td>
</tr>
<tr>
<td>str</td>
<td>1083</td>
</tr>
<tr>
<td>substr</td>
<td>641</td>
</tr>
<tr>
<td>subtract_with_carry_engine</td>
<td>910</td>
</tr>
<tr>
<td>constructor</td>
<td>911</td>
</tr>
<tr>
<td>suffix</td>
<td>match_results, 1091</td>
</tr>
<tr>
<td>sum</td>
<td>valarray, 952</td>
</tr>
<tr>
<td>sungetc</td>
<td>basic_stringbuf, 1002</td>
</tr>
<tr>
<td>swap</td>
<td>490</td>
</tr>
<tr>
<td>basic_string</td>
<td>638, 646</td>
</tr>
<tr>
<td>pair</td>
<td>494</td>
</tr>
<tr>
<td>array</td>
<td>742</td>
</tr>
<tr>
<td>basic_filebuf</td>
<td>1048</td>
</tr>
<tr>
<td>basic_fstream</td>
<td>1054</td>
</tr>
<tr>
<td>basic_ifstream</td>
<td>1058</td>
</tr>
<tr>
<td>basic_ios</td>
<td>993</td>
</tr>
<tr>
<td>basic_iostream</td>
<td>1021</td>
</tr>
<tr>
<td>basic_istream</td>
<td>1011</td>
</tr>
<tr>
<td>basic_istringstream</td>
<td>1041</td>
</tr>
<tr>
<td>basic_ostream</td>
<td>1024</td>
</tr>
<tr>
<td>basic_ofstream</td>
<td>1056</td>
</tr>
<tr>
<td>basic_ostringstream</td>
<td>1044</td>
</tr>
<tr>
<td>basic_regex</td>
<td>1082</td>
</tr>
<tr>
<td>basic_streambuf</td>
<td>1002</td>
</tr>
<tr>
<td>basic_stringbuf</td>
<td>1036, 1037</td>
</tr>
<tr>
<td>basic_stringstream</td>
<td>1044</td>
</tr>
<tr>
<td>deque</td>
<td>747</td>
</tr>
<tr>
<td>forward_list</td>
<td>754</td>
</tr>
<tr>
<td>function</td>
<td>548, 549</td>
</tr>
<tr>
<td>list</td>
<td>761</td>
</tr>
<tr>
<td>map</td>
<td>784</td>
</tr>
<tr>
<td>match_results</td>
<td>1092</td>
</tr>
<tr>
<td>multimap</td>
<td>788</td>
</tr>
<tr>
<td>multiset</td>
<td>794</td>
</tr>
<tr>
<td>packaged_task</td>
<td>1176, 1177</td>
</tr>
<tr>
<td>pair</td>
<td>493</td>
</tr>
<tr>
<td>priority_queue</td>
<td>767</td>
</tr>
<tr>
<td>promise</td>
<td>1162, 1164</td>
</tr>
<tr>
<td>queue</td>
<td>764</td>
</tr>
<tr>
<td>set</td>
<td>791</td>
</tr>
<tr>
<td>shared_ptr</td>
<td>582, 584</td>
</tr>
<tr>
<td>stack</td>
<td>769</td>
</tr>
<tr>
<td>thread</td>
<td>1134, 1135</td>
</tr>
<tr>
<td>tuple</td>
<td>500</td>
</tr>
<tr>
<td>unique_lock</td>
<td>1147</td>
</tr>
<tr>
<td>unique_ptr</td>
<td>574</td>
</tr>
<tr>
<td>unordered_map</td>
<td>800</td>
</tr>
<tr>
<td>unordered_multimap</td>
<td>803</td>
</tr>
<tr>
<td>unordered_multiset</td>
<td>809</td>
</tr>
<tr>
<td>unordered_set</td>
<td>806</td>
</tr>
<tr>
<td>valarray</td>
<td>952, 956</td>
</tr>
<tr>
<td>vector</td>
<td>773, 775</td>
</tr>
<tr>
<td>vector&lt;bool&gt;</td>
<td>777</td>
</tr>
<tr>
<td>weak_ptr</td>
<td>587, 588</td>
</tr>
<tr>
<td>swap(unique_ptr&amp;, unique_ptr&amp;)</td>
<td>576</td>
</tr>
<tr>
<td>swap_ranges</td>
<td>861</td>
</tr>
<tr>
<td>sync</td>
<td>basic_filebuf, 1052</td>
</tr>
<tr>
<td>basic_istream</td>
<td>1019</td>
</tr>
<tr>
<td>basic_streambuf</td>
<td>1004</td>
</tr>
<tr>
<td>sync_with_stdio</td>
<td>ios_base, 986</td>
</tr>
<tr>
<td>syntax_option_type</td>
<td>awk, 1070</td>
</tr>
<tr>
<td>basic</td>
<td>1070</td>
</tr>
<tr>
<td>collate</td>
<td>1070, 1105</td>
</tr>
<tr>
<td>ECMAScript</td>
<td>1070</td>
</tr>
<tr>
<td>egrep</td>
<td>1070</td>
</tr>
<tr>
<td>extended</td>
<td>1070</td>
</tr>
<tr>
<td>grep</td>
<td>1070</td>
</tr>
<tr>
<td>icase</td>
<td>1070</td>
</tr>
<tr>
<td>nosubs</td>
<td>1070</td>
</tr>
<tr>
<td>optimize</td>
<td>1070</td>
</tr>
<tr>
<td>system_category</td>
<td>470</td>
</tr>
<tr>
<td>system_clock</td>
<td>rep, 607</td>
</tr>
<tr>
<td>system_error</td>
<td>code, 477</td>
</tr>
<tr>
<td>system_error</td>
<td>476, 477</td>
</tr>
<tr>
<td>what</td>
<td>477</td>
</tr>
<tr>
<td>t</td>
<td>binomial_distribution, 924</td>
</tr>
<tr>
<td>negative_binomial_distribution</td>
<td>926</td>
</tr>
<tr>
<td>table</td>
<td>ctype&lt;char&gt;, 673</td>
</tr>
<tr>
<td>tan</td>
<td>955, 968</td>
</tr>
</tbody>
</table>

© ISO/IEC N3092
INDEX OF LIBRARY NAMES
numpunct, 688
trunc, 968
try_lock, 1148
    unique_lock, 1146
try_lock_for
    unique_lock, 1146
try_lock_until
    unique_lock, 1146
	<tuple>, 495
tuple, 495, 496, 742
get, 502
make_tuple, 500
operator! =, 503
operator <, 503
operator <=, 503
operator ==, 503
operator >=, 503
operator >, 503
operator >=, 503
pack_arguments, 501
swap, 500
tie, 501
tuple, 497, 498
tuple_cat, 501, 502
tuple_element, 494, 502, 743
tuple_size, 494, 502, 742
type_index
    hash_code, 609
    name, 609
    operator! =, 609
    operator <, 609
    operator <=, 609
    operator ==, 609
    operator >, 609
    operator >=, 609
    type_index, 609
type_info, 98, 450
type_info::name
    implementation-defined, 451
	<typeinfo>, 450, 608
uflow
    basic_filebuf, 1049
    basic_streambuf, 1005
uint16_t, 442
uint32_t, 442
uint64_t, 442
uint8_t, 442
uint_fast16_t, 442
uint_fast32_t, 442
uint_fast64_t, 442
uint_fast8_t, 442
uint_least16_t, 442
uint_least32_t, 442
uint_least64_t, 442
uint_least8_t, 442
uintmax_t, 442
uintptr_t, 442
unary_function, 533, 534, 544
unary_negate, 538
uncaught_exception, 456
undeclare_no_pointers, 593
undeclare_reachable, 592
underflow
    basic_filebuf, 1049
    basic_streambuf, 1005
    basic_stringbuf, 1037
    strstreambuf, 1220
underflow_error
    underflow_error, 465
unexpected, 455
unexpected_handler, 426, 454
unget
    basic_istream, 1019
uniform_int_distribution, 921
    a, 921
    b, 921
    constructor, 921
uniform_real_distribution, 922
    a, 922
    b, 922
    constructor, 922
uninitialized_copy, 566
uninitialized_copy_n, 566
uninitialized_fill, 566
uninitialized_fill_n, 567
unique, 864
    list, 760
    forward_list, 753
    shared_ptr, 583
unique_copy, 864
unique_lock
    constructor, 1144, 1145
destructor, 1146
lock, 1146
mutex, 1147
operator bool, 1147
operator =, 1146
owns_lock, 1147
release, 1147
swap, 1147
try_lock, 1146

INDEX OF LIBRARY NAMES
wstring_convert, 665
wstringbuf, 973, 1034
wstringstream, 973

xalloc
  ios_base, 987
xsgetn
  basic_streambuf, 1004
xspun
  basic_streambuf, 1006

yield
  this_thread, 1136

zero
  duration, 601
  duration_values, 597
## Index of Implementation Defined Behavior

The entries in this section are rough descriptions; exact specifications are at the indicated page in the general text.

<table>
<thead>
<tr>
<th>Entry</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#pragma</code></td>
<td>407</td>
</tr>
<tr>
<td>additional formats for <code>time_get::do_get_date</code></td>
<td>694</td>
</tr>
<tr>
<td>alignment</td>
<td>76</td>
</tr>
<tr>
<td>alignment additional values</td>
<td>76</td>
</tr>
<tr>
<td>alignment of bit-fields within a class object</td>
<td>222</td>
</tr>
<tr>
<td>allocation of bit-fields within a class object</td>
<td>222</td>
</tr>
<tr>
<td>argument values to construct <code>basic_ios::failure</code></td>
<td>993</td>
</tr>
<tr>
<td>assignability of placeholder objects</td>
<td>541</td>
</tr>
<tr>
<td>behavior of attribute scoped token</td>
<td>170</td>
</tr>
<tr>
<td>behavior of iostream classes when <code>traits::pos_type</code> is not <code>streampos</code> or when <code>traits::off_type</code> is not <code>streamoff</code></td>
<td>973</td>
</tr>
<tr>
<td>behavior of non-standard attributes</td>
<td>171</td>
</tr>
<tr>
<td>bits in a byte</td>
<td>5</td>
</tr>
<tr>
<td>choice of larger or smaller value of floating literal</td>
<td>26</td>
</tr>
<tr>
<td>concatenation of some types of string literals</td>
<td>27</td>
</tr>
<tr>
<td>conversions between pointers and integers</td>
<td>101</td>
</tr>
<tr>
<td>converting characters from source character set to execution character set</td>
<td>16</td>
</tr>
<tr>
<td>converting pointer to function into pointer to object and vice versa</td>
<td>102</td>
</tr>
<tr>
<td>defining <code>main</code> in freestanding environment</td>
<td>58</td>
</tr>
<tr>
<td>definition and meaning of <code>_STDC_</code></td>
<td>408</td>
</tr>
<tr>
<td>definition and meaning of <code>_STDC_VERSION_</code></td>
<td>408</td>
</tr>
<tr>
<td>derived type for <code>typeid</code></td>
<td>98</td>
</tr>
<tr>
<td>diagnostic message</td>
<td>2</td>
</tr>
<tr>
<td>distinctness of string literals</td>
<td>27</td>
</tr>
<tr>
<td>dynamic initialization of static objects before <code>main</code></td>
<td>60</td>
</tr>
<tr>
<td>dynamic initialization of thread-local objects before entry</td>
<td>60</td>
</tr>
<tr>
<td>effect of <code>array::front()</code> and <code>array::back()</code> on zero-sized array</td>
<td>742</td>
</tr>
<tr>
<td>effect of calling <code>basic_filebuf::setbuf</code> with non-zero arguments</td>
<td>1051</td>
</tr>
<tr>
<td>effect of calling <code>basic_filebuf::sync</code> when a get area exists</td>
<td>1052</td>
</tr>
<tr>
<td>effect of calling <code>basic_streambuf::setbuf</code> with non-zero arguments</td>
<td>1039</td>
</tr>
<tr>
<td>effect of calling <code>ios_base::sync_with_stdio</code> after any input or output operation on standard streams</td>
<td>986</td>
</tr>
<tr>
<td>effect on C locale of calling <code>locale::global</code></td>
<td>662</td>
</tr>
<tr>
<td>encoding of universal character name not in execution character set</td>
<td>25</td>
</tr>
<tr>
<td>error category for errors originating outside the operating system</td>
<td>431</td>
</tr>
<tr>
<td>exception type when <code>shared_ptr</code> constructor fails</td>
<td>579–581</td>
</tr>
<tr>
<td>exceptions thrown by standard library functions that do not have an exception specification</td>
<td>430</td>
</tr>
<tr>
<td>execution character-set and execution wide-character set</td>
<td>17</td>
</tr>
<tr>
<td>exit status</td>
<td>444</td>
</tr>
<tr>
<td>extended signed integer types</td>
<td>71</td>
</tr>
<tr>
<td>formatted character sequence generated by <code>time_put::do_put</code> in C locale</td>
<td>696</td>
</tr>
<tr>
<td>headers for freestanding implementation</td>
<td>422</td>
</tr>
<tr>
<td>interactive device</td>
<td>8</td>
</tr>
<tr>
<td>linkage of <code>main</code></td>
<td>58</td>
</tr>
<tr>
<td>linkage of names from Standard C library</td>
<td>423</td>
</tr>
<tr>
<td>locale names</td>
<td>660</td>
</tr>
<tr>
<td>mapping from name to catalog when calling <code>messages::do_open</code></td>
<td>704</td>
</tr>
<tr>
<td>mapping header name to header or external source file</td>
<td>20</td>
</tr>
<tr>
<td>mapping physical source file characters to basic source character set</td>
<td>15</td>
</tr>
<tr>
<td>mapping to message when calling <code>messages::do_get</code></td>
<td>704</td>
</tr>
<tr>
<td>meaning of <code>asm</code> declaration</td>
<td>167</td>
</tr>
<tr>
<td>meaning of attribute declaration</td>
<td>136</td>
</tr>
<tr>
<td>negative value of character literal in preprocessor</td>
<td>399</td>
</tr>
<tr>
<td>nesting limit for <code>#include</code> directives</td>
<td>401</td>
</tr>
</tbody>
</table>
number of threads in a program under a freestanding implementation, 11
umerical values of character literals in #if directives, 399
parameters to main, 58
passing argument of class type through ellipsis, 94
physical source file characters, 15
presence and meaning of native_handle_type and native_handle, 1129
rank of extended signed integer type, 82
representation of char, 71
required libraries for freestanding implementation, 4
result of exception::what, 454
result of inexact floating-point conversion, 80
result of right shift of negative value, 116
return value of bad_alloc::what, 449
return value of bad_cast::what, 452
return value of bad_exception::what, 454
return value of bad_typeid::what, 452
return value of char_traits<char16_t>::eof, 615
return value of char_traits<char32_t>::eof, 616
return value of type_info::name(), 451
search locations for "" header, 400
search locations for <> header, 400
semantics of extern linkage|hyperpage, 167
semantics of linkage specification on templates, 315
semantics of non-standard escape sequences, 25
set of blank characters defined by regex_traits::isctype, 1076
signedness of char, 71
signedness of plain integral bit-field, 223
sizeof applied to fundamental types other than char, signed char, and unsigned char, 106
stack unwinding before call to std::terminate(), 391, 396
start-up and termination in freestanding environment, 58
string resulting from __func__, 193
support for extended alignment, 528
support for over-aligned types, 106, 559, 565
supported multibyte character encoding rules, 614, 617
text of __DATE__ when date of translation is not available, 408
text of __TIME__ when time of translation is not available, 408
type of ios_base::streamoff, 1215
type of ios_base::streampos, 1215
type of ptdiff_t, 115
type of regex_constants::error_type, 1073
type of streamoff, 614
type of streampos, 614
type of u16streampos, 615
type of u32streampos, 616
type of wstreampos, 617
type of array::const_iterator, 741
type of array::iterator, 741
underlying source of random numbers for random_shuffle, 866
underlying type for enumeration, 152
use of non-POF function as signal handler, 460
value of ctype<char>::table_size, 672
value of character literal outside range of corresponding type, 25
value of multicharacter, 24
value of result of inexact integer to floating-point conversion, 81
value of result of unsigned to signed conversion, 80
value of wide-character literal containing multiple characters, 24
value of wide-character literal with single c-char that
is not in execution wide-character set, 24
values of a trivially copyable type, 69
Values of various ATOMIC_..._LOCK_FREE macros, 1112
whether get_pointer_safety returns pointer_safety::relaxed
or pointer_safety::preferred if the implementation has relaxed pointer safety., 593
whether time_get::do_get_year accepts two-digit
year numbers, 694
whether an implementation has relaxed or strict pointer
safety, 65
whether locale object is global or per-thread, 657
whether sequence pointers are copied by basic_filebuf
move constructor, 1047
whether sequence pointers are copied by basic_stringbuf
move constructor, 1036
whether source of translation units must be available
to locate template definitions, 16
whether values are rounded or truncated to the required precision when converting between
time_t values and time_point objects., 607
which functions in Standard C++ library may be recursively reentered, 429